Capitolo 001 - Prima di partire

Qui comincia l'avventura...

Primo inserimento: Settembre-Ottobre 2001

L'ATTREZZATURA

L'equipaggiamento di base per partire è presto detto.

Una certa conoscenza di C, forse non tanta: Cocoa è un *framework* (immagino ne parlerò a lungo poi), quindi ha già pronto un sacco di roba, non credo ci sarà bisogno di una conoscenza estrema di C o Objective C (che al momento attuale ignoro) per venirne fuori in maniera dignitosa.

Esperienza di programmazione: averne è più facile di quanto si pensi. È sufficiente avere una buona logica di ragionamento, piuttosto che conoscere tecniche strane. Poche e chiare idee su ciò che si vuole ottenere.

Queesto sito NON intende quindi discutere di programmazione e di linguaggio C, anche se di programmazione e di linguaggio C se ne parla in abbondanza. In altre parole, i requisiti minimi per giovarsi di tutto ciò che è scritto qui consistono nella conoscenza delle nozioni fondamentali di programmazione e in una buona infarinatura sul linguaggio C. Se non possedete questi requisiti, troverete difficili molti passi. Vi consiglio quindi di appropriarvi di qualche risorsa (in rete, o anche in libreria) su programmazione e linguaggio C.

Inglese da masticare: questo è importante. Immagino ci saranno molte cose da leggere ed approfondire, e dall'inglese non si scappa. Ciò che aiuta è che non dobbiamo conversare in inglese, basta leggerlo, cosa mooolto più semplice. Un dizionario a portata di mano può servire.

Buona volontà: non la prometto nemmeno io. Ignoro l'impegno richiesto, ignoro quanto tempo potrò dedicare (poco...). Le tappe del viaggio saranno a misura (mia e vostra, se qualcuno viene con me).

IL MEZZO DI TRASPORTO

Parto leggero: Mac Os X installato su un powerbook G3-500; con un iMac dovreste essere già a posto; i Developer Tools installati, che dovrebbero bastare per editare, compilare e completare applicazioni (è facile: basta usare il terzo disco che viene con Mac Os X: nel primo c'è X, nel secondo il 9.1, nel terzo, appunto, i dev tools). Una connessione internet per recuperare informazioni dal sito di Apple o dovunque se ne trovino. Fine.

***Agosto 2005**: Siamo arrivati alla versione 10.4.2 di Mac OS X, mentre l'ambiente di sviluppo è giunto alla versione 2.1. Il sistema operativo è distribuito su un DVD. Per installare XCode dovete cercare l'applicazione di installazione all'interno del DVD*

stesso. XCode non è normalmente installato con il sistema operativo. Alternativamente, XCode è scaricabile gratuitamente dal sito degli sviluppatori Apple. Si deve entrare a far parte della Apple Developer Connection. Esiste una modalità di iscrizione gratuita che permette di accedere alle risorse on-line, ma non dà diritto a materiale, sconti o altre agevolazioni.

LA STRADA

La strada è quanto mette a disposizione Apple in forma più o meno gratuita con i Developers Tools, i file di Aiuto installati, documenti aggiuntivi, tutorial, eccetera, eventualmente supportati da qualcos'altro che si trova in rete. Non avrò scrupolo di leggere i documenti forniti, farne un sunto in italiano e propinarvelo. Qui e lì aggiungerò pezzi derivati dalla mia esperienza, eventualmente ispirati a qualche altro libro o documento. Se qualche pezzo vi parrà la riproposizione di qualcosa che avete già letto altrove, non dite che non vi avevo avvertito. Ah, ho comprato anche il libro *Learning Cocoa* della O'Reilly, scritto direttamente da Apple, ma non ho ancora idea quanto servirà.

LA META

Raggiungere la meta non è l'obiettivo di questo viaggio. L'obiettivo di questo viaggio è capire come usare Cocoa per scrivere applicazioni. No, ma che dico.

L'obiettivo di questo viaggio è di divertirsi programmando Cocoa.

Magari faticando e passando notti davanti a schemi di computer. E badate: non voglio imparare Cocoa per diventare un programmatore Macintosh, o scrivere applicazioni da vendere poi come shareware o vero e proprio programma commerciale. L'imparare Cocoa sarà per me un atto di pura inutilità, artistico, avulso dalla logica del profitto. Voi che leggete avrete anche altri obiettivi, ma sono vostri, ne ho il massimo rispetto, ma sono appunto vostri. Tenete presente però il mio obiettivo quando viaggiate con me (e vedrete che indulgo in cose futili).

Stavo divagando: la meta. Poiché *imparare a programmare in Cocoa* è un po' generico, si rischia di studiare tanto, di applicare poco, e di applicare quello che si è imparato su esempi e non su problemi reali. Quindi, mi prefiggo un obiettivo: la realizzazione di un programma che non sia un giocattolo o fine a se stesso, ma che possa potenzialmente servire a qualcosa. Questo significa che, ogni volta che ho imparato ed applicato qualcosa, il passo successivo sarà di imparare qualcos'altro propedeutico all'obiettivo finale.

Ciò che intendo realizzare è un programma per la catalogazione di cd, hard disk, e memorie di massa in genere (temo di non poter provare i floppy sul powerbook...). Chiamerò il tutto *progetto CDcat*, giusto per dare un nome fantasioso.

Ora, non è che il mondo ha un estremo bisogno di un programma del genere, ma il progetto mi sembra abbastanza semplice da poter essere conseguito, ed abbastanza complesso da rivelarsi interessante. Cosa ha di interessante un CDcat? Beh, intanto si può cominciare a leggere un disco (e quindi ad accedere al file system) e mettere tutti i dati in una finestra. E già qui sarebbe un buon punto. Poi, questi dati li salviamo in un

file. Dopo, si impara a leggere più di un cd, e mettere i dati all'interno di un catalogo. Dopo, si gestiscono contemporaneamente più cataloghi. E le ricerche? Dove le mettiamo le ricerche di file all'interno del catalogo? Di più, proviamo anche a stampare delle copertine!

Insomma, prevedo di dover utilizzare molte delle caratteristiche interessanti di un programma, e poi metterò dentro tutto quello che mi verrà in mente (o che qualcuno suggerirà... che so, andare in rete... cose del genere).

Pronti? Via.

Capitolo 002 - Il Sistema Operativo

Vado via veloce, giusto per avere un'idea...

Sorgenti: Leggo, traduco e riassumo dai documenti OSX Technology Overview e Cocoa Fundamentals di Apple.

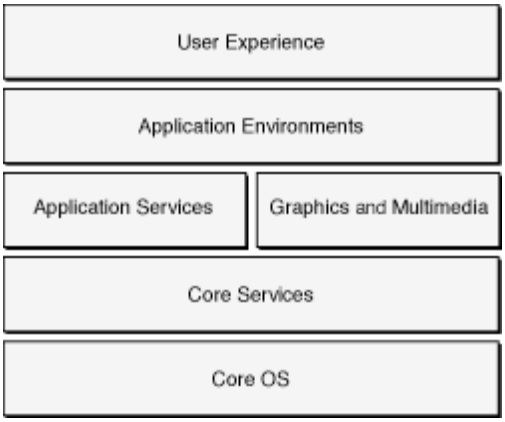
Primo inserimento: Settembre-Ottobre 2001. Parziale riscrittura: 24 settembre 2006.

System Overview

Programmare per Mac Os X richiede almeno una minima conoscenza del sistema operativo. Senza pretendere di essere esaustivo, corretto e soprattutto particolarmente originale, ho saccheggiato a piene mani il documento Mac Os X Technology Overview, cui rimando i più coraggiosi.

Cipolle

Il tipico modello cognitivo utilizzato per esporre architetture informatiche molto strutturate è la classica cipolla. Come in una cipolla ci sono diversi strati, uno sopra l'altro, che partono dal cuore della cipolla e via via crescono verso l'esterno, l'ambiente operativo di Mac Os X si può descrivere come una cipolla.



I diversi strati, chiamati `layer`, partono dal cuore. Il primo layer, quello più interno, è infatti chiamato Core OS: qui si trova il `kernel`, i driver dei vari dispositivi ed i comandi di basso livello del sistema operativo. Qui risiede `Darwin`, una sorta di cappello per tutta una serie di tecnologie, che può essere considerato a tutti gli effetti un sistema operativo della famiglia Unix, nella sua variante BSD. Darwin è open-source (nel senso che il codice sorgente è disponibile a tutti per miglioramenti e modifiche) ed è utilizzabile anche da solo. Tra le varie tecnologie incorporate in Darwin c'è `Mach`, elemento di base di tutta la faccenda, che provvede ad una serie di servizi di base (memoria protetta, gestione del multi-tasking, memoria virtuale, supporto real-time) senza i quali nulla funzionerebbe; c'è il `sottosistema BSD`, che fornisce comandi di basi per la gestione dei processi, della politica degli accessi ed il supporto di base per la

rete; ci sono i driver dei dispositivi (mouse, tastiera, eccetera); c'è il supporto dei vari file system (molti, e non solo quelli tipici di Apple, ma anche Unix e Windows, anche se mi irrita l'impossibilità di scrittura su NTFS); c'è il supporto per i servizi di base della rete.

Il layer successivo è chiamato `Core Services`. Si tratta in pratica di una libreria di funzioni a disposizione di tutti (non solo del sistema operativo, ma anche e soprattutto delle applicazioni) per la gestione di tipi di dati astratti (stringhe, ad esempio, ma anche URL, dati XML, le Preferenze, eccetera). è (dovrebbe essere) altamente ottimizzata, ed alla base di ogni possibilità di scambiare informazioni tra applicazioni e sistema operativo. In pratica, all'interno dello strato Core Services si trovano tutte quelli funzioni che non hanno a che fare con l'interfaccia grafica. Lo strato cerca di dare un'aspetto più pulito e maneggiabile alle funzioni messe a disposizione dal kernel. Se ad esempio nel kernel ci sono tutte le chiamate per gestire i file (aprire un file, chiudere un file, eccetera, ad un livello molto basso), nei Core Service si trova il File Manager, che fa le stesse cose in maniera più pulita, e combina opportunamente il tutto per rendere più facile la programmazione.

Il passo successivo è in realtà bipartito, perché si trovano due layer in parallelo. Il primo si chiama `Application Services`, e raccoglie tutta una serie di tecnologie che possono essere utilizzate in maniera uniforme da chiunque. In questo modo, applicazioni diverse riutilizzano gli stessi componenti software per eseguire le stesse cose (che so, visualizzazione di pagine HTML attraverso il WebKit, sistemi di ricerca attraverso il Search Kit, cose del genere), con grande beneficio per l'utente finale, che non deve imparare modi diversi per fare le stesse cose. Il secondo è chiamato `Graphics And Multimedia`, e come si evince dal nome raccoglie tutte le tecnologie per la visualizzazione 2D e 3D delle informazioni, compresa Quicktime (e quindi anche tutto l'audio). Si tratta di un layer bello corposo, in quanto rientrano appunto QuickTime, Open GL, ma anche Core Audio e Core Video, i sistemi di stampa, ColorSync, eccetera. Questi due layer forniscono in pratica tutto ciò che occorre ad una applicazione per interagire con un utente a livello schermo, audio e video.

L'ultimo layer di interesse è finalmente lo `Application Environment`, che chiamerò in italiano l'ambiente applicativo, all'interno del quale si possono sviluppare le applicazioni Macintosh. È da qui che, per la maggiornanza degli sviluppatori, si parte per lo sviluppo dell'applicazione. In realtà non esiste un solo ambiente applicativo, ma ne esistono diversi, a seconda delle tecnologie utilizzate: Carbon, Cocoa, Java, Applescript, BSD/X11 e WebObjects (una volta, e per qualcuno è ancora disponibile, esisteva un ulteriore ambiente, chiamato Classic, dove programmare per Max OS 9). Questi ambienti applicativi forniscono i diversi mattoncini con cui si possono costruire le applicazioni. Un programmatore più decidere di usare i mattoncini di tipo BSD per fare la propria applicazione, un altro può scegliere i mattoncini di tipo AppleScript, un terzo Carbon. Io invece ho scelto i mattoncini di tipo Cocoa (ma ci saranno invasioni di campo, o meglio, di vari altri campi).

Costruire un ambiente applicativo è lo scopo finale di un sistema operativo. Un sistema operativo copre fondamentalmente i dettagli operativi del calcolatore sul quale risiede, nascondendo appunto le varie differenze hardware, interagendo con il mondo esterno, e mascherando il tutto attraverso una API, Application Programmatic Interface (qualcosa del genere), in pratica una collezione di funzioni che permettono il funzionamento delle applicazioni. Di più: chiamare funzioni di questa API è l'unico

modo che hanno le applicazioni per interagire con il calcolatore.

Detto questo, appare chiaro che Mac Os X è un sistema operativo molto ricco: non fornisce un solo ambiente applicativo, ma sei! In realtà, mentre Carbon, Cocoa, Java e AppleScript sono in grado di costruire applicazioni sostanzialmente simili, quelle con la tipica interfaccia Macintosh, gli ambienti BSD/X11 e WebObject sono adatti solo per particolari applicazioni.

GLI AMBIENTI OPERATIVI

L'ambiente BSD/X11

L'ambiente BSD è un ambiente tipico di Unix, dove si possono scrivere applicazioni senza interfaccia grafica, rispettando le interfacce Posix e FreeBSD. Non avendo interfaccia grafica, sono applicazioni che sono lanciate da linea di comando (ad esempio da Terminale) ed operano generalmente con file (che in Unix è un concetto piuttosto esteso). Non si deve sottovalutare la potenza di questo ambiente, visto che con strumenti sviluppati all'interno di questo ambiente si possono fare moltissime operazioni che si fanno con molta più grazia con una interfaccia grafica (potrebbe sorprendervi il fatto che si può anche navigare il web all'interno di questo ambiente...).

L'ambiente X11 è uno strato di librerie e meccanismi per aggiungere una interfaccia grafica (uno standard in ambiente Unix, piuttosto portabile tra diverse realizzazioni) alle applicazioni dell'ambiente BSD. L'eleganza della programmazione in X11 è per certi versi ancora insuperata a vent'anni (più o meno) dalla sua nascita. Molti porting da Unix o applicazioni multi-piattaforma (ad esempio, allo stato attuale, OpenOffice) utilizzano in massima parte questo ambiente.

L'ambiente Classic

L'ambiente Classic non esiste più. Era l'ambiente all'interno del quale funzionano le vecchie applicazioni Macintosh. Non c'è ragione per cui qualcuno debba sviluppare in ambiente Classic. Sul mio iMac Intel questo ambiente non esiste.

E' un po' buffo trattare così male l'ambiente Classic, che nel bene e nel male ci ha accompagnato dal 1984 al 2000... cosa volete, è la vita!

L'ambiente Carbon

Carbon è l'insieme di tutte le chiamate possibili al sistema operativo, ripulite da ogni complicazione dovute a Unix, comprensive di grafica, file system, network, eccetera. È in pratica la API nuda e cruda. Utilizzando direttamente le chiamate messe a disposizione da Carbon è possibile scrivere applicazioni perfettamente adeguate a funzionare con Mac Os X. Il problema di Carbon è che bisogna lavorare a basso livello, reinventando la ruota ogni volta che si riproduce un meccanismo tipico delle applicazioni. Il vantaggio di Carbon è che, lavorando a basso livello, non ci sono orpelli inutili e si raggiunge un elevato grado di efficienza. Carbon è l'ambiente applicativo fondamentale del Mac Os X, quello cui rivolgersi per scrivere le applicazioni standard, con i buoni vecchi linguaggi di programmazione (che so, C e C++).

L'ambiente Java

Java è un ambiente di sviluppo per il linguaggio omonimo. Si possono così scrivere sia applicazioni in puro Java (facilmente trasportabili in altri ambienti), sia programmare in Java nativo producendo applicazioni per il Mac Os X, perdendo la portabilità dell'applicazione.

L'ambiente Applescript

Più che un ambiente a se stante, AppleScript permette di scrivere applicazioni complete utilizzando molte tecnologie Cocoa, tanto da confondersi spesso con questo ambiente. Di fatto, gli strumenti di base per la scrittura di applicazioni in AppleScript sono gli stessi utilizzati da Cocoa.

L'ambiente WebObjects

WebObjects è un ambiente applicativo che comprende una serie di strumenti e costrutti adatti alla realizzazione di applicazioni su Web. Si possono realizzare sistemi con dati dinamici, interagendo con database e altre applicazioni, e costruire servizi web tramite SOAP, XML e tecnologie similari. Mi pare di ricordare che lo AppleStore è interamente realizzato con WebObjects.

L'ambiente Cocoa

Siamo finalmente arrivati a Cocoa.

Cocoa è una libreria di pezzi di codice che fanno la maggior parte del lavoro ripetitivo legato alle applicazioni per Mac Os X. Dovrei dire che Cocoa è una collezione di classi (un framework, come si dice) sulla quale basare lo sviluppo di applicazioni per Mac Os X. L'affermazione è perfettamente lecita se uno capisce cosa significa classi, e l'associa alla programmazione orientata agli oggetti.

Finché non parlo di questa programmazione object oriented, occorre spiegare in altro modo Cocoa.

Cocoa è una estesa libreria di componenti software che si possono utilizzare per costruire applicazioni per Mac Os X. In realtà Cocoa si pone ad un livello un po' più alto della API di Carbon: se Carbon è l'insieme dei vari mattoncini, Cocoa è un insieme di blocchi pre-fabbricati con i quali costruire in maniera più rapida le applicazioni per Mac Os X. Come gli elementi prefabbricati, è molto facile costruire software mettendo assieme nel giusto ordine gli elementi: basta solo aggiungere qui e lì un po' di collante. Altre volte invece, volendo costruire edifici particolari, gli elementi prefabbricati dovranno essere ritoccati qui e lì per adattarli alle nuove esigenze (che so, aprire una finestra qui, allargare quella porta là, cose del genere). Potrebbe essere viceversa molto difficile fare cose esotiche, grattacieli, palazzi barocchi, architetture bizzarre. Per queste cose, non si può che usare Carbon. Tuttavia, la ricchezza espressiva di Cocoa dovrebbe essere così grande e l'interazione coi livelli inferiori così completa, da rendere difficile raggiungere i limiti operativi di Cocoa.

Tra gli elementi prefabbricati presenti in Cocoa, ci sono molte cose: ci sono ad esempio, già pronti per l'uso, tutti gli elementi dell'interfaccia utente, già con aspetto aquatico: pulsanti, check box, campi di testo, eccetera. C'è un prefabbricato che rappresenta un Documento: basta solo colorarlo con le vernici giuste ed abbiamo già

pronto un meccanismo per presentare e salvare i dati. Cose del genere. Di elementi prefabbricati ce ne sono veramente tanti. Proprio questo è uno dei problemi di Cocoa: ci sono talmente tanti elementi che conoscerli tutti è quasi impossibile. Non si può pensare di studiarli tutti e poi cominciare a sperimentare. Conviene impadronirsi di qualche elemento, e poi cominciare subito. Quando avrò bisogno di un qualche costrutto particolare, prima di farmelo da solo, converrà impiegare un po' di tempo per vedere se non è già disponibile in Cocoa sotto qualche forma. Ci sono ottime possibilità che ci sia già disponibile non dico quello che mi occorre, ma almeno qualcosa di simile (e che potrà essere modificato alla bisogna).

La maggior parte dei nomi degli elementi di Cocoa ha un nome che comincia con 'NS': una stringa ad esempio è un elemento di tipo NSString. Ho il sospetto che questa convenzione derivi dal fatto che Cocoa è l'erede dell'ambiente applicativo OpenStep di NextStep (ricordate? Next era la società fondata da Steve Jobs al momento della sua uscita da Apple; era circa il 1987 o giù di lì). Sono quindi circa quindici anni che le idee e la realizzazione di Cocoa sono in giro, per cui dovrebbe essere una cosa abbastanza assestata e corretta.

Si possono utilizzare diversi linguaggi di programmazione per lavorare con Cocoa, ma il predestinato è Objective-C (ObjC). ObjC non è parte di Cocoa, ma Cocoa è basato fortemente su tale linguaggio; gli oggetti di Cocoa sono scritti appunto in ObjC.

ObjC comprende al suo interno il C standard. In pratica, è il C standard con una serie di costrutti aggiuntivi, e si possono mescolare senza problemi istruzioni C ed istruzioni ObjC. Questo significa che in ObjC si possono utilizzare anche gli ambienti BSD/X11 e Carbon, mescolandoli gli uni con gli altri. Benché non una cosa pulitissima, è di una comodità spaventosa...

Ciò che si ottiene scegliendo di lavorare con Cocoa è un insieme di funzionalità già disponibili senza scrivere nemmeno una riga di codice. Senza far nulla, ad esempio, si può avere un'applicazione funzionante con una sola riga di codice (per altro, scritta automaticamente dall'ambiente di sviluppo). Ora, l'applicazione non farà molto, anzi, praticamente nulla. Però si avrà un'applicazione, che apre una finestra, presenta dei menu, si comporta come tutte le altre applicazioni, può stampare (una pagina bianca, va da sé) e perfino rispondere a semplici comandi AppleScript.

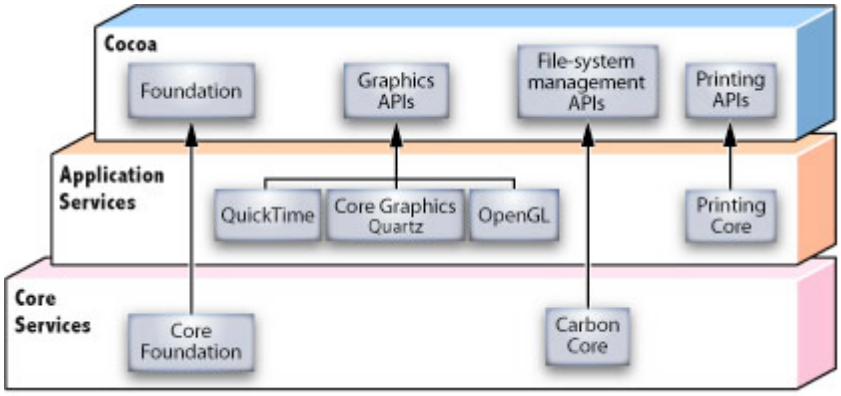
A disposizione ci sono poi dei meccanismi per la costruzione di applicazioni basate su finestre e per applicazioni basate su documenti (la distinzione sarà chiara più avanti), tutti gli elementi dell'interfaccia utente quali pulsanti, menu, finestre di varie fogge, pulsanti di spunta, visualizzatori di testi e di immagini, cose così. Ma poi anche meccnismi per interagire con il sistema, per gestire file, per stampare. per gestire elementi multimediali, interfacce di rete.

Cocoa viene fornito sotto forma di una coppia di Framework; dirò velocemente che un framework è una collezione di elementi con uno scopo comune. Ad esempio, il framework 'pareti' raccoglie tutti quei prefabbricati utili per la costruzione delle pareti, distinto dal framework 'tetto' che raccoglie tegole, solai ed altri oggetti del genere.

I due framework sono il `Foundation Kit` (che raccoglie in pratica le funzioni svolte dal layer Core Services) e l'`Application Kit` (le funzioni dei layer Application Services e Graphic and Multimedia Services). Questi due framework cercano di astrarre

i layer inferiori a Cocoa e fornire un ambiente pulito ed ordinato. Come già detto, i framework dovrebbero bastare alla maggior parte dei programmatori. Tuttavia, se ritenuto necessario, è possibile scendere di livello e sporcarsi direttamente le mani accedendo ai layer inferiori (ad esempio, comandare direttamente OpenGL per ottimizzare la resa a video di una animazione). Tutto è facilitato dal fatto che le API sono scritte in C e ObjC ne è una estensione.

L'importanza di questi due framework si può riassumere nelle seguenti affermazioni: non si può realizzare una applicazione (con finestre, menu, eccetera) con Cocoa senza utilizzare lo Application Kit. E non si può scrivere un codice funzionante Cocoa senza utilizzare il Foundation Kit.



FOUNDATION FRAMEWORK

Il Foundation Framework raccoglie in pratica tutti gli elementi di Cocoa che non hanno un corrispettivo nell'interfaccia utente. Il suo scopo è triplice: estendere le funzionalità di ObjC fornendo servizi di base per la programmazione object oriented (come vedremo poi); fornire servizi di base riguardanti il sistema operativo (e quindi memoria, controllo di esecuzione dei programmi, rete, eccetera); fornire supporto per il file system e le stringhe.

Con qualche dettaglio in più, gli elementi si possono raggruppare in diverse famiglie, come le seguenti:

- gestione di oggetti che hanno un valore, come i numeri, le date, le ore, insomma per gestire dati a livello astratto e trasformarli secondo diversi meccanismi ;
- gestione delle stringhe, loro costruzione, formattazione, eccetera. Da notare che Cocoa gestisce caratteri in accordo alla specifica Unicode, e quindi è in grado di gestire con facilità alfabeti non latini ed altre complicazioni del genere;
- gestione di collezioni di dati, sotto forma di vettori, insiemi, dizionari;
- servizi del sistema operativo, come la gestione del file system, la gestione di dati attraverso URL, la comunicazione tra processi e la gestione dei thread e dei task;
- persistenza degli elementi, con le funzioni necessarie per l'archiviazione e il recupero dei dati;
- servizi di scripting, per integrare AppleScript all'interno dell'applicazione;
- servizi ausiliari per ObjC, Meccanismo delle notifiche, oggetti distrubuiti e comunicazione remota, eccetera.

APPLICATION FRAMEWORK

Lo application framework contiene tutti gli elementi utili per realizzare l'interfaccia grafica guidata dagli eventi (cioè, che reagisce a mouse e tastiera): ci sono finestre, pulsanti, menu, barre di scorrimento e cose del genere. La cosa bella di questo framework è che si occupa di tutti i dettagli noiosi (ridisegno delle finestre, dei controlli, gestione dei testi, cose del genere); è composta da una quantità impressionante di elementi, che possiamo dividere in tre grossi gruppi:

- elementi dell'interfaccia utente vera e propria;
- caratteristiche aggiuntive, come i font, le immagini, i colori, i suoni;
- servizi aggiuntivi di interfaccia, come il supporto per drag'n'ndrop, gli appunti per il copia/incolla, la stampa, la gestione dei documenti.

C'è perfino un elemento che aiuta nel controllo ortografico del testo (quest'oggetto mi interessa tanto che penso di studiarlo comunque, anche se non vedo come potere entrare in un programma che cataloga cd).

ALTRI FRAMEWORK

Sono disponibili altri framework, che forniscono strumenti aggiuntivi per le applicazioni, che non sono però sempre necessari (e quindi mantenuti separati). Un elenco, parziale ed incompleto, potrebbe essere:

- Core Data, per la gestione di basi di dati;
- Sync Services, per la gestione di elementi da sincronizzare (ad esempio con diavolerie elettroniche attraverso Bluetooth);
- Address Book, per gestire i contatti della Rubrica Indirizzi all'interno dell'applicazione;
- Preference Panes, per scrivere un pannello delle preferenze;
- Screen Saver, per scrivere un salvaschermo;
- WebKit, per visualizzare elemento scritti in HTML.

Detto questo, partiamo dall'inizio: la programmazione object oriented e il linguaggio Objective C.

Capitolo 003 - Object Oriented Programming

Dove si introduce un paradigma.

Sorgenti: Un po' copiato, un po' inventato...

Primo inserimento: Settembre-Ottobre 2001

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

La programmazione orientata agli oggetti o, come si dice, la programmazione object oriented *OOP*, è un modo diverso da quello classico di considerare la programmazione dei computer.

Ci sono due metodi fondamentali per interagire con le cose. Il primo metodo specifica prima l'azione, poi la cosa (l'oggetto) sulla quale eseguire l'azione. Se parliamo di finestre, quelle reali, si deve lavorare così: `Aprire: Finestra`. Se poi volete passare alla porta, abbiamo un messaggio simile: `Aprire: Porta`. Il meccanismo, una volta capite le basi, funziona piuttosto bene.

Ma c'è un altro metodo: proviamo a dire prima l'oggetto, e poi l'azione che si vuole compiere su questo oggetto: `Finestra: Aprire`. Sembra un passaggio da poco, ma abbiamo spostato l'obiettivo dalle azioni agli oggetti.

Essere incentrato sulle azioni, sulle funzioni, sulle operazioni, è un approccio classico dei calcolatori. Dopotutto, il linguaggio macchina dei calcolatori è effettivamente qualcosa del tipo: indicazione dell'operazione, dati sui quali l'operazione deve essere eseguita. Si profila così una divisione piuttosto netta tra le operazioni e i dati. Mi ricordo che, nell'età giurassica quando studiavo queste cose, uno dei libri fondamentali del giovane informatico era un testo di Wirth (il professore -svizzero, mi pare- che ha inventato il linguaggio Pascal) intitolato: *Algoritmi + strutture dati = programmi*. Come dire, date le operazioni da eseguire , e i dati sui quali eseguirle, abbiamo fatto l'applicazione. Indico questo modo di programmare procedurale, oppure orientato alle procedure, nel senso che sono le sequenze di operazioni a *fare* il programma.

Parallelamente a questa scuola di pensiero se ne sviluppava un'altra, all'inizio un po' sotterraneamente e solo negli ambienti accademici, e poi con sempre maggiore diffusione, si proponeva la scuola della programmazione orientata agli oggetti; qui il punto centrale del discorso è l'oggetto, inteso come un ente software non meglio definito con alcune importanti caratteristiche: auto-contenuto, disponibile al colloquio e ad essere modificato senza sforzo eccessivo. Un oggetto è auto-contenuto in quanto contiene al suo interno (*incapsula*, si dice) sia i dati sia le operazioni che su questi dati operano. Un oggetto è quindi un ente software non semplicemente passivo, come può esserlo un numero, ma qualcosa un po' più attivo, un numero in grado di eseguire da solo le operazioni che interessano.

Per chi, come me, ha imparato a programmare con l'approccio procedurale, la OOP

all'inizio appare un po' bizzarra. Ma poi, una volta compresa, diventa molto più naturale. Addirittura, adesso mi trovo ad usare tipici linguaggi di programmazione in ambito procedurale (il C, per capirci), distorcendoli in modo tale da farli diventare object oriented. E comunque alcuni concetti alla base della OOP sono indispensabili anche in ambito procedurale per essere un buon programmatore.

DIVERSI MODI DI PROGRAMMARE

La programmazione per oggetti, o programmazione orientata agli oggetti OOP è una metodologia di programmazione la cui caratteristica principale è il concetto di oggetto. Nella OOP tutto si basa, in maniera uniforme, sul concetto di *Oggetto*: ogni singolo programma, al più semplice al più complesso, si basa su un insieme di Oggetti che interagiscono tra di loro.

Un oggetto non è altro che un insieme di dati, assieme alle procedure necessarie per operare su questi dati; l'operazioni sui dati di un oggetto è esclusivo delle procedure facenti parti dell'oggetto.

Per apprezzare la differenza della OOP rispetto alla programmazione *classica*, partiamo dall'inizio.

Quando si deve risolvere in termini informatico un problema, a meno di casi molto semplice, è difficile riuscire ad avere qualcosa di compiuto e maneggevole. È giocoforza suddividere il problema in un insieme di sotto-problemi, di natura più semplice. Tradizionalmente, l'approccio utilizzato è stato quello di una suddivisione di tipo procedurale o funzionale. Si tratta di dividere le operazioni da svolgere in un insieme di moduli funzionali. L'intero programma è dunque composto da un insieme di moduli funzionali che operano, sequenzialmente o in parallelo, su strutture dati predefinite.

Supponiamo che il problema sia di leggere, ordinare e stampare in insieme di dati (che so, una lista di indirizzi). Esiste allora la struttura dati generali (una rappresentazione opportuna della lista di indirizzi) e di tre moduli funzionali separati, uno per la lettura dei dati ed il riempimento della struttura, uno per effettuare l'ordinamento di questi dati, uno per la stampa in bella copia o presentazione dei dati. Ognuno di questi tre moduli funzionali può essere diviso in altri moduli, ad esempio la presentazione dei dati può avvenire secondo diverse modalità, a video e a stampa; ogni tipo di presentazione è svolto da un separato modulo funzionale.

Questa metodologia classica o *procedurale* nasce e si afferma con i linguaggi di programmazione strutturata, come il Pascal ed il C. Del resto, la netta separazione tra i comandi e i dati è nella natura dei processori, le cui istruzioni di linguaggio macchina sono proprio costruite come comando e dati sui quali opera il comando stesso. Questi metodi procedurali, estremamente potenti, efficaci e *naturali* (per il processore), entrano in difficoltà di fronte a questioni di stampo *ingegneristico*: l'adattabilità a mutate condizioni e la riusabilità delle funzioni.

Un programma costituito da una base dati comune e dai blocchi funzionali adatti a lavorare su questa base dati diventa facilmente intrattabile quando si modifichi, anche leggermente, la struttura della base dati. In questo caso, una modifica sulla base dei dati ha impatto sull'intero programma, in quanto ogni blocco funzionale deve essere (potenzialmente) modificato, per tenere conto appunto di questa modifica. Viene da sé

che è molto difficile riutilizzare uno dei blocchi funzionali (ad esempio, il blocco di ordinamento) sopra un'altra base dati, proprio perché l'ordinamento è tagliato sopra quella determinata base dati.

LE BASI DELLA OOP

Di fronte a questo metodo di progettazione, si è nel tempo affermato un diverso metodo, appunto OOP. In questa metodologia, la scomposizione non si basa più sull'operazione da eseguire, ma l'oggetto, inteso come modello dell'entità sulla quale si opera. Il programma è adesso costituito da un insieme di entità interagenti, ciascuna provvista di un struttura dati e dell'insieme di metodi adatti a manipolare quella struttura dati. Questi metodi sono inoltre l'unico modo con cui interagire con l'oggetto. Ciascun oggetto dunque incapsula i propri dati e ne difende l'accesso diretto da parte dell'esterno: i cambiamenti del mondo esterno non influenzano i dati all'interno, e viceversa, i dati interni non influenzano i dati esterni. Ne segue una netta separazione tra l'interno e l'esterno, che permette di recuperare i due punti deboli dell'approccio procedurale. Modifiche anche pesanti di strutture dati non hanno impatto sul resto del sistema, una volta garantita l'uniformità di comportamento dell'oggetto. A questo punto, per utilizzare un oggetto, basta sapere solamente quali sono le operazioni che si possono svolgere su di esso per poterlo utilizzare in contesti anche molto diversi tra loro.

La OOP è un modo diverso di pensare e di progettare sistemi. Non si contrappone totalmente al metodo procedurale (un singolo oggetto, alla fin fine, si costruisce con un approccio procedurale), ma è un metodo alternativo per la realizzazione di programmi complessi. Del resto, scomporre i problemi in oggetti, e poi questi oggetti in oggetti più semplici, è più vicino al mondo con cui l'essere umano interagisce con il mondo reale, per cui la OOP dovrebbe essere una strategie di risoluzione dei problemi più semplice e naturale.

In fin dei conti, la nostra lingua parla per soggetto e predicato, ovvero per oggetto ed operazione svolta o subita da questo oggetto. È il soggetto (informaticamente, l'oggetto) la base di ogni discorso.

La OOP non si limita a modificare l'oggetto della scomposizione: l'uso degli oggetti presenta molte altre caratteristiche interessanti che facilitano le metodologie di programmazione. Un linguaggio orientato agli oggetti ha almeno le seguenti caratteristiche:

Oggetti
: È basato sul concetto di oggetto, e deve fornire quindi gli strumenti per definire, creare, gestire e distruggere oggetti.

Ereditarietà
: Fornisce la possibilità di costruire nuove categorie di oggetti, riutilizzando le caratteristiche di altri oggetti e definendone di nuove; dopotutto, un'automobile è una versione specializzata dell'oggetto autoveicolo.

Polimorfismo
: Consente di trattare allo stesso modo (chiamare con lo stesso nome procedure che eseguono lo stesso compito) oggetti differenti, e di avere la possibilità di mantenersi abbastanza generici nel farlo (avere oggetti non tipizzati). Dopotutto, l'operazione di rifornimento si fa su di una automobile, su di un motorino e su di

una barca, sostanzialmente nello stesso modo, pur nella differenza degli oggetti che sono riforniti.

Binding dinamico

Consente l'uso incrementale degli oggetti: se il programmatore modifica una singola procedura, deve essere possibile ricompilare ed effettuare il caricamento di quella singola procedura e non dell'intero programma; in altre parole ancora, quale operazione sia effettivamente realizzata deve poter essere deciso all'ultimo momento utile, in maniera dinamica, perché nel frattempo qualcosa potrebbe essere cambiato (ma di questo concetto, qui mi rendo conto poco chiaro, se ne parlerà ancora).

INTERVALLO STORICO

Uno dei primi linguaggio OO ad essere creato è stato lo *Smalltalk* (correva l'anno 1972). Una particolare versione di Smalltalk era alla base del sistema operativo di Lisa, il progetto della Apple immediato predecessore del Macintosh. Solo verso la metà degli anni ottanta l'ondata OOP ha raggiunto i linguaggi di programmazione procedurali, portando a versioni orientate agli oggetti dei classici Pascal e soprattutto C. Più o meno contemporanei sono i due linguaggi OOP derivati dal C, ovvero il fortunato C++, utilizzato ormai dalla maggior parte dei programmatori, ed il meno fortunato Objective C, sul quale Cocoa si basa. Dal punto di vista dell'orientamento agli oggetti, tuttavia, ObjC è sicuramente il meglio riuscito, essendo una filiazione diretta dello Smalltalk: le estensioni del C che portano all'ObjC sono dirette applicazioni delle regole sintattiche dello Smalltalk. I puristi diranno invece che il C++ è un linguaggio *bastardo* che introduce concetti e costrutti OOP su uno strato essenzialmente procedurale, e riesce a mescolare malamente le due cose.

Capitolo 004 - Object Oriented Programming

Dove si conclude il paradigma.

Sorgenti: Copiare da una sola fonte è plagio, da tante, ricerca: un vecchio libro sullo Smalltalk, un po' dal libro Apple su Objective C.

Primo inserimento: Settembre-Ottobre 2001

IL NOME DELLE COSE

Un oggetto è composto indissolubilmente di una struttura di dati privata che descrive lo stato dell'oggetto e da un insieme di segmenti di codice, chiamati metodi, che accedono in lettura e scrittura alla struttura dati interna dell'oggetto. Utilizzare i metodi è l'unico modo per accedere alla struttura dati interna dell'oggetto. Tanto per dirne una, ciò non è vero per il C++, che può accedere tranquillamente e direttamente ai dati interni (perlomeno una parte).

L'insieme di tutti i metodi di un oggetto è la sua interfaccia verso il mondo esterno. Una rappresentazione abbastanza utilizzata per rappresentare un oggetto e i suoi metodi è quella di utilizzare un cerchio per racchiudere la struttura dati propria dell'oggetto, ed un guscio attorno ad esso suddiviso in tanti pezzetti, uno per ciascun metodo, per raffigurare l'interfaccia dell'oggetto.

Gli oggetti sono contrassegnati da un identificatore; nei linguaggi procedurali ogni variabile è identificata attraverso il nome della variabile stessa; ugualmente nella OOP gli oggetti hanno un nome attraverso il quale possono essere utilizzati. L'identificatore può essere una costante, vale a dire l'oggetto pre-esiste nella ambiente operativo del programma. Nella maggior parte dei casi, gli identificatori sono in realtà dei *puntatori* o dei *contenitori* degli oggetti cui si riferiscono.

Riassumendo. Un oggetto è fondamentalmente un tipo di dati astratto: consiste in una struttura dati ed in un insieme di operazioni che si possono svolgere su questi dati. Chi usa l'oggetto non deve conoscere la struttura interna, né come siano realizzate le funzionalità messe a disposizione. Tutto quello che occorre sapere per usare un oggetto è che cosa è e come usarlo: un identificatore che ce lo faccia riconoscere e un'interfaccia esterna disponibile a tutti.

QUESTIONE DI METODO

Per poter utilizzare un oggetto non c'è altra strada che attivare uno dei metodi messi a disposizione dall'oggetto. Questa operazione è chiamata inviare un **messaggio** all'oggetto. Un messaggio è quindi una richiesta inviata ad oggetto allo scopo di attivare una delle operazioni disponibili, realizzata attraverso uno dei metodi appartenenti all'interfaccia dell'oggetto.

L'invio di un messaggio coinvolge tre enti: il **ricevente**, l'oggetto **destinatario** del messaggio, identificato da un nome; un **selettore**, ovvero una stringa che identifica il metodo di cui si richiede l'attivazione; gli **argomenti**, opzionali, da passare come parametri al metodo.

Supponendo di avere un oggetto identificato da `LaMiaFinestra`, si può richiedere l'apertura della finestra tramite un messaggio del tipo

```
LaMiaFinestra apri
```

Dove `LaMiaFinestra` è l'identificatore della mia particolare finestra, ed `apri` il selettore del metodo. Se c'è anche una serranda, possiamo dire

```
LaMiaSerranda apri: 50%
```

In cui l'oggetto `LaMiaSerranda` è ancora aperta, ma questa volta esiste un argomento che dice di quanto deve essere aperta (la metà). Notiamo per inciso una caratteristica dei linguaggi OOP: lo stesso identificatore di metodo (la stringa 'apri') è utilizzata in due contesti diversi, applicata a due oggetti diversi. Non c'è in questo caso alcun conflitto di nomi, in quanto il primo `apri` si rivolge ad oggetti di tipo `finestra`, mentre il secondo ad oggetti di tipo `serranda`.

Questa 'confusione' di nomi risulterà molto utile quando si tratta di polimorfismo e binding. Il risultato dell'invio di un messaggio è sempre un oggetto. L'oggetto prodotto può essere utile oppure no. Nel secondo caso, non ci faremo alcun scrupolo a non utilizzarlo (e l'ambiente operativo lo distruggerà subito al posto nostro). Nel primo caso, invece, bisogna procurare un posto dove immagazzinare l'oggetto, ad esempio in una variabile creata apposta.

CLASSI ED OGGETTI

Gli oggetti sono divisi in vari **tipi**; come esiste un generico oggetto automobile dal quale poi si ricavano la mia auto, la tua auto, l'auto di tizio, eccetera, esiste un *modello* per gli oggetti di un certo tipo, la **Classe**.

La classe definisce il tipo degli oggetti, ovvero come l'oggetto viene costruito. Attraverso la classe, sono definite la struttura dei dati interni e i metodi afferenti all'oggetto. La definizione dei metodi richiede anche l'esplicita indicazione delle operazioni che i metodi eseguono: in pratica, la classe raccoglie il codice realizzativo di tutti i metodi dell'oggetto, che sono in comune tra tutti gli oggetti.

Anche la struttura dati è comune a tutti gli oggetti di una data classe, anche se ovviamente i valori contenuti all'interno di questa struttura dati sono diversi da oggetto ad oggetto. La classe automobile insomma contiene tutti metodi per descrivere ed

operare sulle automobili, ed il posto per i dati afferenti ad ogni singola automobile (il colore, la cilindrata, eccetera). Ogni singolo oggetto poi raccoglie nel proprio interno i valori (rosso, milledue, eccetera).

Un oggetto della classe Automobile si dice **Istanza** della classe. Una classe è una rappresentazione platonica di un oggetto. Quando in un linguaggio OOP mi serve un oggetto, devo **istanziare** una data classe, per dire che mi serve un oggetto fatto in un certo modo. L'ambiente operativo allora produce una copia della classe, e la fornisce sotto forma di oggetto, istanza della classe richiesta.

Una classe raccoglie dunque le proprietà comuni di un insieme di oggetti, e specifica il comportamento di tutti gli elementi. Un programmatore OOP trova nella classe il suo lavoro principale. Si tratta di definire la classe, la sua struttura dati, e definire compiutamente tutti i metodi che la classe è in grado di realizzare. In pratica, definisce un tipo di oggetto e i messaggi che tale oggetto è in grado di ricevere.

Una volta definita la classe, il programmatore (ma anche no, può essere un altro programmatore) passa ad utilizzare la classe, creando oggetti di quella classe e inviando loro i messaggi. La parte interessante è che nel costruire classi, non mi interesso di chi dovrà utilizzare la classe.

Sono un produttore di automobili: posso certamente presumere quali saranno le richieste dei clienti, ma tutto quello che posso fare è produrre un modello di automobile e rendere disponibile la sua interfaccia.

Viceversa, chi utilizza le classi, non è interessato (anzi, il linguaggio OOP glielo proibisce espressamente) ai dettagli interni di realizzazione della classe, ma semplicemente gli interessa sapere cosa fa quell'oggetto.

Sono allora un cliente di automobili: io voglio un'automobile per andare a fare gite fuori porta, e non mi interessano i dettagli costruttivi dell'auto, se non limitatamente alle questioni di interfaccia (quanto consuma il motore, il colore, se fa rumore, quanto costa, dettagli del genere). Ma che l'auto sia costruita in Italia assemblando parti costruite in Giappone, o che sia costruita in Thailandia utilizzando parti costruite in Brasile, la cosa non mi interessa (fatta salva, ovviamente, la qualità dell'interfaccia).

EREDITARIETÀ

Ogni oggetto appartiene ad una classe , che ne specifica il funzionamento. Una delle caratteristiche fondamentali della OOP è l'**ereditarietà**. Per spiegarla, faccio un esempio.

Abbiamo la Classe Automobile. Qualcuno l'ha scritta per noi, e quindi abbiamo già a disposizione tutto quello che ci serve per viaggiare in automobile. Però adesso vogliamo una macchina speciale, un Taxi.

Un Taxi è tale e quale ad un'automobile, però ha qualche caratteristica diversa (il colore, ad esempio) ed in più (c'è un tassametro a bordo). Le caratteristiche diverse possono non solo essere attributi (il colore, il numero dei passeggeri) ma anche funzionali: pensiamo ad esempio che il 'costo per chilometro' va calcolato in modo diverso. Con un'auto normale, abbiamo il prezzo della benzina e l'ammortamento del

costo, con un taxi abbiamo una tariffa a tempo e/o a distanza. Data però la grande somiglianza che esiste tra un'automobile normale ed un taxi, pare brutto costruire una nuova classe da zero solo per trattare queste piccole differenze.

L'idea è allora di prendere la classe Automobile e specializzarla, definendo la classe Taxi come una figlia della classe Automobile, di cui eredita tutte le caratteristiche. Inoltre, e soprattutto, la classe Taxi aggiunge di suo qualcos'altro.

Ad esempio, ha bisogno di un metodo per la Prenotazione (cosa che un'auto normale normalmente non ha), e di modificare pesantemente il metodo del costo orario. Nel primo caso la classe Taxi aggiunge un nuovo metodo, nel secondo sovrascrive un metodo precedente (tecnica detta di **overloading**).

Tutto ciò avviene generalmente con molta semplicità: la definizione della classe Taxi si fa normalmente all'interno di un linguaggio OOP richiamando la definizione della classe da cui ereditare, e definire esplicitamente le nuove caratteristiche (dati interni, o metodi) o le modifiche a quelle esistenti (metodi; non è possibile modificare le strutture dati esistenti).

La classe Automobili si dice essere la **superclasse** della classe Taxi, e viceversa, la classe Taxi è una **sottoclasse** di Automobili. Una sottoclasse eredita la struttura dati della superclasse, e può aggiungere nuovi dati, ma non può cancellarne.

Le istanze della sottoclassi avranno dunque una struttura dati più grande o al più uguale a quella della superclasse. Per quanto riguarda i metodi, la sottoclasse eredita tutti i metodi della superclasse, e ne può aggiungere di nuovi o sovrascrivere, in tutto o in parte, quelli vecchi. Non si può cancellare un metodo della superclasse, ma se lo riscrivo facendogli fare nulla, è come se lo avessi cancellato.

GERARCHIA DELLE CLASSI

Il procedimento può essere reiterato, ed anzi, normalmente, è proprio costruendo sottoclassi e sottoclassi che si scrive un programma. Si viene così a costruire una gerarchia di classi, in cui ogni classe ha almeno una superclasse, e può avere (o anche no) diverse sottoclassi. Esiste normalmente un unico capo di questa gerarchia delle classi, tipicamente un oggetto primitivo, il cui unico scopo è di fornire un modello base per definire altre classi. L'oggetto base non ha normalmente struttura dati (almeno significativa) e possiede giusto quei metodi di uso generico che servono a far nascere e morire un oggetto.

Non esiste un limite superiore al numero di livelli di sottoclassi che si possono costruire. Anzi; l'attività di un programmatore OO consiste nel mettere assieme un programma assemblando una serie di classi. Più classi conosce, più è probabile che trovi quelle che gli servono, senza bisogno di scriverne lui stesso. Se proprio non trova la classe adatta, se la scrive lui, cercando la classe più simile che gli riesce di trovare, in modo da sovrascrivere il numero minimo di metodi e cercando di utilizzare il lavoro già fatto.

Sono a questo punto in grado di definire meglio Cocoa:

è un insieme di classi predefinite da Apple per la costruzione di programmi

funzionanti all'interno del sistema operativo Mac Os X.

Scrivere un programma utilizzando Cocoa significa mettere assieme un po' di classi e, attraverso lo scambio di opportuni messaggi, realizzare le operazioni richieste dall'utente.

BINDING

Un messaggio richiede un oggetto destinatario ed una indicazione del metodo richiesto. È ovviamente una cosa buona e giusta che l'oggetto destinatario sia in grado di rispondere al messaggio, altrimenti possono nascere guai. Esistono fondamentalmente due metodi per fare questo controllo (**binding**), statico o dinamico.

Il **binding statico** richiede che sia noto da subito il destinatario (al momento della compilazione): il compilatore controlla che tra i metodi dell'oggetto destinatario sia presente quello richiesto. Questo processo porta ad una efficienza in sede di esecuzione, ma occorre essere precisi in sede di stesura del codice.

Questo è l'approccio tipico della metodologia di programmazione procedurale: il controllo di tipo e la verifica della fattibilità dell'operazione è controllato in sede di compilazione. Chi ha programmato in C ricorderà certamente i messaggi di errore dovuti al fatto che si richiedono operazioni non lecite sui tipi quali assegnare numeri floating point a variabili reali.

Chi invece si è avvicinato alla programmazione usando ad esempio Hypercard o Applescript, ha scoperto che un contenitore può contenere qualsiasi cosa, ed è solo in sede di esecuzione (all'ultimo momento possibile) che l'ambiente operativo si accorge di operazioni non lecite (ad esempio, estrarre il terzo elemento di una lista, quando il dato non è una lista...). Questo è un esempio di **binding dinamico**, in cui il controllo che una data operazione sia possibile e lecita per una dato oggetto avviene all'ultimo momento, cioè all'invio del messaggio.

Questo approccio è estremamente flessibile (posso chiedere ad un oggetto il suo colore, senza dovermi preoccupare se l'oggetto è un automobile o un taxi), ha qualche problema di prestazioni (devo fare il collegamento messaggio-procedura da eseguire ogni volta che invio il messaggio) e qualche problema in più (se mando il messaggio all'oggetto sbagliato, potrei accorgermene quando è troppo tardi). Ovviamente, la cosa migliore sarebbe avere a disposizione entrambe le modalità, ed in effetti così spesso accade (per lo meno, in Objective C).

POLIMORFISMO

Dicevo prima che una delle caratteristiche divertenti del binding dinamico è di chiedere il colore ad un oggetto senza doversi preoccupare di cosa sia l'oggetto destinatario. Questo è un esempio di **polimorfismo**.

Il prerequisito del polimorfismo è proprio il fatto di poter avere metodi con lo stesso nome in oggetti anche completamente diversi. Se ho un insieme di oggetti diversi che devo dipingere di colore giallo, per sapere di quanta pittura ho bisogno, invio ad ogni oggetto un messaggio che chiede loro la superficie da colorare. A questo livello, non ho

alcun interesse sulla natura degli oggetti, mi basta sapere che il primo oggetto ha una superficie di tre metri quadri, il secondo di venti centimetri quadri, ed il terzo di due metri quadri. Solo che il primo oggetto è una porta d'automobile, il secondo una maniglia ed il terzo una scatola di cartone.

Fare la stessa cosa con un linguaggio procedurale avrebbe richiesto molto lavoro ed un sottile distinguo ogni volta che chiedo la superficie ad un oggetto. Con il polimorfismo è molto più semplice. Programmare classi con metodi aventi lo stesso nome è una possibilità, non un vincolo. Quindi il polimorfismo ha senso solo se serve.

Il vantaggio del polimorfismo si ha dove si riconosce un comportamento simile tra due classi diverse tra loro, per cui ha senso avere lo stesso nome. Avere lo stesso nome ingenera confusione dove i metodi hanno in realtà compiti concettualmente diversi.

GERARCHIA E RETI DI OGGETTI

Un programma scritto secondo la metodologia OOP è soggetto a due linee guida strutturali.

La prima struttura è stata evidenziata esplicitamente, ed è la gerarchia delle classi. Esiste un oggetto base, la madre di tutti gli oggetti, dal quale discendono tutti gli oggetti, attraverso il meccanismo delle sottoclassi. Di sottoclasse in sottoclasse, ogni oggetto che costituisce il nostro programma può venire classificato (!) per quanto riguarda il suo funzionamento. Ma questa gerarchia dà un'idea di come funziona ogni singolo oggetto, ma non giustifica il funzionamento dell'intero programma.

Abbiamo detto che il programma sono un po' di oggetti che si scambiano messaggi. Ovviamente, gli oggetti non si scambiano messaggi a caso, ma con criterio. Esiste quindi tra gli oggetti una *rete* di collegamenti che rendono conto delle strade percorsi dai messaggi, che costituisce una buona parte delle fatiche della costruzione di un programma completo.

La rete dei collegamenti non deve essere statica, ovvero non devono essere indicati fin dall'inizio tutti i possibili collegamenti che esistono tra gli oggetti, ma è possibile creare collegamenti al volo, mantenerli per po', chiuderli, cose così.

Ci sono linee di comunicazione completamente transitorie, come un oggetto che nasce, manda un messaggio che richiede una data operazione, e poi muore avendo compiuto il suo dovere; un po' come collegarsi in internet tramite il telefono da casa: il mio oggetto-computer scambia messaggi (pacchetti di dati) con l'oggetto-web finché rimango collegato.

In altri casi invece, piuttosto interessanti, la comunicazione è continua e sempre attiva; come essere collegati in permanenza ad internet tramite linea dedicata: il computer potenzialmente continua a scambiare dati col web.

Una rete permanente di linee di comunicazione è indispensabile per costruire strutture di oggetti cooperanti; una finestra è composta da un oggetto-titolo, due oggetti-barre di scorrimento laterali, un oggetto-pulsante di chiusura, un oggetto-contenuto della finestra stessa, eccetera. In pratica, una finestra è un gruppo di oggetti che cooperano tra loro per dare luogo alle funzionalità che tutti conosciamo. Il legame tra

i vari oggetti è statico: l'oggetto barra di scorrimento comunica con l'oggetto contenuto per informarlo del fatto che deve mostrare una certa porzione di documento, l'oggetto pulsante di zoom comunica a tutti di portarsi nella posizione di zoom,e via così.

Queste comunicazioni tra oggetti sono strutturali, nel senso che partecipano alla struttura del programma: sono l'ossatura su cui il programma si basa. Il mezzo più comodo (ma non l'unico) è di immagazzinare nella struttura dati dell'oggetto gli altri oggetti con cui l'oggetto in questione ha relazioni stabili. Cocoa chiama questa variabile **outlet**: un outlet è dunque la linea dedicata che intercorre fra un oggetto ed un altro oggetto con cui comunica.

È bene precisare che questo concetto non è proprio della OOP, ovvero, si fa OOP anche senza il concetto di outlet. Tuttavia, un outlet è molto comodo proprio nelle situazioni in cui l'interfaccia utente è modellata per oggetti; in particolare, diventa un mezzo potentissimo quando la variabile outlet è gestita più o meno automaticamente dall'ambiente operativo. L'inizializzazione degli outlet (quando la struttura degli oggetti è costruita opportunamente, ad esempio attraverso Interface Builder, vedremo poi) avviene in maniera automatica, per cui le connessioni sono già pronte e disponibili quando si tratta di usarle. Ovviamente, una variabile outlet non deve necessariamente rimanere fissa; l'oggetto cui si riferisce può cambiare durante il corso del programma, sia automaticamente sia per ragioni connesse allo sviluppo del programma stesso.

IL PARADIGMA MVC

Un vecchio concetto (deriva dallo Smalltalk), molto utile soprattutto quando si ha a che fare con programmi che interagiscono con l'utente, è il **paradigma MVC**, *Model-View-Controller*. Questo modo di vedere le cose è utile per separare tra loro alcune funzionalità fondamentali di una applicazione. Il paradigma non fa parte della OOP, ma si tratta ancora una volta di un concetto utile.

L'idea è di dividere gli oggetti in tre categorie, gli oggetti Modello, gli oggetti Vista e gli oggetti Controllori.

Gli *oggetti Modello* contengono appunto un modello del funzionamento, sono la base della conoscenza, sono i depositari del modo di lavorare. Come tali, raramente hanno una rappresentazione visibile. Un oggetto in grado di trattare elenchi di indirizzi, ad esempio, ordinarli, manipolarli, eccetera, è un oggetto Modello. Sono oggetti molto importanti, che definiscono la struttura dell'applicazione.

Gli *oggetti Vista* rappresentano qualcosa direttamente visibile su di uno schermo, ed in generale rappresentano un modo per esporre e rendere disponibile qualcosa. Gli elementi dell'interfaccia utente come finestre, menu, pulsanti, controlli in genere, sono tutti oggetti Vista. Poiché in definitiva un oggetto Vista rappresenta dei dati e modalità di interazione, i modi di rappresentazione sono mutevoli e diversi, tanto che spesso ci si trova a doverne costruirne di nuovi, magari a partire da oggetti già fatti.

Ovviamente Cocoa ne mette a disposizione moltissimi, e moltissimo si può fare semplicemente arrangiando opportunamente gli oggetti Vista messi a disposizione. Utilizzare oggetti Vista già costruiti rende un'applicazione consistente con le altre applicazioni. Mettere a punto gli oggetti Vista e come questi sono legati tra loro è la parte più divertente dell'applicazione, ed esistono applicazioni apposite per fare ciò con

facilità.

Gli *oggetti Controllore* si trovano a metà strada tra gli oggetti Vista e gli oggetti Modello. Se gli oggetti Vista sono solo Forma e gli oggetti Modello sono pura sostanza, chi si preoccupa di mediare tra questi due estremi sono gli oggetti controllore. L'insieme degli oggetti vista che costituisce una rappresentazione dei dati e un modo di interazione con l'utente si riferisce generalmente ad un oggetto controllore.

Quando, ad esempio, si fa clic sopra il pulsante `Cerca` di un dialogo per la ricerca di stringhe all'interno di un testo, l'oggetto pulsante nulla sa fare se non inviare un messaggio all'oggetto controllore segnalando l'intenzione dell'utente di cercare qualcosa. L'oggetto controllore si preoccupa di verificare cosa è successo, recupera le altre informazioni dagli altri oggetti vista (il testo da cercare), verifica lo stato corrente (ad esempio, se deve fare attenzione alle maiuscole/minuscole), raccoglie tutte le informazioni al contesto, ne verifica la consistenza, e poi, finalmente, interagisce coll'oggetto Modello (che è a questo punto il depositario di tutta la conoscenza) inviandogli un messaggio di ricerca testo.

Come si può capire dall'esempio, un oggetto Controllore è molto legato all'applicazione specifica, in quanto è l'oggetto che consente l'esecuzione delle operazioni.

La ripartizione degli oggetti in tre categorie consente di semplificare ancor di più il processo di progettazione di una applicazione. Avendo completamente separato gli oggetti Vista, la costruzione di una interfaccia utente consiste nel disporre opportunamente una serie predefinita di oggetti Vista all'interno di finestre (anch'essi oggetti Vista...); visto che questi oggetti si portano dietro struttura dati e funzionalità, risulta facile e veloce costruire una interfaccia utente quasi funzionante: vedremo che Interface Builder è lo strumento per fare ciò.

Per inciso, i famosi ambienti di sviluppo di tipo *Visual* (Hypercard ne è stato il predecessore) non sono altro che l'estremizzazione di questo paradigma: l'ambiente di programmazione consente la disposizione degli elementi dell'interfaccia (gli oggetti Vista), ai quali si associano in qualche modo, non proprio pulitamente, gli oggetti Controllore (lo script) e Modello (un po' disperso tra le funzionalità realizzate).

Non sempre il paradigma MVC è utile (molti programmatori ne fanno tranquillamente a meno); ad esempio, programmi in cui gli oggetti Vista sono fondamentali ed il collegamento con il modello molto stretto (non ci sono molti controlli e verifiche, ed il contesto è meno importante), si può eliminare il livello Controllore.

Certo è che l'uso del paradigma MVC porta nella maggior parte dei casi ad una applicazione di facile ed immediata strutturazione, comprensione e manutenzione. La separazione tra la parte frivola (Vista) e la parte seria (Modello) permette di concentrare lo sforzo in una sola direzione (e di dividere con facilità il lavoro tra più programmatori), mentre il controllore rimane la parte più difficile da fare: difficile perché ci sono tante cose di cui tenere conto, ma non perché ci siano cose difficili di per sé.

Capitolo 005 - Objective C

Ok la programmazione object-oriented; ma in pratica? Ci occorre un linguaggio di programmazione.

Sorgenti: Segue da Objective C

Primo inserimento: Settembre-Ottobre 2001

OBJECTIVE C

Objective C è una estensione del linguaggio C con alcuni costrutti aggiuntivi per poter programmare secondo la metodologia OOP. Pare addirittura che ObjC sia in realtà un semplice C mascherato.

Parentesi aperta. Normalmente, un compilatore trasforma le istruzioni scritte in una determinata forma in codice macchina eseguibile. Molti linguaggi possiedono un pre-compilatore che si occupa di preparare in maniera acconcia la lista delle istruzioni, ma non è concettualmente un oggetto che esegue compiti fondamentali. Tuttavia, in questo caso, il pre-compilatore ObjC trasforma il linguaggio ObjC in linguaggio C vero e proprio. Poi, un normale compilatore C è in grado di produrre l'applicazione che interessa. Verificheremo questa cosa più avanti, quando andremo a fare conoscenza con il compilatore. Chiusa parentesi.

Per il momento, parliamo del linguaggio ObjC e vediamo come è fatto. In questo documento non parlerò assolutamente di C, lo darò per scontato; così non fosse, ci vorrebbe un intero libro per parlarne, e quindi vi rimando a questi, che trovate abbondantemente (beh, insomma... ormai sono quasi tutti sul C++).

D'altra parte, la OOP richiede l'appropriazione di concetti un po' differenti, e quindi credo che conoscere come si fa a definire una matrice di puntatori a funzioni che hanno come argomento un puntatore a struttura non sia fondamentale per imparare a programmare in ObjC. Ritengo quindi che una conoscenza superficiale del C sia sufficiente per muovere i primi passi senza necessariamente essere colti da dubbi in ogni momento.

ObjC è un sovrainsieme del C, nel senso che contiene al suo interno l'intero C; il compilatore ObjC (ammesso che esista e non si tratti del solo precompilatore) è in grado quindi di trattare tranquillamente programmi scritti in C normale. I file scritti in linguaggio C ed i file scritti in linguaggio ObjC si distinguono per l'estensione del file (lo so, lo so, non è bello in un ambiente Macintosh, ma siamo stati abituati bene... e poi, credetemi, tante volte è molto comodo aggiungere un'estensione, fin troppo comodo...). Un file in C ha l'estensione `.C`, mentre un file in ObjC ha l'estensione `.M` (non so perché). L'estensione del file dovrebbe distinguere se sul file opera il preprocessore standard (i file C) oppure il preprocessore speciale ObjC (i file M).

Dicevo delle estensioni del C con orientamento agli oggetti che danno luogo al ObjC; la maggior parte di queste ricordano in maniera stretta costrutti del linguaggio Smalltalk, uno dei linguaggi OOP *puri*, in cui tutto (ma proprio tutto) è un oggetto.

Intanto, comincio a parlare di Oggetti. non mi stancherò mai di ripetere: Un *Oggetto* è l'insieme di una *Struttura Dati*, definita dalla *Classe*, e da una serie di operazioni che usano questi dati. Le operazioni si chiamano *Metodi*, ed i metodi operano sui valori della struttura dati, le **Variabili d'Istanza**. Un oggetto, o meglio la sua realizzazione concreta, un'**Istanza**, è un costrutto che raccoglie come un tutto indissolubile una struttura dati valorizzata (variabili d'istanza) con un gruppo di procedure (metodi). I metodi dell'oggetto sono l'unico modo per accedere o manipolare le variabili d'istanza. Se non c'è un metodo per manipolare direttamente la variabile, non ci si può far nulla. Inoltre, i metodi sono legati all'oggetto, nel senso che i metodi si applicano solo all'oggetto per cui sono definiti.

TIPO D'OGGETTO

In ObjC, un oggetto è dichiarato come un `id`, cioè una variabile in grado di identificare un oggetto di tipo id, come un numero intero è di tipo int o un numero floating point di tipo float:

```
id ilMioOggetto;
```

sospetto che il tipo `id` non sia altro che un puntatore a `void`, un metodo classico in C per indicare un puntatore a qualcosa che non si sa bene cosa.

Ecco, sospetto infondato. Sono andato a guardare la definizione di id; è un puntatore ad una struttura, che contiene una variabile Class, che NON è un tipo predefinito, ma una struttura piuttosto complessa e complicata che non voglio indagare oltre. Dove ho trovato tutto ciò? nei file `objc.h` e parenti, di difficile reperibilità...

Il tipo `id` non convoglia altre informazioni che non siano: è un oggetto. Come si fa allora a sapere con che razza di oggetto abbiamo a che fare?

Una della variabili d'istanza di ciascun oggetto è in effetti `isa`, che è un puntatore alla classe cui l'oggetto appartiene (ma la variabile d'istanza non può essere letta dall'esterno! beh, da qualche parte ci sarà un metodo per conoscerla..., sempre ammesso che occorra conoscerla: il binding dinamico ed il polimorfismo si basano proprio sul fatto di ignorare il tipo dell'oggetto). Utilizzando questa variabile il compilatore e l'ambiente di sviluppo sono in grado di capire che tipo di oggetto stanno considerando, e di attivare le operazioni conseguenti.

MESSAGGI

Per inviare un messaggio ad un oggetto, si usa la seguente sintassi:

```
[ oggetto-ricevente messaggio ]
```

Visto che questa è una istruzione C, alla fine dell'invio del messaggio deve esserci un ';'. Abbiamo poi anche detto che un messaggio provoca l'esecuzione di un metodo, che può generare un risultato. In definitiva, una semplice istruzione in ObjC che invia un messaggio ad un oggetto e mette da parte il risultato è:

```
risultato = [ ricevente messaggio ];
```

Ora, i messaggi possono avere degli argomenti; questi si scrivono nel seguente modo:

```
<nome metodo>: argomento <etichetta>: argomento <etichetta>: argomento ...
```

Ad esempio, avendo un oggetto `mouse` che riceve messaggi di movimento ad una locazione di coordinate X e coordinata Y, potremmo avere un messaggio formulato così:

```
[ mouse moveToX: 100 moveToY: 50 ]
```

che dice al mouse di spostarsi alla coordinata (100,50). Ci sono due appunti. Il metodo, in questo caso, non si chiama `moveToX`, ma in maniera completa: `moveToX:moveToY:`. L'etichetta può non essere presente, ma allora il metodo cambia nome (è un altro metodo); ad esempio, si potrebbe avere il metodo `moveToXY::` (si noti la presenza del doppio due punti)

```
[ mouse moveToXY: 100 : 50 ]
```

C'è infine un altro metodo per dare argomenti, ovvero mettere più argomenti assieme, separandoli da una virgola; il metodo a questo punto si chiama `moveToXY:` (si noti UN solo due punti):

```
[ mouse moveToXY: 100, 50 ]
```

Ho messo le possibilità in ordine di preferenza stilistica: è molto più bello avere metodi completi di tutte le etichette (sono più chiari e leggibili, quasi discorsivi), meno bello non avere etichette, assolutamente brutto avere argomenti multipli.

Sfruttando il fatto che l'esecuzione di un metodo produce un risultato, si possono inviare più messaggi all'interno di una stessa riga di istruzione:

```
[ mouse moveToX: 100 moveToY: [ mouse clickV] ];
```

Questa immaginaria istruzione sposta la coordinata `y` del mouse alla coordinata y dell'ultima locazione dove il mouse ha fatto clic.

CLASSI

Le classi sono oggetti. Sono oggetti di tipo speciale, perché servono essenzialmente a produrre altri oggetti, istanze della classe. La classe è l'oggetto ideale, di cui tutte le istanze sono immagini (un concetto un po' platonico, ma tant'è). Una classe dichiara le variabili d'istanza (ma non le definisce... c'è una sottile distinzione non banale), e definisce tutti i metodi che gli oggetti istanza della classe possono utilizzare.

Intervallo: la distinzione tra **dichiarare** e **definire** una variabile; questo discorso va bene anche per altri linguaggi di programmazione (ad esempio, il C). Quando dichiaro qualcosa (una classe, una variabile), affermo solamente la sua esistenza. Non è costruito alcun spazio informatico per questa cosa: dichiarare una variabile col nome `aloha`

significa che, da qualche parte all'interno dell'applicazione, esiste un posto indicato appunto come `aloha` dove è conservato il valore di questa variabile; il compilatore, incontrando una dichiarazione, la deve *risolvere*, ovvero andare a trovare questo posto, in modo che poi sappia rintracciare il valore. Ma il posto deve esistere. Creare il posto è compito della definizione. Definendo una variabile col nome `aloha`, si crea il posto dove immagazzinare questa variabile. All'interno di una applicazione deve esistere una ed una sola definizione e quante si vogliono dichiarazioni.

Riprendendo la classe: dichiara le variabili d'istanza, cioè dice che esistono, ma non ne crea lo spazio. In pratica, non ne ha; dichiarare le variabili d'istanza è utile al momento della definizione di una istanza (definizione!), che crea lo spazio necessario per conservare tutte le variabili d'istanza.

Una classe definisce invece tutti i metodi: vale a dire, tutti i metodi devono prevedere il loro bravo pezzo di codice che verrà eseguito al ricevimento del messaggio apposito. Tutte le istanze di questa classe condividono questi metodi, ed è per questo che i metodi sono definiti dalla classe. Invece, tutte le istanze hanno le proprie variabili d'istanza, ed è per questo che la classe dichiara le variabili d'istanza. Per convenzione, agli oggetti Classe si danno nomi che cominciano con lettera maiuscola (come ad esempio `LaMiaClasse`), mentre agli oggetti Istanza della classe si danno nomi che cominciano con lettera minuscola, `eccoLaIstanza`.

Abbiamo detto altrove che ogni classe è sottoclasse di un'altra classe, tranne una classe che avevano chiamato primitiva; un termine più usato è **root**, radice. Nel caso di Cocoa, questa classe è chiamata `NSObject`. Tutte le classi di Cocoa sono discendenti (più o meno vicine) di questa Classe, la madre di tutte le Classi.

Mirko Viviani mi corregge: le classi root sono in realtà almeno due (basta leggere l'introduzione della documentazione relativa al Foundation Framework, anzi, basta guardare la figura): le classi root presenti all'interno del framework sono `NSObject`, la più importante ed utilizzata, e `NSProxy`, che prende momentaneamente il posto di un'altra classe, perché quest'ultima è lontana oppure non abbiamo voglia di gestirla.

Benchè sia concettualmente possibile creare una nuova gerarchia di Classi a partire da un nuovo oggetto da noi definito, nella pratica la cosa è così complicata e inutile che non viene mai fatta.

Mirko Viviani mi corregge ancora; in effetti costruire una nuova gerarchia di classi a partire da un diverso oggetto root non è molto difficile; ma, per quanto riguarda Cocoa, in pratica non serve.

La gerarchia delle classi funziona in questo modo: in principio c'era `NSObject`. Attraverso il meccanismo dell'ereditarietà, da `NSObject` sono state definite nuove classi figlie; e le figlie hanno generato altre figlie. Ogni figlia riceve dalla madre (superclasse) tutte le sue funzionalità (variabili d'istanza e metodi); alcune figlie aggiungono caratteristiche (variabili d'istanza), altre funzionalità (nuovi metodi), altre ancora sia metodi che variabili, altre figlie un po' più ribelli riscrivono metodi...

Sempre per merito di Mirko Viviani mi vedo costretto a commentare il testo seguente, che non è preciso dal punto di vista informatico, lo è di più dal punto di vista pratico.

In mezzo a tutto questo bailamme, si nota l'esistenza di classi un po' bizzarre, nullafacenti, con nessuno scopo nella vita se non quella di servire ad esempio. A ben vedere, è proprio il caso di `NSObject`, che di suo non ha proprio nulla di interessante.

`NSObject` non è una classe nullafacente, anzi. Realizza una moltitudine di metodi importantissimi per lo sviluppo delle applicazioni; ma sono metodi di uso generale: per avere funzioni specifiche dell'applicazione, bisogna costruire sottoclassi.

Per avere una classe che serve a qualcosa, bisogna verosimilmente aggiungere qualche variabile d'istanza e qualche metodo interessante, che altrimenti `NSObject` è piuttosto noioso. Classi progettate in modo da non avere vita propria, ma destinate a funzionare solo come matrice per altre sottoclassi sono chiamate **Classi Astratte**. Non esistono cioè istanze della classe astratta, ma solo istanze generate a partire da sottoclassi di questa classe astratta.

In effetti, il concetto di classe astratta è differente. Una classe è astratta quando devono essere necessariamente estese per poter funzionare. Se non fossero estese, non possono proprio funzionare e generare errore. `NSObject` non è una classe astratta. Invece, a sentire la documentazione, `NSProxy` lo è.

In pratica una classe astratta nasce per raccogliere assieme una serie di funzionalità comuni ad un insieme di oggetti, ma che devono essere rifinite a mano per funzionare correttamente. Un po' come quelle solette mangiadore da inserire all'interno delle scarpe: al supermercato esiste lo stampo già fatto, ma per utilizzare la soletta, dovete ritagliare la sagoma delle dimensioni del vostro piede...

CLASSI E TIPO

Una Classe è a tutti gli effetti un *tipo*, ovvero la definizione di una classe definisce un tipo di dato (qui tipo ha lo stesso significato dei tipi standard del C, come ad esempio int, float, eccetera). È quindi perfettamente lecito utilizzare il nome della classe `MiaClasse` in costrutti come `sizeof( MiaClasse)`, e soprattutto per dichiarare oggetti (ah, finalmente ci siamo arrivati):

```
MiaClasse    * ilMioPrimoOggetto ;
```

Ecco, ci siamo: questa è una dichiarazione (dichiarazione: dico che esiste da qualche parte una istanza che si chiama `ilMioPrimoOggetto`, che è di tipo `MiaClasse`) di una istanza di oggetto, ed il fatto di aver indicato esplicitamente il tipo di questo oggetto significa che ho utilizzato il binding statico. Alternativamente, avrei potuto utilizzare una dichiarazione più neutra, del tipo

```
id ilMioSecondoOggetto ;
```

Il vantaggio principale del binding statico è che, sapendo bene come stanno le cose, il compilatore produce codice più efficiente e veloce. D'altra parte, si perde in flessibilità (questa è una caratteristica standard del mondo, nota come *congettura della coperta corta*: ogni cosa è come una coperta corta; se copri i piedi, rimane fuori la testa, se copri la testa, rimangono fuori i piedi).

Si può fare binding statico anche utilizzando come tipo una delle classi ancestrali dell'oggetto che ci interessa; se `MadreClasse` è una superclasse di `MiaClasse`, la dichiarazione

```
MadreClasse *ilMioPrimoOggetto ;
```

avrebbe dichiarato un oggetto di tipo `MadreClasse`. Dove sta il vantaggio? Il compilatore ottimizza il codice trattando `ilMioPrimoOggetto` come una `MadreClasse`, ma poi, durante l'esecuzione, esegue ugualmente il binding dinamico per arrivare (se il caso) ad usare i metodi di `MiaClasse`; così facendo si risparmia un po' di lavoro al binding dinamico (c'è meno strada da fare per arrivare in fondo), aumentando un po' le prestazioni...

Ancora una volta, Mirko Viviani mi corregge: ad esempio, il codice prodotto per gestire:

```
id stringa;
```

o

```
NSString *stringa;
```

è assolutamente lo stesso. Ciò che si guadagna con l'esplicita tipizzazione della variabile è un controllo dei metodi al momento della compilazione, con conseguente generazione di avvertimenti se il metodo utilizzato non è definito per l'oggetto in esame.

Quando, durante l'esecuzione, si invoca un metodo, sempre e comunque è utilizzata una specifica funzione, chiamata `objc_msgSend()`, che fa parte del cosidetto meccanismo di *run-time* dello ObjC. Il manuale del linguaggio ObjC fornito con la documentazione dei Developer's Tool sconsiglia caldamente la chiamata diretta di questa funzione, ma di lasciar fare al compilatore.

OGGETTI CLASSE

Devo aver già detto che una classe è un oggetto (speciale, senza variabili, cose del genere), e definisce tutti i metodi che poi sono utilizzati dagli oggetti creati a partire da quella classe. Ma un oggetto Classe, che si può individuare esplicitamente tipizzandolo come tipo `Class`, può avere metodi propri. Anzi, i metodi propri sono verosimilmente gli unici utilizzabili su di oggetto Classe, in quanto gli altri metodi utilizzano quasi certamente i valori delle variabili d'istanza (che un oggetto Classe non ha in quanto le dichiara ma non può definirle).

Insomma, gli oggetti `Class` sono oggetti a tutti gli effetti, che possono essere tipizzati dinamicamente (eh, che brutta locuzione: significa che posso dichiararli come `id`, ma poi usarli come `Class`, sono cioé soggetti al binding dinamico), possono ricevere messaggi, ereditare metodi da altre classi, eccetera. Sono speciali perchè sono creati direttamente dal compilatore (non devo definirli esplicitamente), non hanno variabili d'istanza e servono come stampi per la produzione di oggetti.

Quindi, ecco come si fa a generare oggetti: per prima cosa, definisco la variabile che individua l'oggetto (cioè, definisco un puntatore all'oggetto).

```
id ilMioOggetto;
```

Avrei potuto anche scrivere con binding statico:

```
MiaClasse *ilMioOggetto ;
```

A questo punto ho definito la variabile che serve a contenere l'oggetto, ma non ho ancora creato l'oggetto; per questo, occorre una nuova istruzione:

```
ilMioOggetto = [ MiaClasse alloc ] ;
```

Cosa succede? Ho inviato il messaggio `alloc` alla classe `MiaClasse`; ogni classe eredita dalla madre di tutte le classi (`NSObject`, ricordo) il metodo `alloc`, che appunto crea lo spazio necessario a conservare un oggetto delle dimensioni di `MiaClasse`.

Ma non è ancora finita. Lo spazio c'è, ma è ancora vuoto ed inutilizzabile. Occorre inizializzarlo, utilizzando un altro metodo ereditato da `NSObject`, chiamato `init`; in realtà le operazioni si fanno normalmente assieme, con una istruzione come la seguente:

```
ilMioOggetto = [[MiaClasse alloc] init];
```

a questo punto, `ilMioOggetto` è pronto a funzionare.

L'unico problema, è che non abbiano ancora idea di come sia fatta la classe...

Capitolo 006 - Objective C

Continua la descrizione del linguaggio di programmazione Objective C.

Sorgenti: Segue da Objective C

Primo inserimento: 24 Ottobre 2001

DEFINIZIONE DI UNA CLASSE

Una classe si definisce in due passi successivi: l'interfaccia e la realizzazione (**interface** e **implementation**); con l'interfaccia si dichiarano i metodi e le variabili d'istanza. Con la realizzazione si definisce (finalmente) la classe, scrivendo il codice che realizzai metodi.

Questa bipartizione si riflette nella (non necessaria, ma caldamente raccomandata) separazione della definizione di classe in due file separati; ancora, il nome dei file è libero, ma è caldamente raccomandato utilizzare il nome della classe con apposito suffisso.

Quindi, l'interfaccia della classe `MiaClasse` si scrive all'interno del file `miaclasse.h`, la realizzazione si scrive all'interno del file `miaclasse.m` (se non l'ho già detto, ignoro totalmente il perché del suffisso m ai file objC).

È ugualmente sconsigliato (ma non impedito) dichiarare/definire più di una classe per file.

Tutte queste prescrizioni servono per rendere poi più facile il lavoro del compilatore, ed anche del programmatore, che con queste poche e semplici regole sa sempre dove mettere le mani per lavorare senza troppa fatica. La divisione in due file del lavoro invece deriva dalla filosofia della OOP, che richiede una netta separazione tra l'interfaccia e la realizzazione.

Si possono utilizzare con profitto oggetti di cui si conosce la sola interfaccia (quindi, il solo file .h) e si ignora totalmente la struttura interna su cui si basa il funzionamento (il file .m).

La cosa avviene talmente spesso che poi non ci si fa più caso...

L'INTERFACCIA

La dichiarazione di una classe funziona in questo modo: si dice che abbiamo un'interfaccia di una classe, e che questa classe è figlia di questa superclasse. Segue la dichiarazione delle variabili d'istanza, e poi dai metodi dell'oggetto. Fine. In linguaggio ObjC:

```
@interface NomeDellaClasse : NomeDellaSuperClasse
{
    dichiarazione delle variabili d'istanza
```

```
}
dichiarazione dei metodi
@end
```

È fondamentale mettere il nome della superclasse, altrimenti la nostra classe diverrebbe una classe `root`; in tal caso, sono tante e tali le variabili d'istanza ed i metodi da definire per rendere tale classe funzionante correttamente che non ne vale la pena. Quindi, mettiamo sempre il nome della superclasse. Se proprio vogliamo avere classi figlie di nessuno, mettiamo almeno `NSObject`.

Mirko Viviani mi corregge: il lavoro per realizzare una classe root non è necessariamente tantissimo. Rimane invece complicato realizzare una nuova classe root che faccia concorrenza e possa riampiazzare NSObject.

La dichiarazione della variabili d'istanza utilizza i tipi standard del C: possiamo quindi dichiarare variabili di tipo intero (`int`, con varie dimensioni: `short`, `long`), floating point (`float`, `double`), ed anche un po' di tipi aggiuntivi messi a disposizione (come altri oggetti...).

La definizione dei metodi richiede qualche commento. In primo luogo, occorre distinguere tra i metodi di classe (che si possono applicare agli oggetti Classe) e metodi applicabili a tutte le istanze (metodi d'istanza). Questi metodi si distinguono per il primo carattere della dichiarazione dei metodi: nel caso di metodi di classe, si comincia con il carattere '+', gli altri cominciano con '-'. Ad esempio il metodo principe per la creazione di oggetti, vale a dire `alloc`, si applica agli oggetti classe, quindi è dichiarato come:

```
+ alloc ;
```

(poiché `NSObject` definisce il metodo alloc ereditato da tutti gli oggetti, non occorre quasi mai ridefinire questo metodo).

Un metodo normale invece si dichiara in questo modo. Considero un metodo che non ha argomenti e non restituisce risultati:

```
- (void) mioMetodo ;
```

Vediamo invece come si dichiarano i tre metodi visti nel capitolo precedente per il movimento del mouse:

```
- (void) moveToX: (int)coordX moveToY: (int) coordY ;
- (void) moveToXY: (int)coordX : (int) coordY ;
- (void) moveToXY: (int)coordX, ... ;
```

USARE ALTRE CLASSI

Capita di dover utilizzare altri oggetti all'interno di un oggetto. Anzi, è certo: per definire una classe deve essere nota almeno la superclasse. Ecco che arriva la potenza della OOP: non è assolutamente importante come è realizzata questa superclasse; ci basta sapere qual è la sua interfaccia verso l'esterno. In altre parole, ci basta sapere come è dichiarata questa classe.

Da sempre, il C mette a disposizione un meccanismo per evitare di scrivere due volte le stesse cose (la dichiarazione della superclasse, nel caso), ovvero la direttiva (al precompilatore) `#include`. Una direttiva al precompilatore è semplicemente un comando, al di fuori del linguaggio C (o C++ o Objective C), diretto solo al precompilatore; quando il precompilatore incontra questo comando, esegue le particolari operazioni richieste (ad esempio, `#include` ricopia il file .h all'interno del file dove si trova questa direttiva). Un uso accorto delle direttive al precompilatore rende molto più facile programmare, leggere e documentare un programma scritto in C. Viceversa, un uso poco accorto delle direttive rende incomprensibile un programma, difficile programmare, impossibile da correggere.

> Aneddoto: ricordo di essermi imbattuto anni fa in una specie di concorso di 'obfuscated C', in cui i partecipanti dovevano scrivere un programma in C, perfettamente funzionante e quindi corretto sintatticamente, ma che fosse il più incomprensibile e 'offuscato' alla comprensione possibile. Uno dei partecipanti appunto produsse un programma che, con l'uso spropositato delle direttive al precompilatore, consisteva in pratica in una sola lettera. Fine aneddoto (poco interessante, me ne rendo conto...).

Il succo del discorso è che utilizzare le direttive `#include` può diventare complicato quando il programma comincia a diventare più complesso di un semplice algoritmo di calcolo. È per questo che ho scoperto con entusiasmo una direttiva al compilatore, che funziona con Objective C (ed anche col C++, ho scoperto), che sostituisce comodamente la `#include`, ovvero la direttiva

```
#import    <nomefile.h>
```

che ha lo stesso effetto della direttiva `#include`, ma che in più evita di incorporare due volte lo stesso file. Mi spiego meglio: supponiamo di avere un file .m (o .c, non fa differenza) più o meno siffatto:

```
#include    "file1.h"

<<serie di istruzioni C>>

#include "file1.h"

<<ancora istruzioni C>>
```

Succede che il file `file1.h` è inserito due volte. Un compilatore mediamente intelligente produce una quantità di errori mostruosa in sede di compilazione. Utilizzando invece la direttiva `#import`, il precompilatore evita di inserire due volte il file `file1.h`, ed il compilatore è contento (in realtà esiste una tecnica molto semplice ma noiosa per ottenere lo stesso risultato senza la direttiva `#import`, ma visto che c'è questa direttiva, non ne parlo).

Mirko Viviani mi ricorda che il costrutto 'import' funziona teoricamente solo con i file Objective C e che non è necessariamente portabile su altre piattaforme.

C'è un'altra possibilità di utilizzare un'altra classe all'interno di una classe, che chiamerò *certificato di esistenza in vita*. La sintassi è molto più semplice di ogni spiegazione. All'interno del file che usa la classe si scrive

```
@class NomeClasseEsistente ;
```

Questa espressione (che temo essere una direttiva al precompilatore ObjC) informa semplicemente il compilatore (ed il linker e tutto il resto) ed il programmatore che da qualche parte esiste una classe che si chiama `NomeClasseEsistente`. Ora, è chiaro che non è che si possano fare molte cose con una Classe senza avere un'idea di come sia fatta (cioè, senza conoscere la sua interfaccia).

In effetti con questo metodo di certificazione si dichiara l'esistenza di classi che sono utilizzate solo come argomenti di metodi, come variabili d'istanza, eccetera. Se poi dobbiamo utilizzarle realmente, non vedo altre possibilità che importare il file .h corrispondente. Stabilisco quindi la seguente regola (operativa, cioè: finché funziona, la uso; appena vedo che non va, la abbandono): in un file .h che contiene la dichiarazione di una classe, importo sempre il file.h della superclasse, e cerco di utilizzare il più possibile la direttiva `@class` per utilizzare altre classi; in un file .m che contiene la definizione di una classe, importo i file .h delle classi che utilizzo.

I VANTAGGI DELL'INTERFACCIA

L'interfaccia di una classe, il file .h, è insomma tutto ciò che serve per lavorare con questa classe; infatti l'interfaccia:

- specifica la posizione della classe all'interno della gerarchia delle classi;
- contiene le *informazioni d'uso* della classe sotto forma dei metodi associati all'oggetto;
- permette di utilizzare la classe some superclasse di altre classi, informando il programmatore della struttura interna (quali siano le variabili d'istanza).

È per questi motivi (e molti altri) che chi programma pulitamente in ObjC produce un file .h per ogni classe.

LA REALIZZAZIONE

La realizzazione di una classe è più o meno come l'interfaccia. Si comincia col dichiarare tutte le variabili d'istanza, per poi definire i vari metodi, in modo molto simile a quanto visto sopra:

```
@implementation NomeDellaClasse : NomeDellaSuperClasse
{
    dichiarazione delle variabili d'istanza
}
definizione dei metodi
@end
```

Ma, attenzione adesso ai giochi di prestigio:

- invece che ridichiarare le variabili d'istanza, uso il file .h corrispondente, così che non ci possano essere errori di copiatura;
- a questo punto, non devo più dire chi è la superclasse, e tutte le altre cose;

Quindi, ciò che rimane è molto più semplice:

```
#import    "NomeDellaClasse.h"
@implementation NomeDellaClasse
definizione dei metodi
@end
```

La definizione dei metodi somiglia molto una serie di funzioni. Se prima avevamo qualcosa come:

```
- (void) moveToX: (int)coordX moveToY: (int) coordY ;
```

adesso bisogna definirlo (insomma, scriverlo):

```
- (void) moveToX: (int) coordX moveToY: (int) coordY
{
    /* serie di istruzioni in C e ObjC */
}
```

All'interno della definizione dei metodi, si possono utilizzare le variabili d'istanza così come sono, col loro preciso nome. Ad esempio, se abbiamo dichiarato in `PrimaClasse.h`

```
@interface PrimaClasse : NSObject
{
    short    contatore ;
}
- (void) aggiungi ;
@end
```

in `Primaclasse.m` possiamo scrivere:

```
#import    "PrimaClasse.h"
@implementation PrimaClasse
- (void) aggiungi
{
    contatore ++ ;
}
@end
```

A questo punto mi sono stufato. So come si definisce una classe, comincio un po' ad usarle... ma, un momento... non so nulla dell'ambiente di sviluppo...

Capitolo 007 - L'ambiente di sviluppo

Dopo il paradigma Object-oriented e il linguaggio Objective C, finalmente si comincia a toccare qualcosa con mano.

Sorgenti: Qualcosa di originale...

Primo inserimento: 29 Ottobre 2001.

Profonde modifiche per aggiornamento XCode 2.1: 3 Agosto 2005.

Integrazione con maggiori spiegazioni su XCode: 24 settembre 2006.

XCode

`XCode` è l'ambiente integrato di sviluppo (Integrated Development Environment, IDE) utilizzato per scrivere, costruire e verificare la correttezza di programmi da far girare sotto Mac OS X. Utilizzando XCode si possono creare applicazioni, plug-in, driver, framework, salvaschermi, eccetera, utilizzando diversi linguaggi, strumenti e tecnologie. sono compresi degli strumenti per scrivere il codice, modificarlo e analizzarlo; per trasformare il codice in codice eseguibile, verificarne la correttezza, eseguirlo passo passo o tutto d'un fiato.

Uno IDE è una sorta di collante che aiuta lo sviluppatore a mantenere consistente il proprio lavoro, ed allo stesso tempo permette di ignorare alcuni dettagli necessari alla produzione di software. è certamente possibile scrivere, costruire ed eseguire del software senza alcun IDE di supporto. In effetti, una volta non esistevano gli ambienti integrati. Uno aveva un `editor` di testi, un `compilatore`, un `linker` e in qualche caso un `loader`. Nei casi fortunati, c'era anche un `debugger`. Un editor serve per scrivere il programma in uno dei linguaggi noti di programmazione. Un compilatore trasforma il testo scritto in linguaggio macchina, producendo quel che si dice file oggetto. Un linker si preoccupa di raccogliere tutti i file oggetto pertinenti all'applicazione, aggiungerci un po' di librerie e produrre un unico file, l'eseguibile. Il loader serve quando l'eseguibile deve essere lanciato in esecuzione; nella maggior parte dei casi tale funzione è svolta dal sistema operativo. Non sempre le cose funzionano al primo colpo, per cui si usa un debugger, ovvero un programma che, facendo le veci del loader e dell'ambiente operativo, manda in esecuzione l'applicazione permettendo di eseguire una istruzione alla volta e verificare cosa sta succedendo. Tutta questa sequenza di operazioni era organizzata a mano, utilizzando tipicamente programmi funzionanti tramite linea di comando, ricorrendo a convenzioni più o meno arbitrarie e stratagemmi assortiti. Poi sono arrivate le interfacce grafiche e gli ambienti integrati di sviluppo, ed il mondo non è stato più lo stesso.

Aggiungo qualche parola per chi fosse in confusione con i termini compilazione e linking. Come è noto, un calcolatore capisce solo di numeri. Per far eseguire delle operazioni ad un calcolatore, occorre impartirgli istruzioni nel così detto Linguaggio Macchina (che altro non è una opportuna sequenza di numeri). Il primo passo per

rendere comprensibile il tutto ad un essere umano è il linguaggio Assembler; questo è un metodo per scrivere istruzioni macchina in formato testo. Il programma Assemblatore non fa altro che tradurre parola per parola le istruzioni assembler in linguaggio macchina (c'è un rapporto uno ad uno tra le istruzioni in linguaggio assembler e le istruzioni in linguaggio macchina). Un secondo passo è di scrivere le istruzioni in un linguaggio di alto livello, ad esempio, il C. Una volta scritto un programma in C, occorre trasformare le istruzioni C in linguaggio macchina; è il compito del compilatore (questa volta ad una singola istruzione C corrispondono generalmente più e molte istruzioni in linguaggio macchina). L'ho fatta troppo facile: in realtà il linguaggio macchina crudo non può essere eseguito; occorre effettuare una operazione detta appunto di linking (effettuata dal linker), che aggiusta tutto il linguaggio macchina e produce una applicazione eseguibile. È ancora tutto molto semplificato (e a volte anche impreciso), ma per il momento basta così.

Con l'avvento dell'interfaccia grafica totale globale, la linea di comando non era più di moda, per cui si è dovuti per forza passare a strumenti per lo sviluppo che non ragionassero più a linea di comando; per fare questo, è nato il concetto di progetto, già presente anche prima, ma non così esplicito. Un progetto è la collezione di tutti i file che partecipano, a vario titolo, alla costruzione dell'applicazione. L'elenco dei file, che prima era scritto su un foglio di carta di fianco allo schermo a fosfori verdi, adesso è scritto esplicitamente su una finestra a video. Non solo, ma anche le relazioni tra gli stessi file, in pratica tutte le direttive ai compilatori, al linker (quali librerie utilizzare, e perché) e al debugger sono presentate a video, in un file che non contiene altro che riferimenti ad altro file, appunto il file di progetto.

Una volta che tutte queste operazioni possono essere svolte attraverso una interfaccia grafica, ecco che nasce l'ambiente integrato di sviluppo. La cosa bizzarra è che, a parte qualche caso isolato, non è che sia cambiato molto (i programmatori sono fondamentalmente dei conservatori: conosco persone che preferiscono lavorare a linea di comando perché dicono essere più veloci a scrivere un comando piuttosto che fare selezionare una voce di menu). L'ambiente integrato di sviluppo non è altro che un modo elegante di lanciare in esecuzione un editor di testi, per compilare un dato file, per linkare tra loro tutti i file oggetto. Ma, sotto sotto, il lavoro sporco è ancora svolto utilizzando linee di comando.

Arrivando alla fine di questa storia edificante, riassumo dicendo che XCode (ma in generale ogni IDE) non è altro che una comoda interfaccia grafica per il processo di sviluppo di una applicazione: invece di avere anche fare con diversi file sparsi nelle cartelle, XCode permette di raccogliere in un unico posto ed in forma elettronica l'elenco dei documenti che concorrono alla costruzione di una applicazione (sì, anche i documenti di progetto e di help); a partire da questi file, mattoni base dell'applicazione, si indicano le operazioni che devono essere svolte per la produzione dell'applicazione stessa, quindi sono specificate le operazioni di compilazione e di linking; fornisce inoltre supporto per la trattazione dei testi tramite un editor, la documentazione di progetto, con la possibilità di costruire un dizionario dei simboli, facilitando la ricerca di informazioni, e per finire supporta il debugging dell'applicazione con qualcosa in più che lanciare in esecuzione il programma di debugging. Con intuitivi comandi impartiti tramite mouse (ed anche con la tastiera per quelli più noiosi) si possono ristrutturare i legami in maniera semplice e veloce, eseguire operazioni complesse con un clic, cose del genere.

Ma, occorre ribadirlo, XCode non fa quasi alcunché del lavoro grosso. XCode non compila, XCode non costruisce interfacce grafiche, XCode non lancia in esecuzione l'applicazione, XCode non permette di "debuggare" una applicazione. Piuttosto, si appoggia a compilatori esterni per compilare applicazioni, a debugger esterni per eseguire le applicazioni una istruzione alla volta; il tutto, però eseguito in maniera integrata, tanto che non ci si accorge del passaggio tra uno strumento ed un altro.

Gli XCode Tool sono un insieme di strumenti forniti direttamente da Apple. L'installazione del sistema (o qualunque versione pre-installata dello stesso) NON include gli XCode Tools. Si può controllare facilmente se sono già presenti o meno: basta verificare se esiste la cartella `Developer` al livello principale del disco. Se esiste, è verosimile che l'installazione abbia già avuto luogo. Se non esiste, o se volete essere sicuri di installare le ultime versioni disponibili, ci sono due strategie: utilizzate il CD/DVD di installazione del sistema, dove sono sempre riportati, oppure scaricarli dalla rete. Nel primo caso, avete una comoda e veloce possibilità di procedere, ma non siete sicuri di avere l'ultima versione disponibile. Nel secondo caso, preparatevi ad una lunga attesa (l'ultima volta che ho li ho scaricati, erano di più di 900 Mega).

Apple distribuisce gratuitamente gli XCode Tools. Occore solamente registrarsi presso la Apple Developer Connection. Ci sono diverse opzioni; la più semplice è appunto gratuita e non vi da altri diritti se non scaricare dalla rete gratuitamente l'ambiente di sviluppo ed altri file interessanti.

Se non avete una connessione di rete veloce o un amico compiacente, e vi accontentate della versione presente nei dischi del sistema operativo, l'installazione è molto semplice. Nel mio caso (10.4) ho due DVD. Inserendo il primo, Sui dischi si trova una cartella chiamata XCode Tools, all'interno della quale c'è il file `XCodeTools.mpkg`. Doppio clic, e parte una procedura di installazione. Le opzioni di default sono sufficienti ad installare una versione correttamente funzionante di XCode.

Una volta installata, l'applicazione XCode vera e propria si trova nella cartella `/Developer/Applications` della partizione che contiene il sistema operativo. Per mia comodità, ho trascinato l'icona di Xcode sul dock, in modo da averlo sempre ad un clic di distanza.

L'installazione di default inserisce anche tutta (ehm, non credo, ma comunque, una gran parte) la documentazione necessaria per lo sviluppo delle applicazioni. Ancora una volta, la versione più aggiornata si trova in rete, ma XCode non mancherà di avvertirvi (nel caso abbiate una connessione ad internet attiva) che è disponibile un aggiornamento della documentazione stessa.

Prima di aprire l'applicazione, ci sono tre concetti da imparare: `Progetto`, `Target`, `Eseguibile` (Project, Target, Executable).

Il Progetto è il costrutto base di XCode: è la raccolta di tutti i documenti relativi ad un particolare progetto di programmazione. Sono compresi i file sorgente (nei vari linguaggi), le risorse che descrivono l'interfaccia grafica, le librerie, i file di help, eccetera. Un documento di XCode è un progetto.

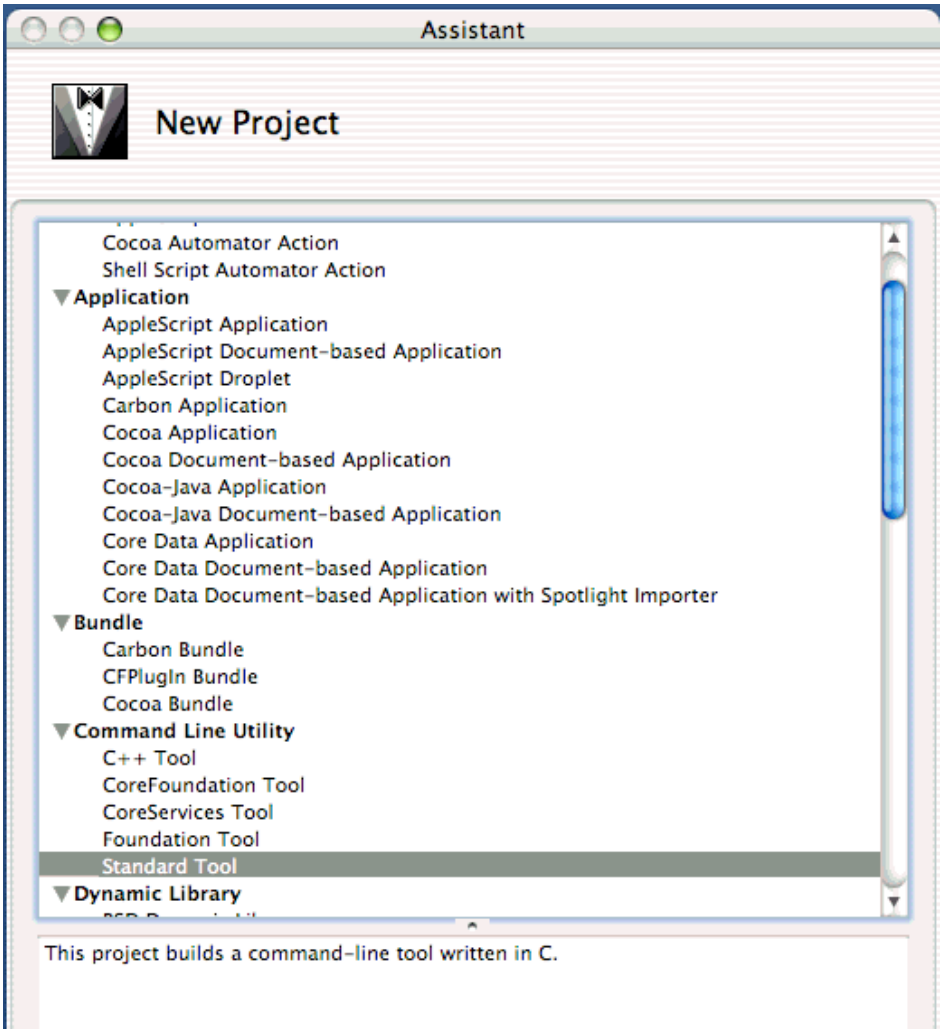
Ogni progetto possiede uno o più obiettivi di realizzazione, i Target; l'obiettivo di un progetto è appunto di produrre qualcosa, ed il target rappresenta questo qualcosa. Un

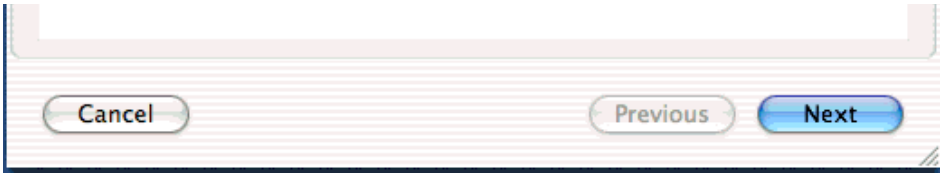
target è definito da una serie di regole per la costruzione dell'applicazione; queste regole indicano, ad esempio, quali file utilizzare per la compilazione, in che ordine, come assemblarli tra loro, eccetera. Ci sono dei casi molto semplici (un file, una applicazione), ma questi sono la norma solo nei primi passi di un programmatore. Generalmente si finisce con l'avere un sacco di file da gestire, ciascuno con le proprie regole. L'insieme di tutte queste regole è appunto un target. Uno stesso progetto può avere diversi target, perché i prodotti finali a cui si può arrivare sono più d'uno. Ad esempio, all'interno di un singolo progetto di programmazione possono trovare luogo diverse realizzazioni, ad esempio una applicazione stand-alone, un plug-in, un comando da eseguire attraverso il terminale, eccetera.

Infine, associato ad ogni Target c'è un eseguibile, ovvero l'applicazione che deve essere lanciata in esecuzione per verificare l'effetto delle operazioni di sviluppo. Normalmente, c'è una associazione diretta tra il target e lo executable, ovvero tra ciò che è stato prodotto dalle operazioni di costruzione dell'applicazione e ciò che si va ad eseguire. Non sempre è così: ci sono dei casi speciali in cui il target non è direttamente eseguibile (ad esempio, un salvaschermo o un pannello delle preferenze). In questi casi, lo Executable è un'altra applicazione (che andrà scelta a mano dal programmatore).

La Finestra di XCode

Ora che abbiamo un'idea di cosa aspettarci, lanciamo XCode e vediamo cosa salta fuori.





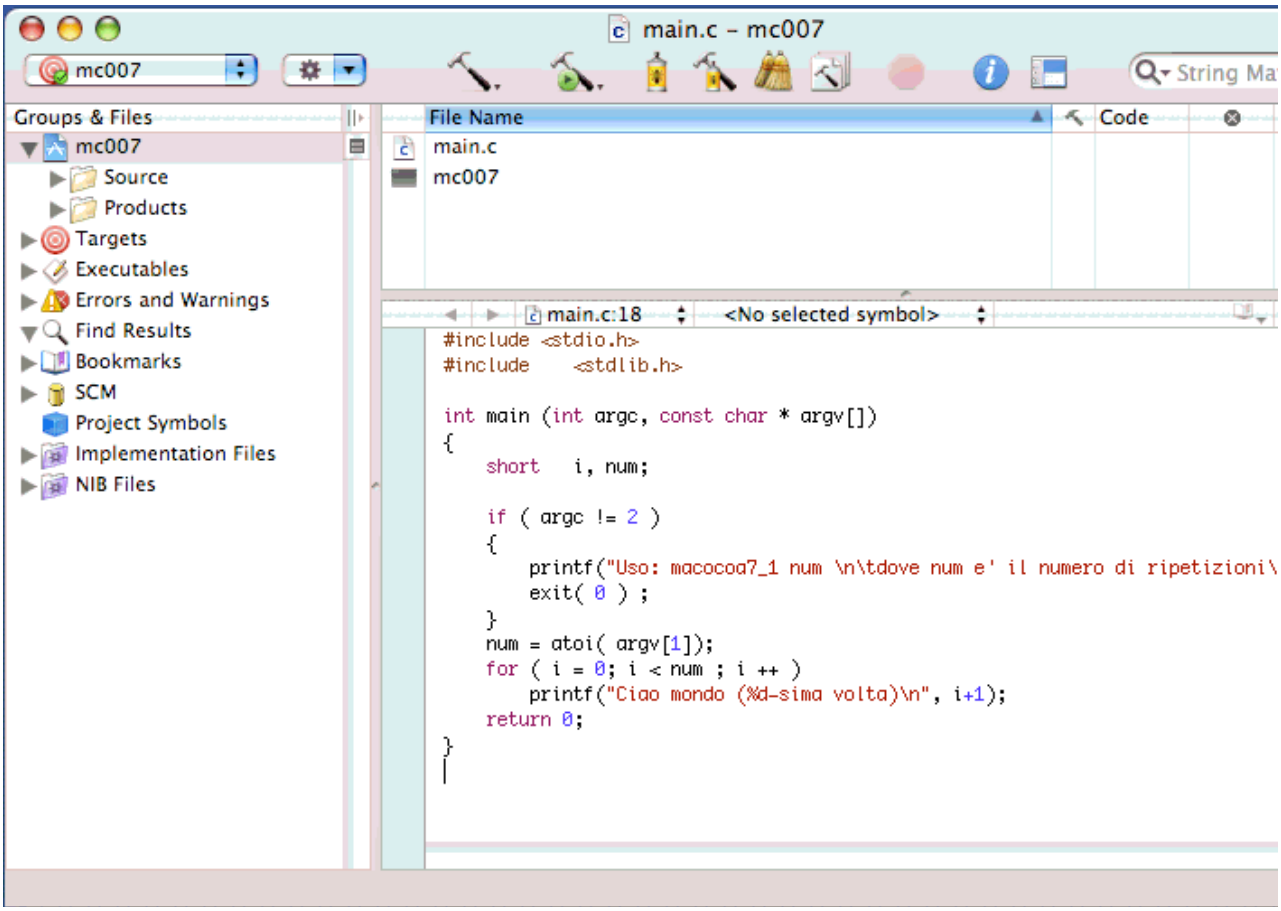
Vedo già il primo problema non appena richiedo tramite menu la creazione di un nuovo progetto. Mi si presenta una finestra con un elenco piuttosto corposo di opzioni.

Cominciare con un progetto interamente vuoto mi pare uno spreco, però riconosco che la maggior parte delle voci non ho idea (beh, non esageriamo, posso fare ipotesi serie...) di cosa siano.

Lo scopo finale sarà costruire una Cocoa Document-based Application, presumendo di passare attraverso una più semplice Cocoa Application, saltando a pie' pari Bundle, Framework e Kernel Extension (per non parlare delle altre cose scritte in Java). Sono interessato anche alla famiglia dei Tool, che presumo essere applicazioni funzionanti a linea di comando o anche meno. Per il momento ignoro tutto il resto e comincio con uno Standard Tool, che mi ispira essere un semplice programma a linea di comando scritto in C.

Infatti.

Dopo aver dato un nome opportuno al progetto (mc007, usando parche quantità di fantasia), quello che salta fuori è un template con il programma 'Ciao Mondo' caro a tutti i programmatori in erba. Un template è il nome comune di uno scheletro precostituito sul quale poi intervenire modificando le parti di interesse.



Vediamo più da vicino la finestra del progetto (che potrebbe differire dalla vostra a causa di diverse versioni di XCode, o semplice personalizzazione dell'interfaccia).

Sulla sinistra c'è una colonna chiamata Group & Files. In cima c'è il progetto, all'interno del quale sono compresi i vari file che gli appartengono. Questi sono organizzati attraverso una serie di cartelle. Attenzione che l'organizzazione non è necessariamente quella dei file sul disco rigido, anzi, generalmente è diversa. Oltretutto, si tratta di collegamenti, i file esistono al di fuori di XCode, ed ogni modifica con altri mezzi esterni a XCode si riflette all'interno di XCode. Facendo doppio clic su uno di questi elementi, si apre un editor adatto al tipo di file.

Gli altri gruppi di elementi (mi veniva da scrivere oggetti, ma adesso questa bella parola generica che era oggetti bisogna riservarla agli oggetti della programmazione OOP) sono chiamati `Smart Groups` e sono di natura diversa; in comune hanno il fatto che sono organizzati in maniera automatica e che si aggiornano per riflettere lo stato del progetto. I gruppi più importanti sono il gruppo che raccoglie i Target del progetto e quello degli Executables. Gli altri gruppi (che possono anche non essere presenti, in dipendenza della configurabilità di XCode) raccolgono informazioni relative al progetto: l'elenco degli errori di compilazione, i risultati delle operazioni di ricerca, segnalibri, eccetera.

Il contenuto del resto della finestra dipende dalla selezione operata nella colonna di destra. Ad esempio, selezionando il progetto, la finestra è divisa in due parti (magari si deve ridimensionare il pannello): in quella superiore sono riportati i file del progetto, e nella parte inferiore, selezionando uno di questi file, si presenta (se disponibile) un editor per quel tipo di file.

L'editor di testo è senz'altro quello più utilizzato per la scrittura dei programmi. Si tratta di un onesto editor di testo, con le tipiche caratteristiche di un editor per programmazione; caratteristica interessante (ma effettive su macchine con adeguati processori) è l'autocompletamento: cominciando a scrivere il nome di un metodo o funzione, l'editor propone una serie di alternative possibili, suggerendo anche la sintassi. L'editor può aprirsi all'interno della finestra principale, o anche su una finestra a parte; è anche possibile scegliere un editor esterno.

Per completare (si fa per dire) l'esplorazione iniziale, rimangono da considerare i menu e le icone presenti all'interno della finestra principale. Queste icone duplicano alcune funzioni disponibili sui menu, quindi non ne parlerò molto. Tra i menu, ci sono i soliti `File`, `Edit`, `Format` e `Find`, che servono alle operazioni base sui file, alla manipolazione del testo e le funzioni accessorie. C'è poi il menu `View`, che serve a riarrangiare le viste all'interno delle finestre, ed il menu `Windows` seleziona le finestre stesse. I tre menu più importanti sono `Project`, `Build` e `Debug`. Con il menu Project si gestisce il progetto: ci sono comandi per aggiungere e togliere file (la riorganizzazione si fa facile con drag'n'drop), e per configurare i target e gli executable (e le build phases, che al momento ignoro). Con il menu Build si passa all'esecuzione delel regole, e quindi alla trasformazione del progetto in un eseguibile: ci sono i vari comandi per compilare, linkare e debuggare. Questo si svolge all'interno del menu Debug, che è molto più vivace quando è attivo il debugger (che, per chi sa, è gdb con una facciata in stile Macintosh). Ignoro i menu `Design` e `SCM` (che è piuttosto scarno), e ricordo l'esistenza del menu

`Help`, che apre le pagine dello help di XCode, ma anche dell'intera documentazione Apple su Cocoa (e Carbon e tutto il resto).

Come dicevo, i pulsanti all'interno della finestra duplicano molte voci del menu, e la cui descrizione puntuale ci porterebbe troppo lontano; potete giocarci un po', oppure consultare il manuale di XCode, che vi spiegherà con ogni dettaglio la funzione di tutto.

SCRIVO IL PROGRAMMA

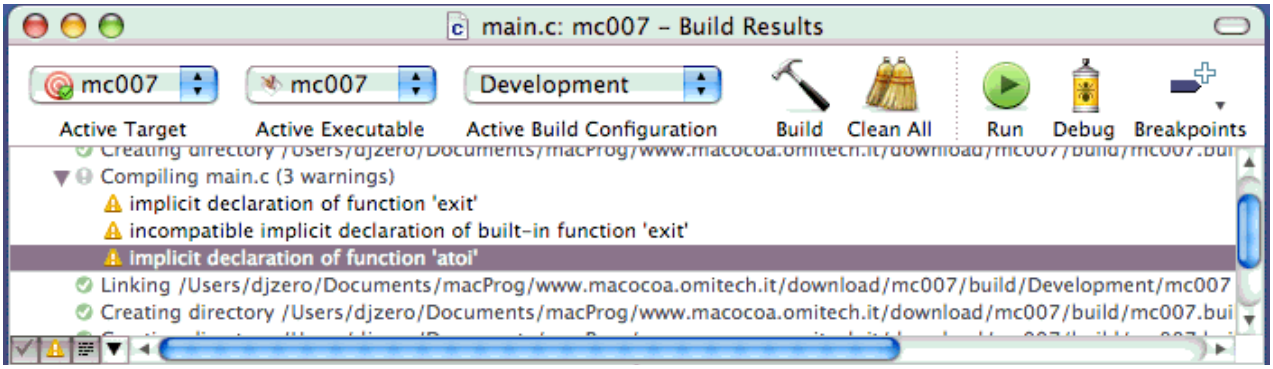
Tutto ciò è molto nebuloso, quindi vediamo in pratica come operare. Abbiamo già pronto, perché il template ce lo ha fornito, un file sorgente `main.c`; si tratta del solito programma 'ciao mondo' che è tradizione scrivere la prima volta che si accede ad un nuovo ambiente di sviluppo. Possiamo fare delle modifiche, siamo all'interno di file dove scrivere istruzioni C, quindi possiamo fare qualche semplice modifica qui e li. Per provare meglio ambienti in cui è possibile utilizzare la linea di comando, per tradizione scrivo `ciao mondo` tante volte quante indicate da un parametro di linea di comando.

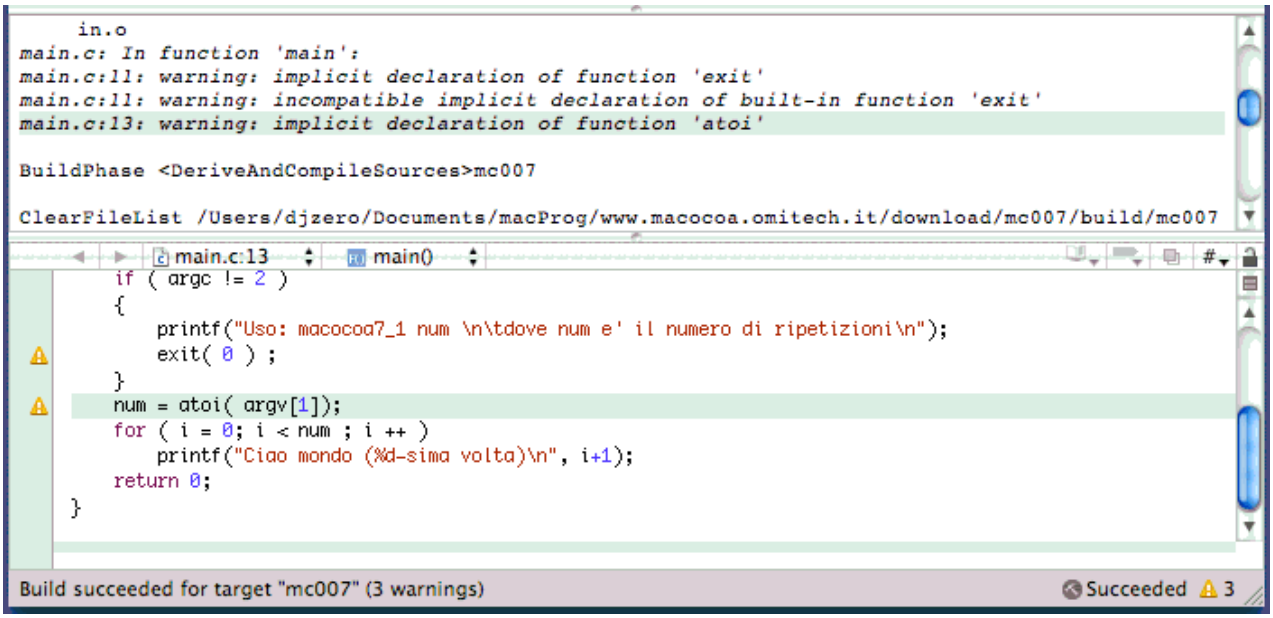
Scriverò quindi il seguente programma, piuttosto stupido:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    short    i, num;

    if ( argc != 2 )
    {
        printf("Uso: mc007 num \n\tdove num e' il numero di ripetizioni\n");
        exit( 0 ) ;
    }
    num = atoi( argv[1]);
    for ( i = 0; i < num ; i ++ )
        printf("Ciao mondo (%d-sima volta)\n", i+1);
    return 0;
}
```

Fatto questo, procedo a compilare il file, o meglio, a fare il Build dell'intero progetto. La procedura di build consiste nell'insieme delel operazioni di compilazione dei file sorgente, seguita poi dall'operazione di linking. C'è un apposito comando di menu, oppure l'icona con un martello.

Non sembra che l'operazione renda felice XCode; vado a vedere la finestra di Build, in cui sono riportate appunto le informazioni per la compilazione ed il linking.





Il succo della faccenda è che il file `main.c` è compilato, e sono riportati tre **warning** (qualcosa di molto meno grave di un errore, che non impedisce in linea di principio il funzionamento del programma, ma che io, paranoico come sono, non accetto nella maniera più assoluta) che riguarda, ad esempio, la dichiarazione della funzione `atoi`, da me usata ma mai dichiarata (e tanto meno definita). `atoi`, come tutti sanno, è una funzione che trasforma il contenuto di una stringa in un numero intero, ed è una classica funzione di libreria del linguaggio C.

Il warning è riportato, molto pulitamente, nel pannello superiore. Nella finestra di mezzo lo stesso warning è evidenziato attraverso i messaggi tipici (e criptici) del compilatore, mentre nel terzo pannello è visualizzato il codice sorgente. Facendo clic sul messaggio di zarning warning, nella finestra del codice sorgente è evidenziata la riga che offende il compilatore.

Sciocco che sono, ho dimenticato di includere il file `stdlib.h`; vado a correggere il file `main.c`, aggiungendo coma seconda riga l'ulteriore direttiva

```
    #include <stdlib.h>
```

e ricompilo. Perfetto, nemmeno un warning (ed in effetti, anche gli altri due warning dipendendo dalla mancata inclusione del file).

Siamo pronti per lanciare in esecuzione il programma. Si utilizza il comando *Run Executable* o l'icona equivalente (un tondo verde con triangolo nero interno, non presente nella mia configurazione, ma che potete comodamente inserire nella toolbar). Si apre la finestra `Run Log`, e all'interno compare la seguente scritta:

```
Uso: mc007 num
    dove num e' il numero di ripetizioni

mc007 has exited with status 0.
```

Sembra sbagliato, ma in realtà è giusto. Non avendo passato alcun parametro al programma, il programma ha stampato il messaggio di avvertimento su come usarlo, e

non ha fatto altro.

Per usare la linea di comando, bisogna aprire il *Terminale*, ed andare a ragionare con la linea di comando. Bene, cercate `Terminale` nella applicazioni, lanciatelo, e portatevi, pieni di buona volontà, nella directory corretta. La mia (il mio utente è djzeroOO, il vostro non lo so) è la seguente

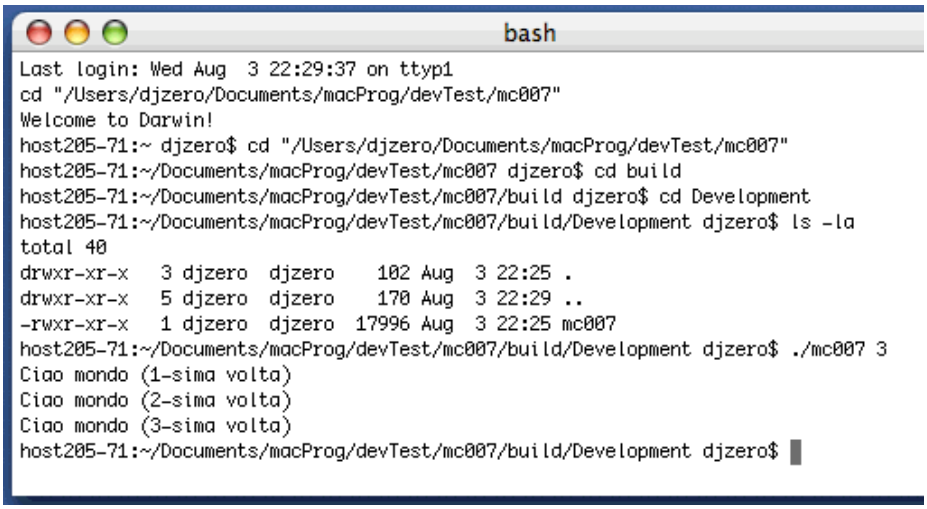
```
/Users/djzero00/Documents/macProg/devTest/mc007
```

La directory parte dal posto dove avete detto di conservare il progetto (per me, `mc007`), e scende ulteriormente di un paio di livelli, dentro la cartella `build/Development`, dove sono appunto conservati i risultati della compilazione.

Ciò che ci interessa è l'eseguibile `mc007`. Per lanciare in esecuzione il programma, si tratta di fare semplicemente `mc007`. Lo faccio, e non funziona... (salta fuori `mc007: Command not found`.) Ah, per forza, il path non è corretto. Scrivo

```
./mc007 3
```

e tutto funziona.



Mi fermo e spiego: quando si ha a che fare con un sistema operativo che rende disponibile una linea di comando (Unix, in primis, e Dos che lo ha copiato...), molta

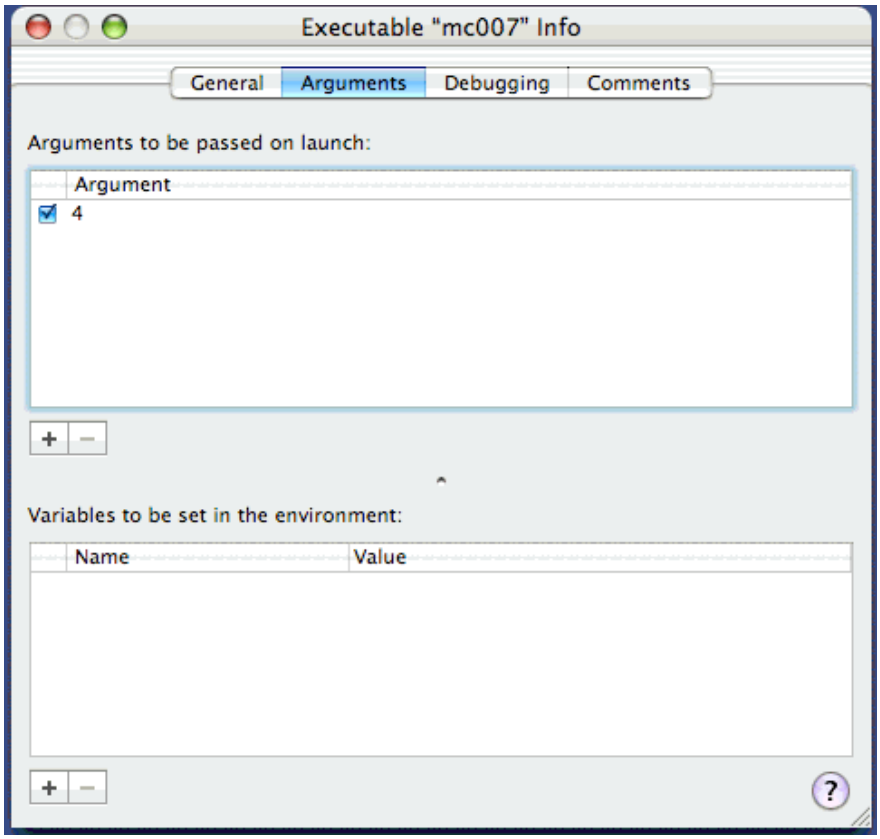
attenzione deve essere posta sui *path*. Un programma è tipicamente eseguito solamente se individuato univocamente non solo dal suo nome, ma dal suo percorso completo. Ovviamente, è molto noioso scrivere ogni volta il percorso completo, per cui sono state create alcune facilitazioni; la prima facilitazione è l'individuazione di alcune cartelle in cui cercare comandi sconosciuti. Queste cartelle sono conservate nella variabile (del terminale) `PATH`. Potete vedere quali sono le cartelle individuate guardando il valore di questa variabile con il comando io ottengo qualcosa del genere:

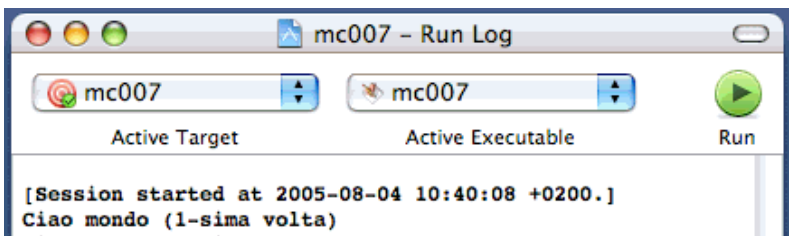
```
DjZeroPB:~ djzero00$ echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
DjZeroPB:~ djzero00$
```

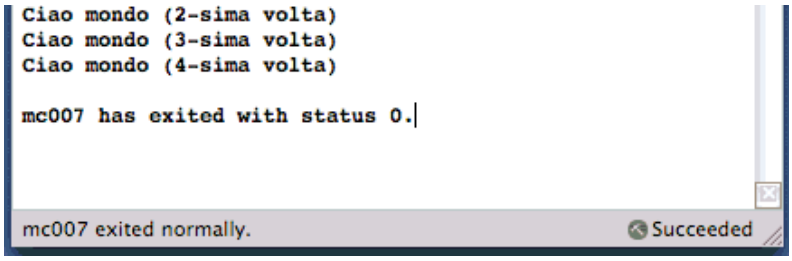
La directory in cui si trova l'eseguibile non compare nell'elenco, quindi, niente esecuzione. Sono quindi costretto a specificare il percorso completo, ma è un po' noioso. Uso allora la seguente scorciatoia, sempre disponibile in un ambiente Unix: esiste la possibilità di indicare "il percorso in cui mi trovo adesso" attraverso il carattere "." (solo il punto). A questo punto, scrivo solamente

```
./mc007 3
```

e tutto comincia a funzionare.







Potete provare con altri numeri, ma non otterrete nulla di più eccitante. Torniamo allora dentro XCode. Esiste un metodo per passare argomenti al programma senza dover passare attraverso il terminale: si tratta di attribuire degli ooportuni valori alle proprietà dell'eseguibile. Per farlo, si seleziona la voce *Edit Active Executable mc007* dal menu *Project*. Si ottiene così una finestra divisa in quattro tab. Selezionando il tab *Arguments*, è possibile aggiungere argomenti di lancio al programma, come abbiamo fatto da linea di comando. Inserisco 4 con il pulsante +. Torno allora ad eseguire il programma. La finestra `Run Log` si attiva e mostra il nuovo risultato. Meraviglia.

Sono arrivato a completare uno dei passi fondamentali nell'avvicinare un nuovo ambiente operativo. Ho preparato un file nel linguaggio di programmazione, l'ho compilato, l'ho eseguito. Fatto questo, tutto il resto sono solo dettagli che si aggiungono.

Capitolo 008 - Un programma in Objective C

Un piccolo programma finalmente Object Oriented.

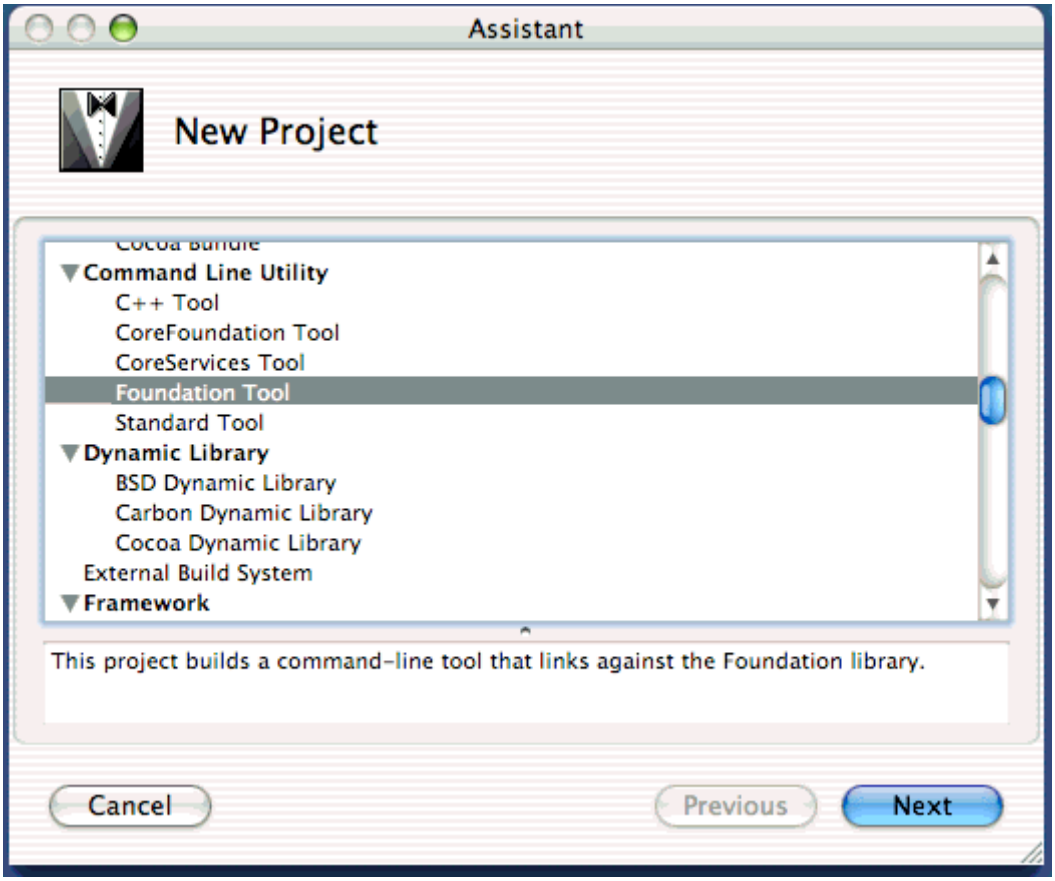
Sorgenti: Tutta farina del mio sacco (si fa per dire...)

Primo inserimento: 5 Novembre 2001

Ritocchi per aggiornamento XCode 2.1: 5 Agosto 2005

CONTATORE

Questa volta proviamo a scrivere un'applicazione in Objective C, ma senza interfaccia utente; quindi, ancora una volta, un tool che si utilizza tramite la linea di comando. Ma il programma sarà scritto in Objective C e presenta una classe.



L'idea è di fare un contatore, ovvero un oggetto che contenga al proprio interno un valore che può essere incrementato, decrementato, letto o impostato ad una dato valore. Nulla di speciale, in effetti, anzi, un metodo più complicato per una cosa che in C si farebbe con una semplice variabile. Però, ci interessa giocare col concetto.

Cominciamo quindi col fare il progetto; apro XCode e scelgo, nella solita finestra dei template di progetto, un **Foundation Tool**; questo template dovrebbe comprendere il

Foundation Framework, che contiene quindi tutte quelle classi che ci occorrono per lavorare. Manca l'*Application Framework*, le classi dell'interfaccia utente, ma ho appena stabilito che ne faccio a meno.

XCode propone quindi un file `main.m` molto semplice:

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // insert code here...
    NSLog(@"Hello, World!");
    [pool release];
    return 0;
}
```

Ignorerò quelle righe misteriose che parlano dell'oggetto `pool`, ed approfondisco quella strana cosa che si chiama `NSLog`. Cerco sul manuale (fa parte delle *Foundation Functions*) e trovo una cosa piuttosto interessante.

LA FUNZIONE NSLOG

`NSLog` è una funzione che scrive una `NSString` sullo `standard error`. Comincio dalle cose che so: nei sistemi Unix ogni programma ha tre file predefiniti sui quali può operare. Questi tre file si chiamano *standard input*, *standard output* e *standard error* oppure, per gli amici, *stdin*, *stdout*, *stderr*. In Unix, qualsiasi cosa è un file, ed ogni risorsa disponibile sul computer è vista come un file; non meraviglia quindi che un dispositivo di input quale la tastiera è mappato in un file denominato `stdin`, l'uscita a video di un programma è mappato su un file chiamato `stdout`, e gli errori sono scritti in un file chiamato `stderr`. Poi, tutto quanto è mescolato assieme, per cui `stdin` è in pratica rappresentato dai caratteri immessi dalla linea di comando, `stdout` sono i caratteri che il programma scrive sulla stessa linea di comando, e `stderr` va ancora una volta a finire sulla linea di comando... (ma ci sono tecniche per farne altro, ad esempio, scrivere il tutto in un vero file...).

`NSLog` accetta un numero variabile di argomenti. Il primo deve essere un oggetto di tipo `NSString` con al proprio interno dei codici di formattazione; questi codici devono corrispondere poi agli altri argomenti. Per farla breve, funziona più o meno come la funzione di libreria standard del C `printf`, tanto cara ed utile a tutti i programmatori, ma usa NSString al posto delle stringhe C (e qualche altra piccola differenza).

Parentesi. Ed allora, perché non uso `printf` e non ci penso più? Ma perché sto programmando in Objective C e Cocoa, ed il programma non utilizza in linea teorica la libreria standard del C che mette a disposizione la funzione `printf`. Chiusa parentesi.

Vediamo a questo punto cosa diavolo è una `NSString`: il sospetto che si tratti di una stringa (insieme di caratteri) è forte. Infatti. Trovo pagine e pagine su `NSString`, a quanto pare uno degli oggetti più importanti del Foundation Framework (ma le stringhe sono in generale uno dei costrutti più importanti di ogni linguaggio di

programmazione...). Senza approfondire la vita ed i miracoli dell'oggetto `NSString`, vedo che quest'oggetto conserva nelle sue variabili d'istanza stringhe di testo rappresentate secondo la codifica *Unicode* (e quindi tratta tranquillamente caratteri in tutte le lingue del mondo), ed ha a corredo una magnifica teoria di metodi adatti a molte bisogna. Cito alla rinfusa metodi per accedere ai singoli caratteri, per sapere quanto è lunga la stringa, per assegnare il valore a partire dai costrutti più disparati, per combinarne più assieme, per spezzarle in diverse altre `NSString`, per fare ricerche e confronti, estrarne numeri; di più, ci sono tutta una serie di metodi da usare quando la stringa è in realtà un path, un url o comunque un nome di file completo.

Finalmente arriviamo alla cosa più esotica, ovvero il costrutto `@""`. Si tratta di un metodo di convenienza supportato dal compilatore ObjC per la costruizione di oggetti NSString. Con questo operatore è possibile definire al volo oggetti di tipo `NSString`; per il momento non serve sapere altro.

PROVIAMO NSLOG

Riassumendo: con `NSLog` posso scrivere delle stringhe sulla riga di comando; queste stringhe sono oggetti di tipo `NSString`, che si possono definire in linea coll'operatore `@""`. La stringa può contenere codici di controllo, funzionando come una `printf` della libreria standard del C. La provo subito, e faccio un programma piuttosto stupido.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // insert code here...
    float    a = 10.34 ;
    int        b = -76546 ;
    char *    c = "salve a tutti";

    NSLog(@"______1234567890_1234567890123456_");
    NSLog(@"aloha_%10.5f_%16d_%.10s_", a, b, c);
    [pool release];
    return 0;
}
```

La prima chiamata a `NSLog` mi serve per mettere una sorta di righello, per controllare che le specifiche di formattazione dei campi nella seconda `NSLog` corrispondano. Infatti nella seconda `NSLog` dico di scrivere un numero `float` in dieci colonne utilizzando cinque cifre decimali, un intero su sedici colonne, e poi una stringa C normale. tutto torna: infatti il programma produce:

```
[Session started at 2005-08-05 17:41:28 +0200.]
2005-08-05 17:41:29.717 mc008[1122] ______1234567890_1234567890123456_
2005-08-05 17:41:29.719 mc008[1122] aloha_  10.34000_           -76546_salve
a tu_

mc008_1 has exited with status 0.
```

Il che ci permette di fare un veloce commento: `NSLog` aggiunge, di suo, davanti al

messaggio che gli diciamo di scrivere, la data, l'ora, il nome del programma ed il suo `pid` (process identifier, l'identificatore numerico del programma).

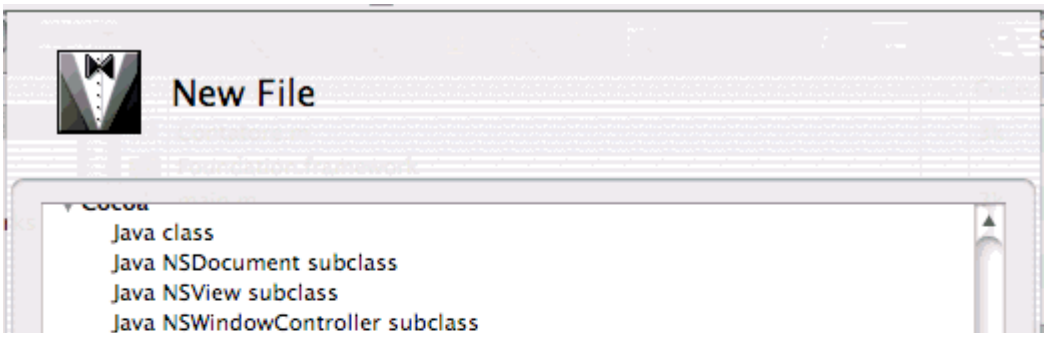
Come per la funzione `printf` dello standard C, nella stringa di formato si possono inserire delle sequenze di controllo che cominciano col carattere %, seguite da un carattere che indica il tipo di valore da scrivere. Ricordo, ad esempio, la lettera "d" per scrivere valori decimali, la lettera "f" per numeri floating point, eccetera (ogni buon manuale di C vi illustrerà la cosa nei dettagli, dilungandosi sulla possibilità di specificare un formato, un'ampiezza, bazzeccole del genere.

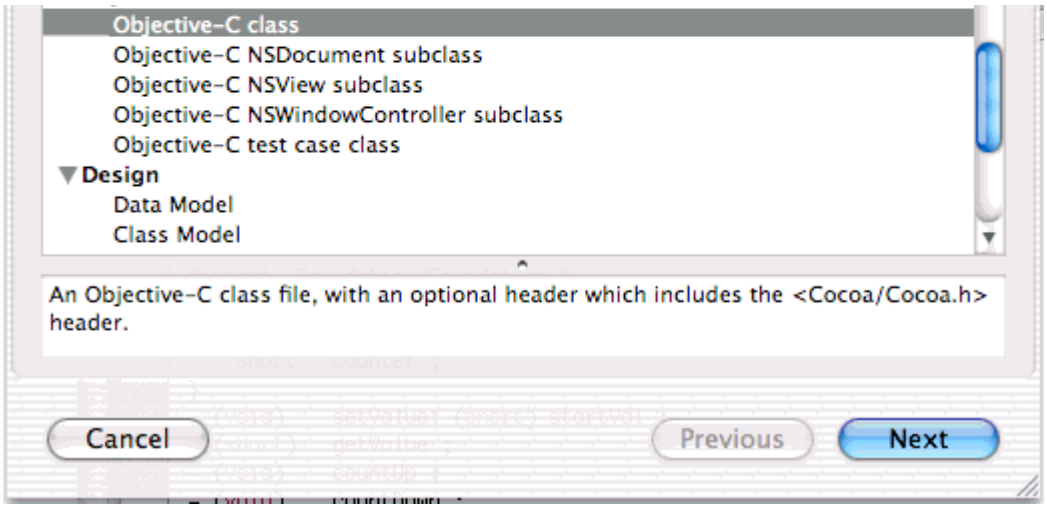
Di suo, NSLog aggiunge la possibilità di scrivere "Oggetti", ed in particolare NSString, con la specifica di formato data dal carattere "@". Come mi precisa Mirko Viviani, nel caso di oggetti, i caratteri "%@" producono come risultato della valutazione di formato una stringa che rappresenta il risultato del metodo `description` applicato all'oggetto stesso. Tale metodo restituisce per gli oggetti *semplici* (numeri e stringhe...) il valore dell'oggetto stesso, mentre per oggetti più complicati (ad esempio NSArray) descrizioni più complesse. È ovviamente possibile, per la classi definite dall'utente, definire il metodo `description` in maniera più o meno fantasiosa ed utilizzarlo come strumento di debug.

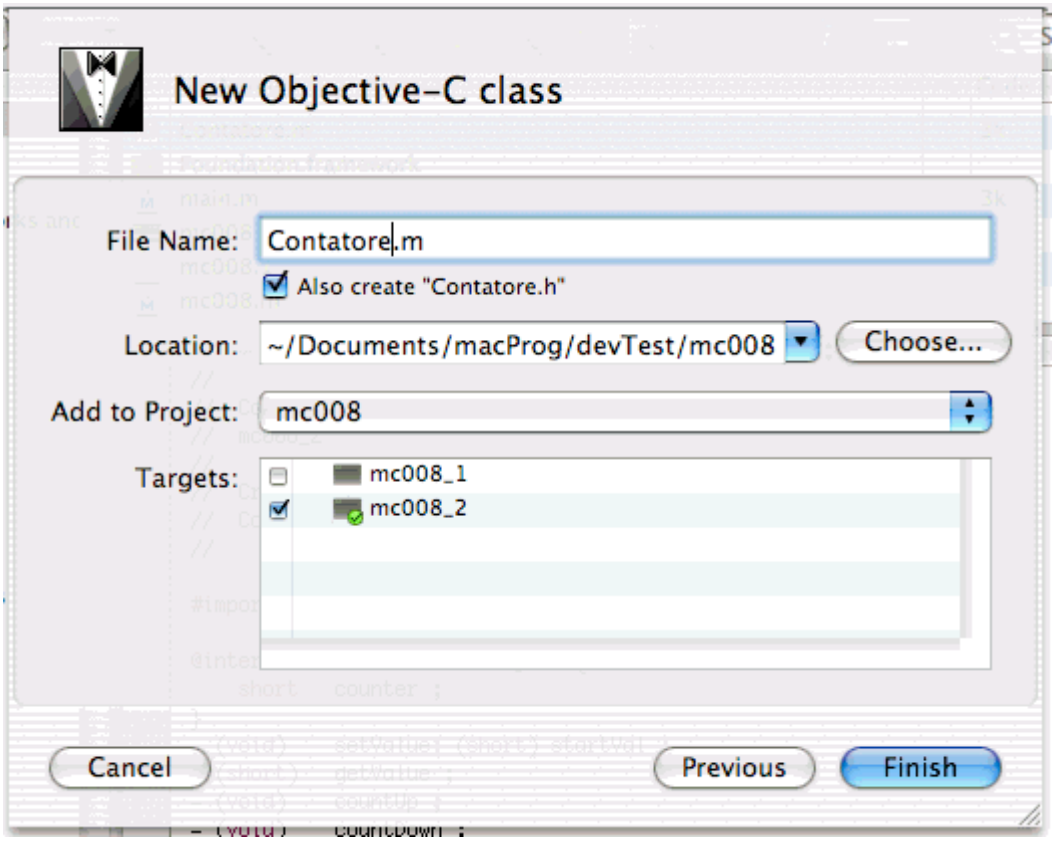
C'è una cosa da notare, e che mi era ovviamente sfuggita. `NSLog` scrive la stringa sullo `stderr`, che NON è ridiretto di default sulla linea di comando, ma sulla **Console**. Ora, qualche ulteriore parola va impiegata. Finché, come abbiamo fatto, il nostro progetto non è una applicazione Cocoa, va fatta partire da linea di comando; `stdin`, `stdout` e `stderr` coincidono (a meno di ridirezione esplicita... una cosa che ci porta troppo lontano...) con la linea di comando, per cui si vedrà tutto tranquillamente sul *Terminale* aperto. Quando costruisco una applicazione Cocoa, se la lancio con un doppio clic e comunque non da linea di comando, `stdin`, `stdout` e `stderr` perdono di significato (non ho un Terminale cui fare riferimento). In questi casi, quando qualche applicazione intende scrivere qualcosa, esiste appunto un file, `console.log`, dove va a finire tutto quanto. Per vedere questo file, un metodo semplice è di utilizzare l'applicazione `Console`, nella cartella `/Applicazioni/Utilities`. Lanciando `Console`, trovate un sacco di messaggi (molta roba vecchia, ma anche recente, messaggi di errori sconosciuti, porcherie sparse...). Quando faremo un'applicazione Cocoa lanciata da doppio clic, qui compaiono le scritte di `NSLog`.

Non è finita: se lancio l'applicazione scritta con Cocoa direttamente da Terminale (si può fare...), `stderr` ritorna sul Terminale...

L'APPLICAZIONE







Torniamo a bomba, ovvero alla costruzione di una applicazione in Objective C. Bisogna definire una classe *Contatore*, dichiarare le variabili d'istanza e definire i metodi. La variabile d'istanza sarà una sola, ovvero una variabile che contiene un numero intero che fa da contatore. Per quanto riguarda i metodi, ci occorre un metodo per leggere il valore del contatore, uno per impostarlo, uno per incrementare il contatore ed uno per decremtnare il contatore. Facciamo partire il Xcode e scegliamo il menu `New File...` Oh, che bello, mi mostra un dialogo che mi chiede il tipo di file che voglio fare. Mi pare ovvio che sceglierò una Classe in Objective C. Il risultato dell'operazione, una volta inserite le informazioni richieste (mi va bene quelle di default, ovvero costruire anche il file .h corrispondente, l'inserimento nel progetto e nel target attivo), è particolarmente carino. Xcode ha infatto già creato per me due file, il .m ed il .h, già riempiti con dei suggerimenti per la costruzione della classe.

Ecco quindi, molto semplicemente, i due file. Il file `Contatore.h`

```
#import <Foundation/Foundation.h>

@interface Contatore : NSObject {
    short    counter ;
}
- (void)    setValue: (short) startVal ;
- (short)    getValue ;
- (void)    countUp ;
- (void)    countDown ;
@end
```

e questo è il file `Contatore.m`:

```
#import "Contatore.h"

@implementation Contatore
-(void) setValue: (short) valore
{
    counter = valore ;
}

-(void) countUp
{
    counter ++ ;
}


-(void) countDown ;
{
    counter -- ;
}

- (short) getValue
{
    return ( counter );
}
@end
```

Utilizzo questa classe in maniera semplice, nel file `main.m`

```
#import <Foundation/Foundation.h>
#import    "Contatore.h"

int main (int argc, const char * argv[])
{
    // lascio stare questa istruzione, che non so cosa sia...
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // definisco il mio oggetto
    Contatore    * miocont ;

    // diciamo che stiamo per partire...
    NSLog(@"La mia prima classe");
    // costruisco ed inizializzo il mio oggetto
    miocont = [[Contatore alloc] init ];
    // utilizzo un metodo per vedere cosa c'e' dentro
    NSLog( @"Valore = %d\n", [miocont getValue ]);
    // adesso imposto il suo valore a 10
    [ miocont setValue: 10 ];
```

```
    // controllo che sia proprio cosi'
    NSLog( @"Valore = %d\n", [miocont getValue ]);
    // poi a venti...
    [ miocont setValue: 20 ];
    NSLog( @"Valore = %d\n", [miocont getValue ]);
    // incremento il contatore
     [ miocont countUp ];
    NSLog( @"Valore = %d\n", [miocont getValue ]);
    // lo descremento
     [ miocont countDown ];
    NSLog( @"Valore = %d\n", [miocont getValue ]);

    // lascio il resto com'e'
    [pool release];
    return 0;
}
```

Compilo (neppure un errore!) l'intero progetto ed eseguo. Ottengo:

```
Nov 02 14:01:49 mc008_2[457] La mia prima classe
Nov 02 14:01:49 mc008_2[457] Valore = 0
Nov 02 14:01:49 mc008_2[457] Valore = 10
Nov 02 14:01:49 mc008_2[457] Valore = 20
Nov 02 14:01:49 mc008_2[457] Valore = 21
Nov 02 14:01:49 mc008_2[457] Valore = 20

mc008_2 has exited with status 0.
```

Ottimo. Tutto funziona. Fin troppo facile.

Adesso, apriamo una finestra e visualizziamo questo contatore da qualche parte...

Temo che non sarà una cosa immediata...

Capitolo 009 - Finalmente Cocoa

Dove il romanzo giustifica il suo titolo (epigrafe del *Capitan Fracassa*).

Sorgenti: C'era un tutorial dal nome *Currency Converter Tutorial* all'interno della documentazione dei Developer's Tools; adesso si trova tutto all'interno di Reference Library / Cocoa Fundamentals / Objective-C, *Developing Cocoa Objective-C Applications: A Tutorial*

Primo inserimento: 12 Novembre 2001

Ritocchi per aggiornamento XCode 2.1: 8 Agosto 2005

Finalmente Cocoa

È tempo di scrivere la prima applicazione Cocoa. Riprendo la classe `Contatore`: voglio metterci una bella (!) interfaccia grafica. Ho quindi una finestra all'interno della quale visualizzo il valore corrente del contatore, più due pulsanti, uno per incrementare il contatore, e l'altro per decrementarlo. Fine. Prima, però, bisogna parlare di **Interface Builder**.

Interface Builder

Interface Builder, per gli amici IB, è una applicazione per costruire l'interfaccia di applicazioni. È una sorta di ambiente visuale per la costruzione di applicazioni, che si integra facilmente con XCode.

Con IB si costruisce l'interfaccia pigliando gli elementi dell'interfaccia, pulsanti, campi di testo, caselle, eccetera, da una serie di palette, e trascinandoli sulla finestra principale della costruenda applicazione. C'è la possibilità di definire nuove finestre, menù e tutto quanto concorre alla rappresentazione grafica di una applicazione. Di più, si possono inserire alcuni legami tra i vari elementi dell'interfaccia, in modo che operando su un elemento accada qualcosa sugli elementi collegati. In altre parole, c'è la possibilità in IB di mandare messaggi tra i vari oggetti che costruiscono l'interfaccia.

File NIB

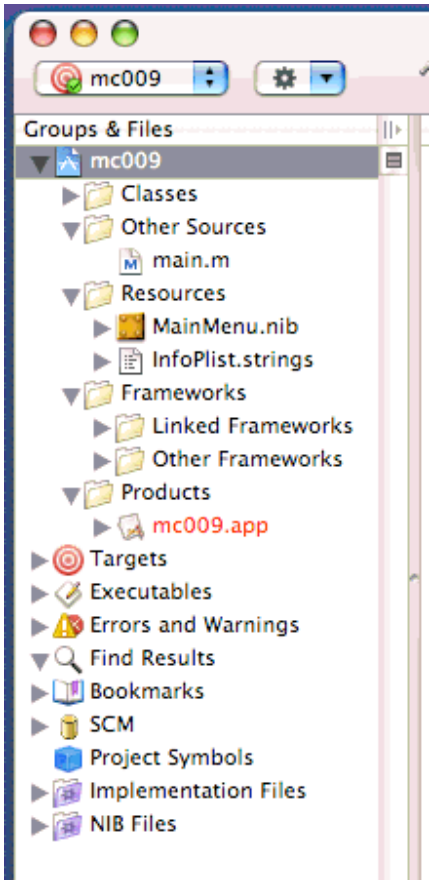
Abbiamo alcuni concetti da esplorare prima di avventurarci in pratica. Il primo concetto è il file **NIB** (che dovrebbe stare per **Nextstep Interface Builder**). Un file nib è associato ad una finestra, o meglio, è un archivio di oggetti generati attraverso IB. Ovvero, IB salva tutto ciò che è in grado di manipolare all'interno di un file nib. Un file nib in genere raggruppa una serie di elementi (oggetti) dell'interfaccia che fanno logicamente parte della stessa parte di programma: mi figuro che una finestra, con tutti i controlli relativi, sia contenuta all'interno di un file nib. Esiste un file nib principale, aperto al lancio dell'applicazione, che contiene il menu dell'applicazione e, verosimilmente, la finestra principale. Pare che convenga dividere le varie finestre di una applicazione in diversi file nib, in modo che siano caricati solo all'occorrenza e non sempre.

All'interno di un file nib possono essere conservati anche gli outlet (ne abbiamo già parlato), ovvero delle variabili che conservano un riferimento ad un altro oggetto dell'interfaccia.

TARGET/ACTION

Il secondo concetto che ci occorre è quello di **Target/Action**. Tipicamente, quando ho un controllo all'interno di una interfaccia (diciamo un pulsante), l'attivazione di questo controllo richiede che sia eseguita una qualche operazione. All'interno del paradigma Object Oriented, stiamo parlando di un oggetto (il pulsante), che, in risposta ad una azione dell'utente, invia un messaggio opportuno a qualche altro oggetto, che si preoccuperà di portare avanti le operazioni. L'oggetto destinatario è il target dell'azione, e l'azione è in pratica il messaggio comunicato. IB mette a disposizione un metodo grafico per realizzare questi collegamenti.

UNA APPLICAZIONE COCOA

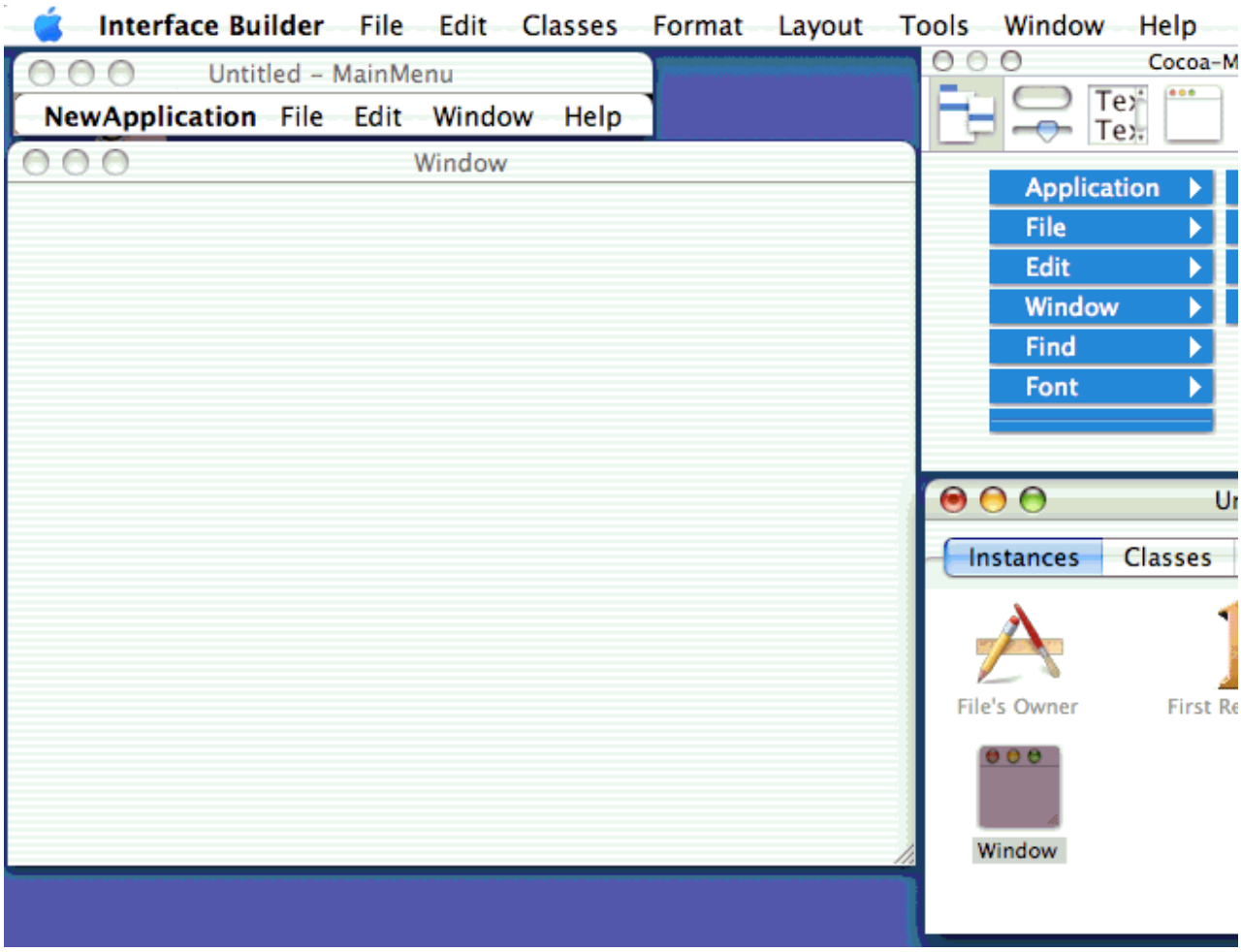


A questo punto, parto con la nostra applicazione. Apro Xcode e specifico un nuovo progetto, questa volta una `Cocoa Application`. Xcode, di suo, ci mette un po' di cartelle e qualche file predefinito. Ho una cartella `Classes` dove sono (saranno) inserite le Classi da me definite e che costituiscono l'applicazione. Nella cartella `Other Sources` c'è il file `main.c`. A vederlo, non è che ci sia molto da modificare, per il momento (è una riga sola):

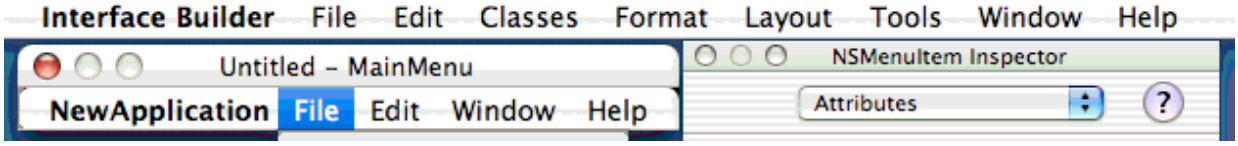
```
#import <Cocoa/Cocoa.h>

int main(int argc, const char *argv[])
{
    return NSApplicationMain(argc, argv);
}
```

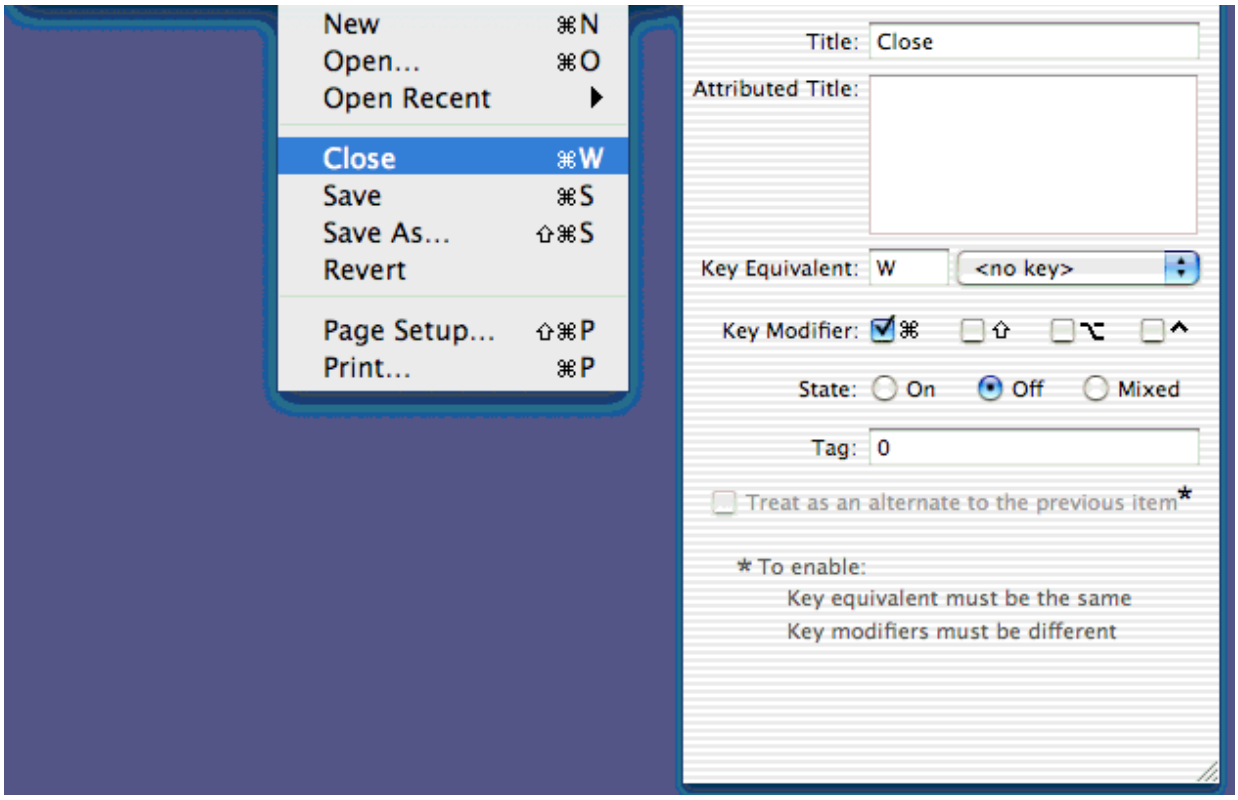
La cartella `Framework` contiene appunto i framework *Foundation, Application e Cocoa*: possiamo intenderli come librerie necessarie alla realizzazione di una applicazione Cocoa. Sono lì contenuti tutti gli oggetti che a vario titolo partecipano ad una applicazione sviluppata in Cocoa. La cartella `Products` contiene nulla al momento (il file indicato in rosso in realtà non esiste ancora), ma alla fine conterrà il risultato del lavoro (l'applicazione vera e propria, insomma).

Ho lasciato per ultima la cartella `Resources` sulla quale appunto l'attenzione. Lascio per il momento da parte il file `InfoPlist.string`, e noto il file `MainMenu.nib`. Facendo un doppio clic su di esso, si apre Interface Builder.



Si apre un'insieme di finestre. Ce ne sono due per la costruzione dell'interfaccia, e due di ausilio per tale operazione.





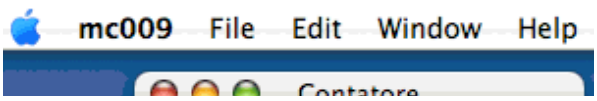
La prima finestra, larga e stretta, denominata `MainMenu.nib - MainMenu`, è un po' speciale perché contiene i menu che fanno parte dell'applicazione. Con un po' di pazienza e un uso dosato dei clic del mouse, è possibile costruire il menu, voce per voce, completo di scorciatoie da tastiera. Allo scopo, è molto utile aprire una ulteriore finestra, dal menu Tools, la voce `Show Inspector`. In questa finestra, dal contenuto variabile in dipendenza dall'elemento selezionato, si trovano, in diversi pannelli, le opzioni di configurazione dell'elemento stesso (nel caso, la voce del menu).

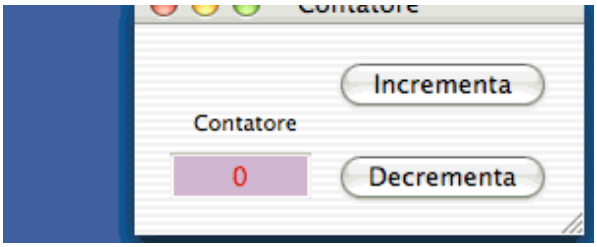
La seconda finestra è proprio una finestra, ovvero il punto di partenza di costruzione dell'applicazione. Questa finestra è ancora vuota, ma si può utilizzare la *palette* delle `Cocoa-view` per aggiungere elementi dell'interfaccia.

Questa palette, divisa in diversi pannelli, contiene infatti, raggruppati per tipo, i vari elementi che possono entrare a fare parte dell'interfaccia di una finestra: bottoni, campi testo, icone, eccetera. Non mancano elementi quali finestre ed interi menu da aggiungere ai menu standard. Con la semplice tecnica del drag and drop si possono trasferire questi elementi dalla palette alla finestra (o alla diverse finestre) dell'applicazione.

L'INTERFACCIA DI CONTATORE

La mia prima applicazione è piuttosto semplice, anche per l'interfaccia. Si diceva che si voleva un campo testo dove rappresentare il valore corrente del contatore e due pulsanti, uno per incrementare il valore e l'altro per decrementarlo. Una scritta aggiuntiva e decorativa completa l'interfaccia.





La finestra da me disegnata è piuttosto orribile a vedersi, e non dovrebbe essere difficile fare qualcosa di più coreografico e piacevole alla vista. Ci sono alcune cose da notare:

- muovendo gli elementi qui e lì all'interno della finestra, ogni tanto compaiono delle linee blu di guida per il posizionamento; il posizionamento stesso poi è *costretto* all'interno di una griglia nascosta. Ora, Apple ha reso disponibile le linee guida di progettazione dell'interfaccia, un tomo corposo dove sono elencate tutte le regole da seguire per avere una interfaccia in puro stile Aqua (c'è un intero tomo di riferimento nella documentazione: *Apple Human Interface Guidelines*). Ebbene, IB aiuta lo sviluppatore a seguire queste regole proprio con queste linee blu. Se posizionate gli elementi quando le linee blu sono visibili, questi elementi si trovano automaticamente nella posizione consigliata da Apple. Ad esempio, un pulsante dovrebbe distare dal bordo della finestra di un certo numero di pixel; un secondo pulsante deve distare dal primo un certo numero di pixel, eccetera. Inoltre, le linee blu aiutano anche ad allineare tra loro pulsanti, testi, e tutti gli elementi di interfaccia. Si è ovviamente liberi di mettere gli elementi grafici dove più aggrada, ma dal momento che seguire le linee guida non costa alcuna fatica e permette all'applicazione di aderire ai consigli di Apple, tanto vale posizionare gli elementi come consigliato.
- c'è sempre la possibilità di verificare il funzionamento dell'interfaccia, l'aspetto dei menu e delle finestre, scegliendo la voce `Test interface` del menu `File`. Ovviamente, ciò che si può verificare è come si comportano i singoli elementi dell'interfaccia quando si operano su di essi, o quando si ridimensionano le finestre, cose del genere, e non il funzionamento vero e proprio (non aspettatevi insomma che selezionando `New` dal menu `File` si apra una nuova finestra...).

LA CLASSE CONTROLLANTE

Ad un certo punto della storia avevo introdotto il modello MVC (Model-View-Controller) come un utile paradigma per la realizzazione di applicazioni object oriented, particolarmente utile nei casi in cui si ha a che fare con una interfaccia grafica. È giunto il momento di applicare questo paradigma. Avendo costruito con IB l'interfaccia, ho già definito tutte le classi di tipo View interessate all'applicazione: sono i pulsanti, la finestra, eccetera.

Ho già oltretutto le classi della parte Model: in realtà è un'unica sola, è proprio la classe Contatore già realizzata (e che adesso riuso tale e quale!!!). Mi mancano le classi di tipo controllore che servono a legare il tutto. Alcune classi controllore non si vedono neppure: sono quelle che si occupano per noi delle funzioni nascoste dell'interfaccia (aprire i menu, passare le richieste alle finestre, eccetera). Mi interessa in realtà una unica classe controllore per mettere in comunicazione le viste (l'interfaccia) con il modello (il contatore). Devo cioè definire una classe che si occupi di mediare le richieste dell'utente

(clic sul pulsante Incrementa o sul pulsante Diminuisci) e le funzionalità svolte da Contatore. Posso fare questa cosa direttamente all'interno di IB.

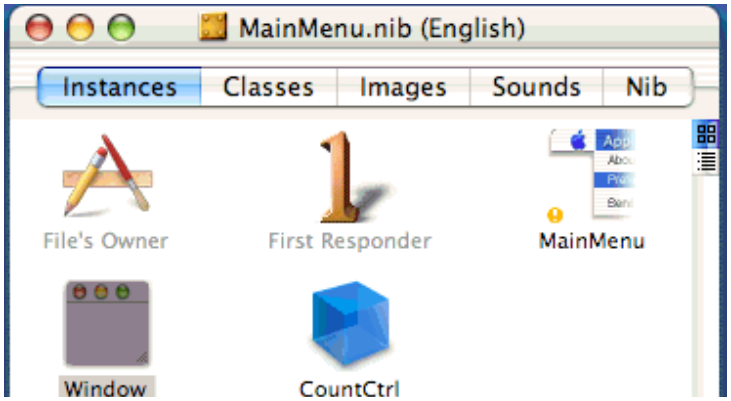
Seleziono il pannello `Classes` dalla finestra `MainMenu.nib`. Compare un elenco di tutte le classi disponibili (rappresentate in forma gerarchica, in una lista normale o nelle colonne). Mi posiziono su di una queste classi, che sarà la superclasse della nostra nuova classe. Poi, con menu contestuale o dal menu `Classes`, dico di volere una sottoclasse. Ecco che si costruisce una nuova classe nell'elenco, alla quale posso dare il nome che voglio.

Ho costruito una classe dal nome `CountCtrl` figlia diretta di `NSObject`. L'idea della classe controllore `CountCtrl` è quella di fungere da centro di smistamento. Gli elementi dell'interfaccia grafica mandano i loro messaggi a `CountCtrl`. Questa classe interagisce con `Contatore` per effettuare le operazioni richieste; il risultato (il valore del contatore) è restituito ancora una volta a `CountCtrl` che lo trasferisce nuovamente all'interfaccia.

Perché `Contatore` non è collegato direttamente all'interfaccia (come programmaticamente farebbe ogni programmatore in programmazione procedurale)? Perché seguendo la filosofia della OOP, `Contatore` non deve sapere, non è proprio interessato, anzi, è meglio che non sappia, cosa viene fatto del suo valore. È bene che questo valore sia inviato alla classe controllore, che si occuperà di tutta la sporca faccenda. In questo modo la classe `Contatore` potrà essere utilizzata con successo molte altre volte, non essendo legata al particolare uso che ne viene fatto. Invece, la classe controllore è molto legata all'applicazione all'interno della quale è nascosta, ed è quindi di difficile (per non dire impossibile) riusabilità. Ma questo è poco importante, visto che la classe controllore si occupa in genere semplicemente di dirigere il traffico, di mettere in bella copia ed evitare pasticci. Cose in genere semplici che si fan presto a fare. In genere, sono le classi modello che contengono tutta la complessità della faccenda.

Detto questo, e creata la classe `CountCtrl`, vediamo di farle fare qualcosa. In primo luogo, faccio in modo che i pulsanti interagiscano con il controllore. Ci aiuta il paradigma target/action sopra discusso. Il pulsante `Incrementa` dovrà in qualche modo individuare la classe `CountCtrl` come target di una azione di incremento del contatore, ovvero, bisogna fare in modo che, una volta cliccato, il pulsante invia un messaggio 'incrementa' all'oggetto `CountCtrl`.

Un momento, non ce l'ho un oggetto destinatario (ho solo dichiarato la classe!). Bene, vado nella vista classi, seleziono la classe `CountCtrl`, e attraverso menu contestuale o voce relativa nel menu `Classes` dico `Instanziate`....

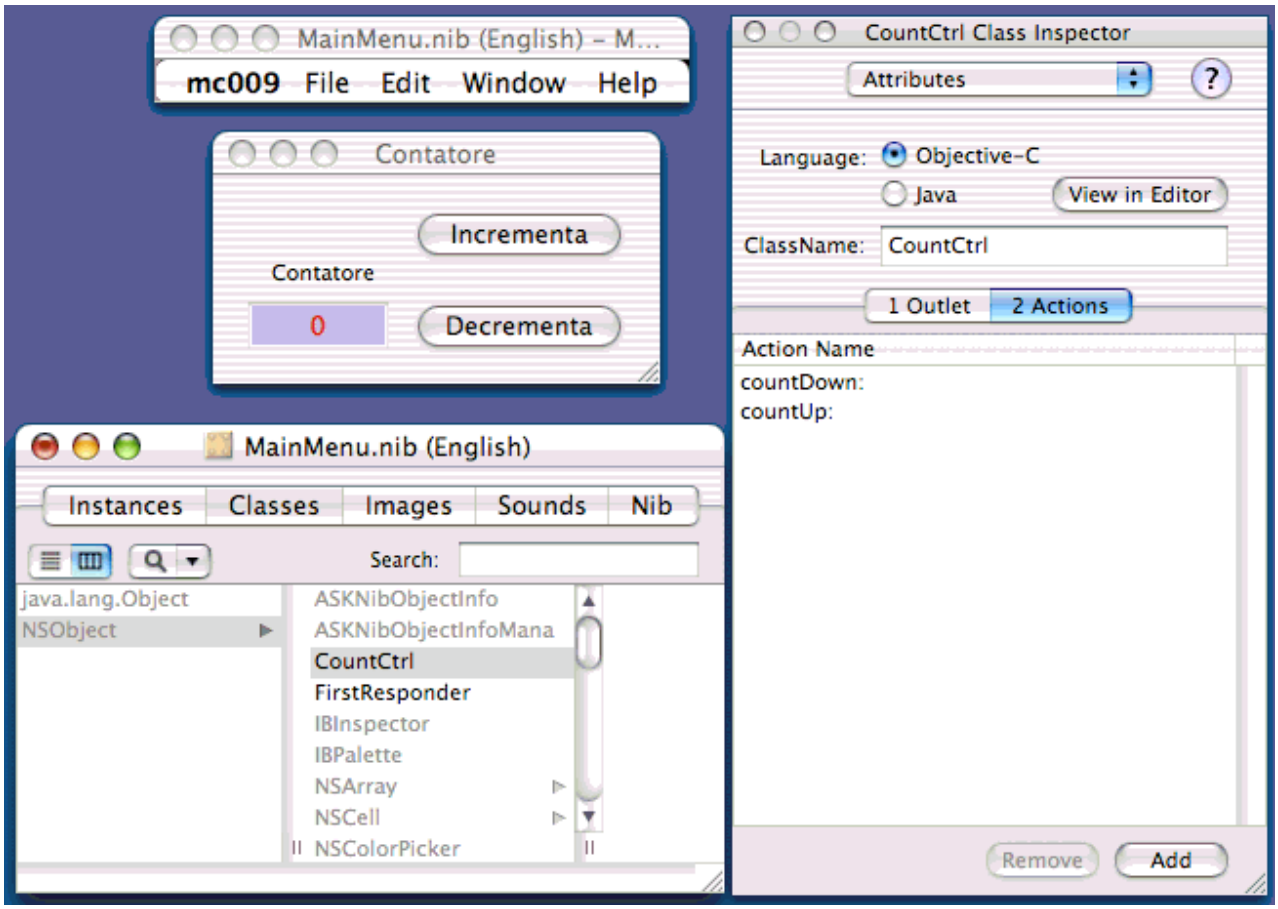


La finestra passa subito alla vista `Instances` dove mostra l'icona di un cubetto dal nome `CountCtrl`. Questa è l'istanza della classe (classe ed oggetto hanno lo stesso nome... non è una bella cosa, potrebbe ingenerare confusione, ma poiché c'è una sola istanza della classe, lo sopporto). Ma io torno indietro alla vista per Classi, perché devo ancora perfezionare la dichiarazione della classe `CountCtrl`.

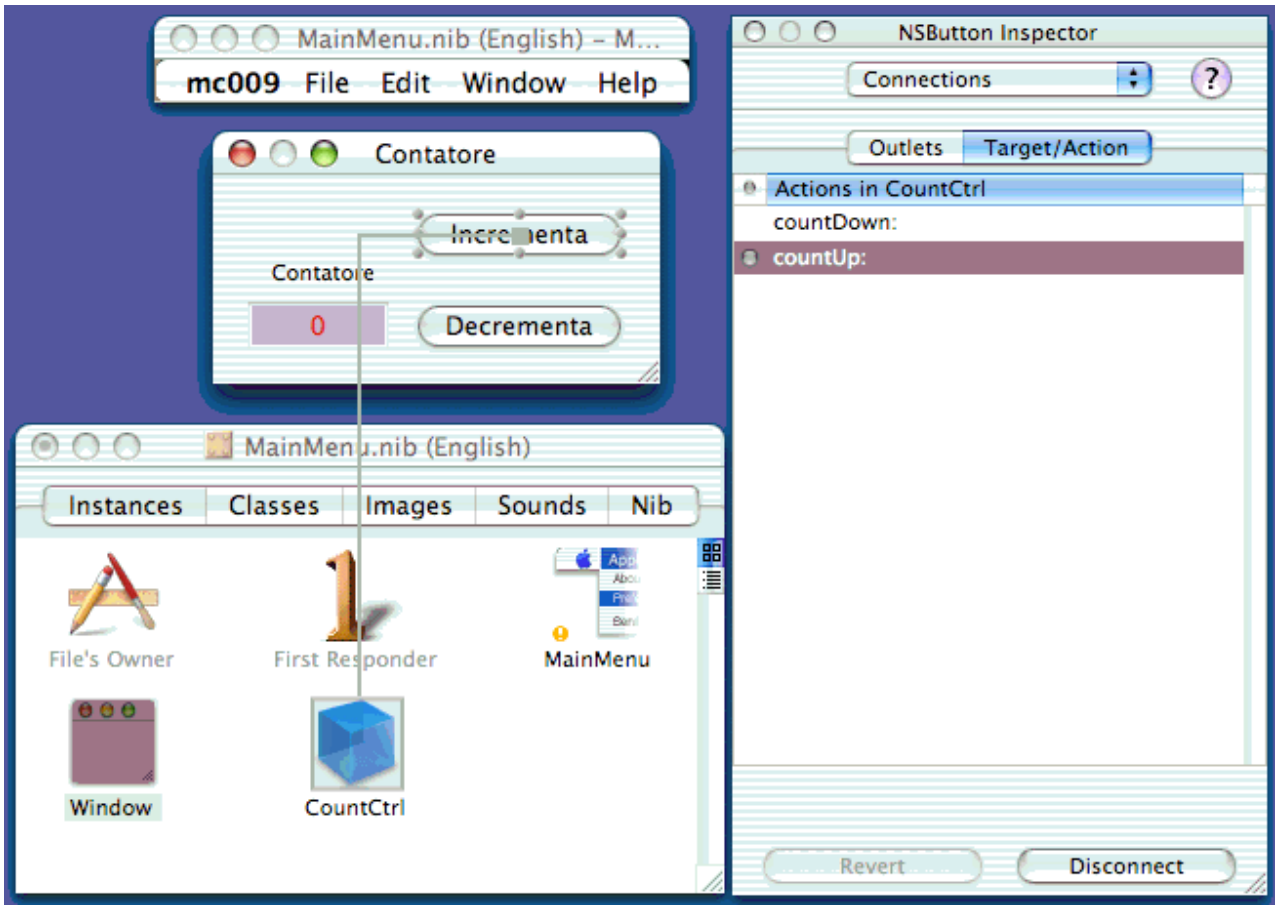
In primo luogo, questa classe deve essere punto di riferimento (target) di due azioni, una per incrementare il contatore `countUp` e una per decrementarlo `countDown`. In secondo luogo, deve stabilire un legame (`cntDisp`) con l'oggetto dell'interfaccia in grado di visualizzare il valore del contatore, cioè il campo di testo.

Detta in altre parole: occorre dichiarare due metodi `countUp:` e `countDown:` ed una variabile d'istanza `cntDisp`. IB chiama queste due cose rispettivamente una Action e un Outlet.

Parentesi terminologica. Non è che Action sia un altro nome per Metodo e Outlet un altro nome per variabile d'istanza. Una Action è un metodo utilizzato per comunicazioni tra oggetti facenti parte dell'interfaccia, perché tipicamente richiesta dall'utente tramite una azione volontaria (un clic). Un Outlet è una variabile d'istanza destinata a contenere un (puntatore ad un) oggetto, utilizzato ancora una volta per la realizzazione di una interfaccia. Fine parentesi.



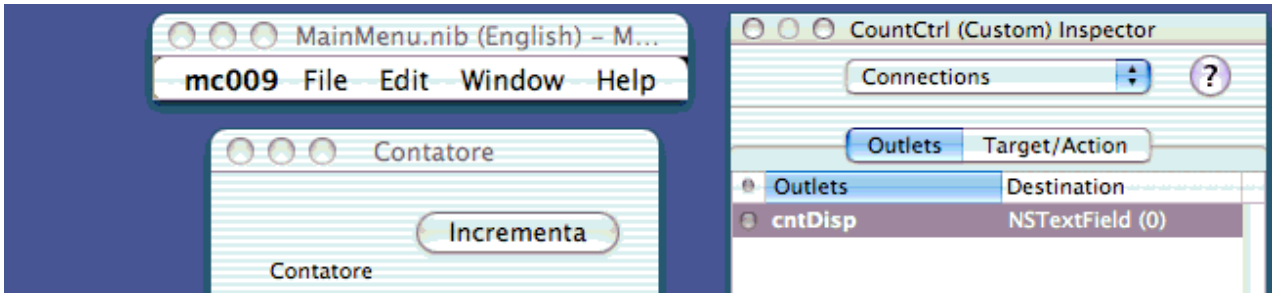
Esistono diversi metodi per definire Action e Outlet, però poi tutti portano allo stesso posto, nella palette dello `Inspector`, sul pannello `Attributes`. Con i pulsanti appositi, aggiungo le due action e il singolo outlet.

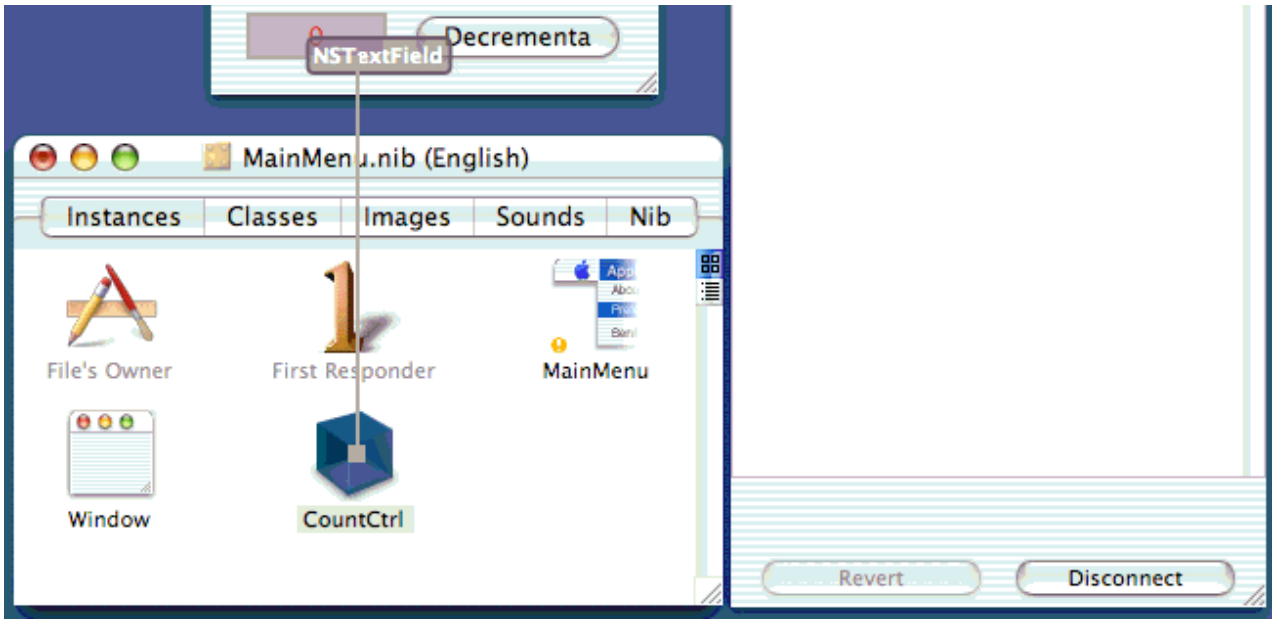


Infine, faccio i collegamenti tra le classi View e la classe Controller. Attenzione al meccanismo: seleziono il pulsante `Incrementa`. Tengo premuto il tasto '`control`'. Trascino il mouse sull'istanza `CountCtrl` (dentro la finestra principale). IB disegna una linea grigia che collega i due oggetti, e contestualmente la finestra `Inspector` diventa quella delle '`Connections`' dell'oggetto pulsante. La voce `Target`è evidenziata, e sono disponibili le due scelte, `countUp` e `countDown`, che avevo immesso in precedenza per l'oggetto `CountCtrl`. Seleziono `countUp` e faccio clic sul pulsante `Connect`.

Tutto questo lavorio (che ci si mette diecimila volte più tempo a scrivere che a fare) è semplicemente un metodo grafico per dire che l'oggetto pulsante `Incrementa`, quando cliccato, manda un messaggio `countUp:` all'oggetto `CountCtrl`.

Faccio la stessa cosa (ma con l'action `countDown`) per il pulsante `Decrementa`.





Poi passo all'outlet, ovvero al collegamento tra il campo testo dove si trova il contatore e la classe controllore. Adesso faccio prima clic sull'istanza di `CountCtrl` e poi, tenendo premuto il tasto '`control`', mi sposto sul campo testo. Questa volta la finestra `Inspector` si apre sulle `Connections` di `CountCtrl` e mostra disponibile la sola voce outlet `cntDisp`. Faccio clic sul pulsante `connect` ed ecco che ho assegnato un valore alla variabile d'istanza. Questo valore è proprio l'oggetto campo testo che mi interessa manipolare.

A questo punto l'interfaccia della nostra applicazione è finita. Bisogna tornare in Xcode per fare i collegamenti tra la classe controllore e la classe modello.

Ma prima, il tocco finale di magia di IB. Seleziono dalla vista per classi della finestra `MainMenu.nib` la classe `CountCtrl`. Con menu contestuale o equivalente, scelgo `Create files for...`. IB costruisce automaticamente due file, `CountCtrl.h` e `CountCtrl.m`, in cui scrivere i dettagli finali della classe, già completi delle informazioni già note (i due metodi `countUp:` e `countDown:` e la variabile outlet).

Adesso si tratta di riempire gli spazi vuoti.

In primo luogo, aggiungo i file della classe Contatore al nostro progetto. Uso il menu `Project` e la voce `Add To Project...`. È bene avere selezionato la cartella `Sources` in modo che vengano piazzati nel posto più logico. C'è comunque la possibilità di spostarli in un momento successivo. I due file della classe `CountCtrl` dovrebbero essere già al posto giusto. Ma adesso abbiamo due problemi.

IL FABBRICANTE DI OGGETTI

Il primo problema è posto dalla domanda: chi costruisce gli oggetti? Per quanto riguarda tutti gli oggetti presenti nel file nib, cioè gli oggetti dell'interfaccia, chi costruisce gli oggetti dovrebbe essere l'applicazione stessa. Mi aspetto che la struttura di programmazione alla base di Cocoa si occupi di costruire i vari oggetti (pulsanti, campi di testo, eccetera) che ho preventivamente definito all'interno di IB. Poiché `CountCtrl`

è una classe definita all'interno di IB, l'oggetto di tale classe presente nel file nib è costruito al lancio dell'applicazione. Brilla per la sua assenza l'oggetto `Contatore` che ci occorre per fare i calcoli.

Non ho al momento idea di quale sia il metodo migliore o quello consigliato da Cocoa per costruire l'oggetto `Contatore`. Ciò che mi è venuto in mente non mi piace particolarmente, ed è suscettibile di revisioni. Tuttavia, il metodo che ora illustro ha il pregio di introdurre un nuovo ed importante concetto di Objective C.

Uno dei metodi di `NSObject`, ovviamente ereditato da tutte le classi, è il metodo `init`. Con questo metodo si realizzano tutte le inizializzazioni interne dell'oggetto; ad esempio, è il posto migliore dove mettere i valori di default delle variabili d'istanza. Ricordo anche che, per costruire un oggetto, prima bisogna inviare un messaggio `alloc` alla classe (che predispone lo spazio in memoria per conservare l'oggetto), e poi appunto il messaggio `init` (che riempie lo spazio precedentemente predisposto con valori sensati). E fin qui, nulla di speciale. Adesso però consideriamo la definizione di una nuova classe. Questa differisce dalla sua superclasse per qualche particolarità, tipicamente un metodo o una nuova variabile d'istanza. Se c'è una nuova variabile d'istanza, è bene inizializzare questa variabile. Non si può usare il metodo `init` ereditato, perché non contiene ovviamente le nuove operazioni. Non ha senso scrivere un nuovo metodo `init` totalmente nuovo (fare cioè l'overloading del metodo `init`); in certi casi non è neppure possibile (ad esempio, non conosciamo la struttura interna dell'oggetto `NSObject`, e quindi qualsiasi init da me definito sarà sbagliato...). Esiste però la possibilità di fare l'overload del metodo `init` senza cancellare il vecchio `init`, o meglio, riutilizzare il vecchio `init` per le cose standard. Il trucco è molto semplice: scrivo il mio metodo di `init`. Però, la prima cosa che faccio all'interno di questo metodo è di eseguire l'`init` della superclasse. In altre parole, il metodo `init` della classe figlia manda il messaggio `init` alla superclasse. Per fare questo, si utilizza una variabile d'istanza presente in ogni oggetto, chiamata **super**. `Super`, in ogni oggetto è un puntatore alla superclasse, in modo da potersi riferire ad essa quando necessario. C'è un problema. Il metodo `init` è così dichiarato:

```
- (id)    init
```

Il metodo non richiede alcun parametro di messaggio, ma restituisce un oggetto (un puntatore a...). Che oggetto? Ma quello che la classe ha appena inizializzato, proprio l'oggetto che sto costruendo. In effetti, ricordo la classica definizione di una istanza d'oggetto (mi cito):

```
Contatore    * miocont ;
miocont = [[Contatore alloc] init ];
```

Definito il puntatore con la prima istruzione, a questo puntatore è assegnato il valore di ritorno del messaggio `init`. Tutto questo lungo discorso per giustificare la seguente struttura del codice del metodo `init`:

```
- (id)    init
{
    self = [super init] ;
    // le nuove inizializzazioni
```

```
    return ( self );
}
```

Compare una nuova variabile, `self`, anch'essa predefinita d'istanza. Rappresenta l'oggetto corrente. Per capire come funzionano le cose, si deve partire dal fatto che qualcuno ha già invocato il metodo `alloc` sulla classe. Questo metodo ritorna un puntatore ad oggetto, cioè, un puntatore ad una zona di memoria ancora vuota destinata a contenere l'oggetto stesso. All'oggetto adesso è inviato il messaggio di `init`. La prima istruzione invia un messaggio `init` alla superclasse. Così facendo, sono eseguite le operazioni di inizializzazione, relativamente alla parte spettante alla superclasse, della zona di memoria dell'oggetto. Il metodo restituisce ancora il puntatore all'oggetto, che non dovrebbe essere cambiato. Questo è il puntatore all'oggetto, che quindi assegno a `self`. Dopo, non devo far altro che inizializzare gli elementi nuovi di pertinenza del nuovo oggetto, e restituire come risultato il puntatore all'oggetto stesso (come appunto aveva fatto il metodo `init` della superclasse).

Questa tecnica, di invocare il metodo della superclasse utilizzando la variabile `super`, non è limitata al solo metodo `init` (anche se qui è usata molto spesso), ma è estesa a tutti i metodi in cui è possibile riutilizzare in gran parte il lavoro già svolto. Addirittura, se la nuova classe si differenzia dalla superclasse solo per i valori iniziali delle variabili d'istanza, è certamente più chiaro e pulito chiamare `init` della superclasse per eseguire le inizializzazioni di default, e poi riscrivere le sole nuove inizializzazioni, anche se in qualche caso ciò significa fare le cose due volte (buttando via quelle fatte la prima volta).

IL CREATORE DEL CONTATORE

Sono a questo punto in grado di dire chi costruisce l'oggetto `Contatore`.

All'interno della classe `CountCtrl` aggiungo una variabile d'istanza, che è proprio l'oggetto `Contatore`. Dentro il metodo `init` di `CountCtrl` faccio le inizializzazioni standard, e poi passo a costruire ed inizializzare proprio l'oggetto contatore. Ci metto di più a spiegarlo che a farlo: Il file `CountCtrl.h` è allora il seguente:

```
#import <Cocoa/Cocoa.h>
#import "Contatore.h"

@interface CountCtrl : NSObject
{
    IBOutlet id cntDisp;
    Contatore    * cnt ;
}
- (id)          init ;
- (IBAction)    countDown: (id)sender ;
- (IBAction)    countUp: (id)sender ;
@end
```

Poi, all'interno di `CountCtrl.m` la definizione del metodo init è questa:

```
- (id) init
{
    self = [super init] ;
    cnt = [ [Contatore alloc] init ];
```

```
    [ cntDisp setIntValue: 0];
    return ( self );
}
```

Ne approfitto anche per dare un valore iniziale a `Contatore`, nullo. A dire la verità, avrei potuto farne a meno, ora che ho imparato il meccanismo di `init`. Infatti, posso aggiungere alla classe `Contatore` il metodo `init`, in modo che, alla creazione, il contatore sia inizializzato a zero:

```
@implementation Contatore
- (id) init
{
    self = [super init] ;
    counter = 0 ;
    return ( self );
}
...
```

LA SCRITTURA DEL VALORE

Diventa questo punto molto semplice per `CountCtrl` passare la palla all'oggetto `Contatore` quando riceve un messaggio da uno dei due pulsanti dell'interfaccia. Pigliamo il metodo per incrementare il valore del contatore:

```
- (IBAction)countUp:(id)sender
{
    short nuovo Val ;
    [ cnt countUp];
    nuovoVal = [ cnt getValue ] ;
    // ed adesso?
}
```

Mi sono imbattuto nel secondo problema: come fare a dire al campo di tipo testo (che è puntato dall'outlet `cntDisp`) di mostrare il nuovo valore del contatore. Mi dico: basta mandargli un messaggio apposito, passandogli come argomento appunto cosa si vuole mostrare nel campo testo. Quindi, vado a vedere cosa dice la documentazione. Il campo testo è un oggetto della classe `NSTextField`, vado e trovo metodi per: controllare l'editabilità e la selezione del testo; per il colore; l'aspetto; il background; il bordo del campo; per agganciare tra loro campi; per selezionare il testo; per impostare il delegato (che diavolo è?)... niente per leggere o scrivere il contenuto.

Ci ho messo circa un paio d'ore di navigazione in mezzo all'aiuto prima di ottenere la risposta più ovvia e razionale. Perbacco, ma `NSTextField` è una sottoclasse di `NSControl`. Se non ci sono metodi propri ed esclusivi di `NSTextField`, è perché sono già stati definiti dalla sua superclasse! (ovvio, vero? dopo tutto quello che ho detto sugli oggetti...).

Infatti, tra le altre miriadi di metodi di `NSControl` ce ne è un gruppo assolutamente fantastico per i nostri scopi, che permette di impostare i valori del campo senza bisogno di formattare in precedenza il valore. Copio brutalmente dalla documentazione:

```
setIntValue:
```

```
- (void)setIntValue:(int)anInt
Sets the value of the receiver's cell (or selected cell) to the integer
anInt.
```

Assolutamente fantastico. Questo mi permette di scrivere il metodo in due sole righe:

```
- (IBAction)countUp:(id)sender
{
    [ cnt countUp];
    [ cntDisp setIntValue: [ cnt getValue ] ];
}
```

Con la prima riga, predispongo il contatore; con la seconda, prima ricavo il valore, che uso come parametro del messaggio setIntvalue: inviato al campo di testo.

Ovviamente, il metodo gemello è presto scritto:

```
- (IBAction)countDown:(id)sender
{
    [ cnt countDown];
    [ cntDisp setIntValue: [ cnt getValue ] ];
}
```

Volendo, avrei potuto fare tutto in una sola riga. Basta riscrivere il metodo `countUp:` (e `countDown:`) del contatore perché restituisca il nuovo valore del contatore. Non sarebbe quindi più necessario invocare il metodo `getValue:` sul contatore...

Anzi, mi pare una talmente buona idea che lo faccio subito.

Riporto tutti e quattro i file interessati.

Cominciamo con `Contatore.h`

```
#import <Foundation/Foundation.h>

@interface Contatore : NSObject {
    short    counter ;
}
- (id)        init ;
- (short)    setValue: (short) startVal ;
- (short)    getValue ;
- (short)    countUp ;
- (short)    countDown ;
@end
```

Poi `Contatore.m`

```
#import "Contatore.h"

@implementation Contatore
- (id)    init
{
    self = [super init] ;
    counter = 0 ;
    return ( self );
}
```

```
-(short) setValue: (short) valore
{
    counter = valore ;
    return ( counter );
}

-(short) countUp
{
    return(++counter) ;
}

-(short) countDown ;
{
    return(--counter) ;
}

- (short) getValue
{
    return ( counter );
}
@end
```

Di seguito, `CountCtrl.h`:

```
@interface CountCtrl : NSObject
{
    IBOutlet id cntDisp;
    Contatore    * cnt ;
}
- (id)        init ;
- (IBAction)    countDown: (id)sender;
- (IBAction)    countUp: (id)sender;
@end
```

e per finire `CountCtrl.m`:

```
#import "CountCtrl.h"

@implementation CountCtrl
- (id)    init
{
    self = [super init] ;
    cnt = [ [Contatore alloc] init ];
    return ( self );
}

- (IBAction)countDown:(id)sender
{
    [ cntDisp setIntValue: [ cnt countDown] ];
}

- (IBAction)countUp:(id)sender
{
    [ cntDisp setIntValue: [ cnt countUp] ];
}
@end
```

Ora si compila l'applicazione, la si esegue, e si è contenti perché funziona. Ci sono ancora migliaia di particolari da mettere a punto (il testo si può editare? non dovrebbe... Che succede se ridimensiono la finestra? Oh che disastro... cose del genere), altre cose

già disponibili *gratuitamente* (provate a stampare, a fare *About*...), ma il grosso pare fatto...

Introduzione
Chiamatemi Livio

Il mio nome è Sandel, Livio Sandel.

Ho cominciato ad usare un Apple nel 1987 con un Macintosh Plus perchè mi serviva un computer per elaborare, scrivere e stampare la tesi di laurea. In effetti sono ingegnere elettronico... Va be', dicevo che per la tesi ed altre storie ho cominciato a programmare il Macintosh per puro diletto, scrivendo in C e Pascal e leggendo i gloriosi Inside Macintosh I-III, producendo qualche programmino qui e lì...

Per questioni di praticità, sono poi passato a HyperCard, dove ho prodotto altre piccole sciocchezze. Nel 1990 ho cominciato a collaborare con M-Macintosh Magazine, per la quale ho scritto diversi articoli di vario genere, tra cui una serie sulla programmazione in HyperCard.

Dal 1994, per motivi legati al mio lavoro, ho lasciato un po' cadere la programmazione col Macintosh (per dedicarmi a programmare in altri campi, ambiente embedded e tempo reale), per arrivare a fare adesso anche altre cose ancora.

Con l'uscita di Mac Os X, mi è colta nuovamente vaghezza di ricominciare a programmare, visto che nulla c'è di meglio di un ambiente vergine come Cocoa non appesantito dalla storia precedente e dalla compatibilità all'indietro...
Naturalmente, un manoscritto...

Stiamo lentamente arrivando al motivo di questo sito: sto dunque cominciando a studiare il sistema operativo, l'ambiente di sviluppo, Cocoa e tutto il resto. Una abitudine che ho preso è di pensare e studiare scrivendo, producendo appunti a ruota libera.

Con un poco di sforzo, mi son detto, trasformo gli appunti personali in ragionamenti scritti che magari possono servire a qualcun altro; in altre parole, faccio un sito sulla programmazione in Cocoa.

Ci saranno tanti siti, e sicuramente migliori, che insegnano la programmazione in Cocoa, ma questo almeno è in italiano.

Comincio a scrivere senza bene sapere come andrà a finire. In effetti ignoro quali siano i problemi e le questioni da affrontare, né se possiedo tutti gli strumenti necessari. In pratica, questo è un diario di viaggio, scritto in tempo reale durante il viaggio. Il viaggio ha una meta, ma, come molti filosofi concorderanno, non è il raggiungimento della meta il primo obiettivo del viaggio. La ricompensa è il viaggio in sè stesso (troverete parecchie di queste perle di saggezza se avrete la

pazienza di seguirmi).

Non ho neppure la presunzione di fare il viaggio da solo. Io, adesso, parto. Se per strada qualcuno vorrà seguirmi, precedermi, indicarmi strade, portarmi sulle spalle o farsi portare da me, non c'è problema, anche a costo di cambiare meta e viaggio in corso. Come dice il sommo poeta, lo scopriremo solo vivendo. Ciò che dovrete sopportare è la mia scrittura, piena di divagazioni,citazioni più o meno nascoste, prolissa certamente e qualche volta noiosa. Pazienza. Se avessi più tempo, scriverei di meno.

Ciò che ho è una buona conoscenza del C, esperienza di programmazione (anche in ambiente Unix!), buona volontà e un po' di inglese. In base alla mia esperienza, sono attrezzato abbastanza bene. Per strada si incontreranno problemi, ma io sono un pessimista positivo: si supereranno certamente, anche se magari saranno necessari sudore e lacrime.

Non ho alcuna pretesa di conoscere la materia di cui parlo in queste pagine. Del resto, questo è il diario di viaggio di un percorso di scoperta. Troverete quindi affermazioni false e tendenziose, per il semplice motivo che in quel momento mi sto sbagliando. Potrò correggermi e contraddirmi, caoticamente ritornare sopra punti già considerati chiusi alla luce di nuovi argomenti, un via vai di errori, certezze e smentite. Insomma, è la vita.

Effetti collaterali

Il primo e principale effetto collaterale di questo sito è una mailing list. Nasce per raccogliere discussioni e quant'altro collegate alla lista principale o a questo sito, e vivrà fino a che qualcuno è interessato a parteciparvi. Tutti possono scrivere. Nessuno modera, ma caccerò senza pietà integralisti, venditori di fumo e spacciatori di messaggi spudoratamente pubblicitari. Gli argomenti più interessanti potranno (ma anche no) finire su questo sito.

In lista, dirò quando questo sito ha subito un aggiornamento.

Indice

001 - Prima di partire

L'attrezzatura; Il mezzo di trasporto; La strada; La meta.

002 - Il Sistema Operativo

System Overview; Cipolle; Gli ambienti Operativi; L'ambiente Cocoa; Foundation framework; Application Framework; Altri Framework.

003 - Object Oriented Programming

Programmazione orientata agli oggetti; Diversi modi di programmare; Le basi della OOP; Intervallo storico.

004 - Object Oriented Programming

Il nome delle cose; Questione di metodo; Classi ed oggetti; Ereditarietà; Gerarchia delle classi; Binding; Polimorfismo; Gerarchia e reti di oggetti; Il paradigma MVC.

005 - Objective C

Objective C; Tipo d'oggetto; Messaggi; Classi; Classi e tipo; Oggetti Classe.

006 - Objective C

Definizione di una classe; L'interfaccia; Usare altre classi; I vantaggi dell'interfaccia; La realizzazione.

007 - L'ambiente di sviluppo

XCode; La Finestra di XCode; Scrivo il programma.

008 - Un programma in Objective C

Contatore; La funzione NSLog; Proviamo NSLog; L'applicazione.

009 - Finalmente Cocoa

Finalmente Cocoa; Interface Builder; File NIB; Target/Action; Una Applicazione Cocoa; L'interfaccia di contatore; La Classe controllante; Il fabbricante di oggetti; Il creatore del contatore; La scrittura del valore.

010 - Attività di contorno

Usare IB e Xcode; EuroConv; L'interfaccia; Dinamica degli oggetti; L'applicazione in quattro istruzioni; Un po' di codice; Svegliati!; L'ultimo metodo.

011 - Localizzazione icone ed altre storie

Stringhe in lingua; Lingue dell'applicazione; Supporto all'internazionalizzazione; Stringhe Localizzate; EuroConv europeo; Icone; About.

012 - Questioni di memoria

Genesi: Alloc ed Init; Apocalisse: release; La vita in un contatore; Assegnare una variabile; Riassunto finale.

013 - Una tabella di file

L'interfaccia; Pescare il nome di un file; Informazioni del file.

014 - L'inizio del Catalogo

L'interfaccia grafica; Ricorsione sulle directory; NSOutlineView; Il codice; Deleghe; Come si parte.

015 - Formattatori

Celle e Formattatori; Formattatori; Farsi un formattatore; Filtro sui file.

016 - Codifica ed Archiviazione

Archiviazione; Protocolli; NSCoding; Punto di partenza; L'interfaccia; Salvataggio e recupero; Modifiche e cambiamenti.

017 - Documenti

NSApplication; Architettura; NSDocumentController; NSDocument; NSWindowController; Comincio; Interfaccia; Proxy; Quattro metodi.

018 - Menu e Palette

Interfaccia; Collegamenti; Ricorsione; Palette.

019 - Sesso Droga e Drag'n'Drop

Protocolli; NSDraggingDestination; NSPasteboard; Cosa droppare; I Metodi.

020 - Ricominciamo!

Rileggendo i file; Fatti e formati nuovi; Dimensioni strane; Nuovo inizializzatore; Nuovi filtri.

021 - Preferirei di no

Le Preferenze; La finestra e la classe; Lettura e scrittura delle preferenze; Le azioni; L'uso delle Preferenze.

022 - Informazioni e notai

Finestra di informazioni; Delega all'applicazione; Il controllore della finestra delle info; Unificazione delle gestioni; Notifiche; Gestire la finestra Info.

023 - Preferenze rivisitate

Ancora Preferenze; Nuovo controllore delle Preferenze; Modificare l'aspetto di NSOutlineView; Ancora notifiche.

024 - A Proposito

A proposito; La finestra; Uovo di Pasqua; Mostrare la finestra; Batti il tuo tempo; Animazione.

025 - AIUTO!

Aiuto; HTML; Registrazione dello Help.

026 - Fritto Misto

Due Errori; Giga Giga Bum; L'Icona del file; La finestra di Info; Salvataggio File.

027 - Solo Volumi, per favore

Dimensioni Enormi; Solo Volumi; La Finestra Modale; Volumi ed Enumerator; Montaggi e Smontaggi; Aggiungere Volumi; Doppio Clic e Select; Nuovo Drop; La Barra del Barbiere; Aggiungi un File.

028 - Barra degli Strumenti

Toolbar; Il delegato alla toolbar; Metodi opzionali.

029 - Ordine, ordine

Ordine ordine; Ordinare array; Un ordine alternativo; L'ordine dell'ordine.

030 - Un ordine migliore

Due funzioni anzi una; Ordine in loco; (quasi) Come il Finder.

031 - Ricerca

Ricerche; La finestra; L'interfaccia della finestra; Cambio menu al volo; Lancio della ricerca; Chi cerca...; Mostrare i risultati; ...Trova.

032 - Barra a Dritta

Nuova documentazione; Volumi; Una nuova classe; Attese piu' misurate; Un'altra differenza e' possibile.

033 - Un pezzo di Carbon

Dimensione e Risorse; Risorse con Carbon; Conto i file.

034 - Prima della pioggia

XCode; Finestra modale; Un nuovo campo.

035 - Rifondazione

Obiettivo; Struttura Dati; Controllori per finestre.

036 - Altre finestre

Preferenze e gestione colonne; L'ordine degli elementi; Ultimi tocchi alla finestra; Una diversa finestra.

037 - Chiare e fresche sorgenti

La sorgente dati; La lista dei volumi; Due liste una sorgente; Pannello informazioni.

038 - Ho una vista

Quattro Classi Basilari; Coordinate; La Scroll View; Disegnare in una view; Perché?.

039 - Primi esperimenti con le viste

Una vista particolare; La CoverView; Contenuto della CoverView; Disegnare in una view; A testa in sù.

040 - Polimorfismo classico

Un classico modello OOP; Un modello per la grafica; Vettori, gruppi e template; Disegnare per elementi.

041 - Costruisco Elementi

Alcune Piccole Modifiche; Il Fabbricatore d'Elementi; Colpire il Giusto Target; Disegnare col fabbricatore; Un Errore Corretto; Ulteriori correzioni.

042 - Eppur si disegna

La palette degli strumenti; Finestra CoverPalWinCtl; Trattare il mouse; Crearsi da soli; L'elemento corrente; Manici; Strumento selezione; Piccolezze.

043 - Drag Zoom Zoom

Spostare elementi; Facili Zoom.

044 - Ridimensionamento e Selezione

Nuovi handle; Selezionare un handle; Selezioni multiple.

045 - Shift Me Up!

Selezioni multiple; Aggiunte multiple; Spostamenti di gruppo; Cerchi quadrati e linee diritte; Un errore ed una cattiva soluzione; Muovere oggetti con la tastiera.

046 - Sposta, Allinea, Raggruppa e Blocca

Tre cose; Blocco e sblocco; Spostamento di livello; Allineamento; Centraggio; Gruppi.

047 - Grigliate

Griglia; Disegno della griglia; Disegnare con la griglia; Il Dialogo della griglia.

048 - Cancella tutto, anzi no

Cancellare elementi; Annulla e Ripristina; Pile di infiniti ripensamenti; Undo nella pratica; Tante piccole cose.

049 - Scrivere in una view

Così fan tutti; La classe CCE_Text; La lista dei file.

050 - Il costruttore diventa ispettore

Morte di un costruttore; Aggiornamento delle informazioni; Cambio dei parametri; Le altre modifiche.

051 - Rotazioni e Rivoluzioni, parte 1

Trasformazioni affini; Matrici di classe; I nuovi elementi grafici; Le altre classi di elementi; Creazione e modifica.

052 - Rotazioni e Rivoluzioni, parte 2

Gli altri metodi di CCE_BasicForm; Rotazioni; Nuova gestione delle notifiche; Informazioni riviste.

053 - Modificare il testo

Doppia idea; Sequenza delle operazioni; Cominciare le modifiche; Disegno e fine delle modifiche.

054 - Le correzioni

Correzioni sui gruppi; Menu contestuali; Testo automatico; Lo stile dei paragrafi.

055 - Immagina...

Elementi Immagine; Impostazione e disegno; Ancora sui menu contestuali; Dimensioni originali; Trasparenza.

056 - Template

Unità di misura; Le dimensioni contano; Template; Dove sono i template; Altre modifiche.

057 - Visto, si stampi

Le classi della stampa; Formato di stampa; Stampa di una copertina; Stampa del catalogo; Pagine e pagine; Il rettangolo della pagina; Decorazioni; Conclusioni preliminari.

058 - Mela C Mela V

Proprietà XML; Copia e Taglia; Incolla; Intervallo: correzioni; Colonne multiple.

059 - Suonala ancora Sam

Il problema e la soluzione; Gli script Applecript; L'interfaccia di LastSongs; La classe NSAppleScript; Aggiornamento del menu; Operazioni da menu.

060 - Ricerche approfondite

Nuove funzionalità; Interfaccia utente; I criteri di ricerca; Visualizzare i criteri; Gestire i criteri; La nuova ricerca; Scegliere con criterio.

061 - Meno finestre per tutti

Finestre ausiliarie; Abilitazione automatica dei menu; Mantenere la dimensione delle colonne.

062 - Esportazione dati

I nomi delle preferenze; Esportare in un file di testo; Esportazione; Formattare il testo esportato.

063 - Multithread

Task e Thread; Il server di catalogazione; Lanciare un thread; Fermate quella catalogazione.

064 - Progettare un salvaschermo

Salvaschermo; La classe ScreenSaverView; L'inizializzazione; Il movimento; Configurazione del Salvaschermo; Debug di un salvaschermo.

065 - Cocoa e Unix

CURL; L'interfaccia; Wrapping e Pipe; CurlWrapper; Questioni di protocollo; Ritorno a curlCtrl.

066 - Che fortuna!

Fortune; Progetti e Target; Uso di fortune; YahFortune; Far fortuna; Ridimensionamento della finestra.

067 - Un pannello delle preferenze

Aggiornamento fortune; Le preferenze; La nuova YahFortune; Il Panello; Ultime avvertenze.

068 - Fortuna stravolta

Scrittura di tagline; Lancio ed arresto dell'applicazione; Mettiamoci un titolo; Diverso comportamento.

069 - Continua Fortuna

Integrazione con Apple Mail; Centraggio della finestra; Gestione dei file; Il modello dei fortune; Scanners!!!; Una collezione di file; Salvo la lista; Calcolo delle probabilità; Alla ricerca dell'applicazione perduta.

070 - Un servizio di fortuna

Servizi; Dichiarazione di Servizio; Preparazione di YahFortune; Codice di servizio.

071 - Plugin x Widget = Facile!

Il widget ed il plugin; Model; Controller; Il plugin; Il progetto XCode.

072 - Key-Value coding

Intro a KVC; Termini; Leggere e scrivere; La realizzazione; Alla prova; Operatori su vettori ed insiemi.

073 - Universal binary

Binari universali; Converto il catalogo; E tutti gli altri; Punti notevoli.

074 - Debugging

GDB e XCode; A cosa serve un debugger; Non mi rompete; Cambiare le variabili; Passo dopo passo; Una sessione di debug; è solo l'inizio.

075 - DownQueue

C'era una coda; I requisiti; L'interfaccia; Drop sull'immagine; Aggiungere un file; Elementi in coda; Toolbar ed altre azioni; Il drawer; Supporto all'applicazione; Chiusura dell'applicazione.

076 - Continua DownQueue

La libreria cURL; Gestione dei trasferimenti; Gestione delle operazioni; Tocchi finali (si fa per dire); Localizzazione.

Capitolo 010 - Attività di contorno

Approfondimenti con una nuova applicazione.

Sorgenti: Varie fonti

Primo inserimento: 21 Novembre 2001

Ritocchi per aggiornamento XCode 2.1: 8 Agosto 2005

Usare IB e XCode

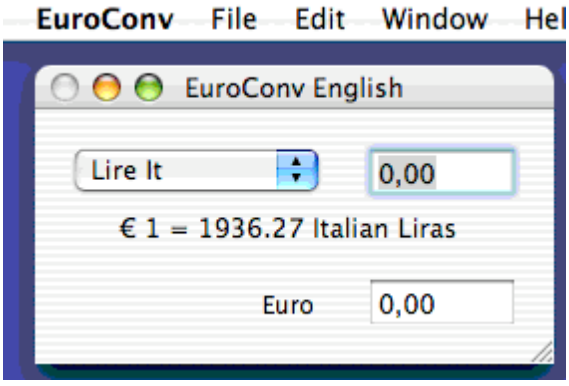
Dopo la realizzazione della prima applicazione in Cocoa, passiamo subito ad una seconda applicazione, rapida e veloce da realizzare, ma che ci permette di esaminare alcuni aspetti trascurati nella (frettolosa) realizzazione dell'applicazione Contatore.

Nota dell'Agosto 2005: all'epoca era un argomento d'attualità; adesso lo è un po' meno, nonostante ricorrenti dichiarazioni di uomini politici. Rimane comunque un utile esercizio.

Questa nuova applicazione permette di calcolare la conversione in Euro tra le valute che partecipano all'unificazione monetaria europea. La cosa è programmaticamente semplice, ma ci sono alcune cose che saranno approfondite oppure esaminate per la prima volta:

- usare meglio IB per la costruzione delle interfacce
- localizzare l'applicazione nelle varie lingue
- aggiungere la propria icona all'applicazione
- finestra di about

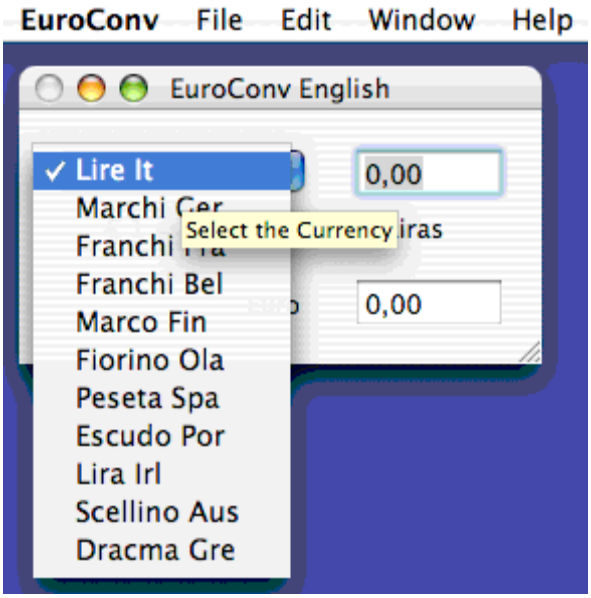
EuroConv



L'applicazione *EuroConv* presenta una interfaccia molto semplice: ci sono due campi di testo per contenere l'uno l'importo in Euro e l'altro l'importo in una delle valute dell'area Euro. Un menu poop-up permette di selezionare la valuta che si intende utilizzare (lire, marchi, franchi, eccetera), e contestualmente si modifica una stringa di testo che informa sul tasso di cambio in Euro di quella particolare valuta. Faccio prima a mostrare la figura che a descrivere oltre la cosa.

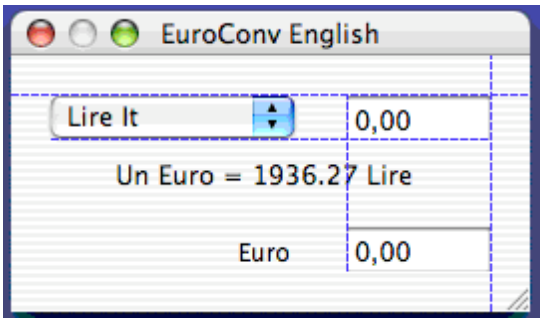
L'idea è che la semplice scrittura di un valore in uno dei due campi scateni la conversione (in un senso o nell'altro) dell'importo. D'altra parte, il cambio di valuta mantiene fisso il valore in Euro e converte il campo della valuta locale. In questo modo, se voglio sapere a quanti franchi francesi corrispondono mille lire, parto dalla valuta italiana, scrivo mille, automaticamente l'importo in euro è aggiornato, seleziono dal menu i franchi, ed ottengo il corrispondente in franchi del valore in Euro (che corrisponde appunto a quello di partenza in lire).

L'INTERFACCIA



Il progetto nasce velocemente: faccio un nuovo progetto in XCode, lo chiamo `m010`, passo subito in IB per manipolare l'interfaccia. Parto dalla finestra, che costruisco velocemente aggiungendo i due campi testo, la stringa e il menu pop-up. L'aggiunta di tutte le valute è la parte più lunga e noiosa. Per aggiungere voci al menu, ho impiegato qualche minuto per realizzare che si fa drag&drop a partire dall'elemento `item` della palette dei menu (ovvio, no?).

Già che ci siamo, modifico anche tutti i menu, inserendo il nome dell'applicazione `EuroConv` in tutti i posti in cui mi pare necessario (nell'`About`, nel `Quit`, nello `Help`). Bene. Adesso comincio a fare le cose che rendono una applicazione Macintosh di tale nome.



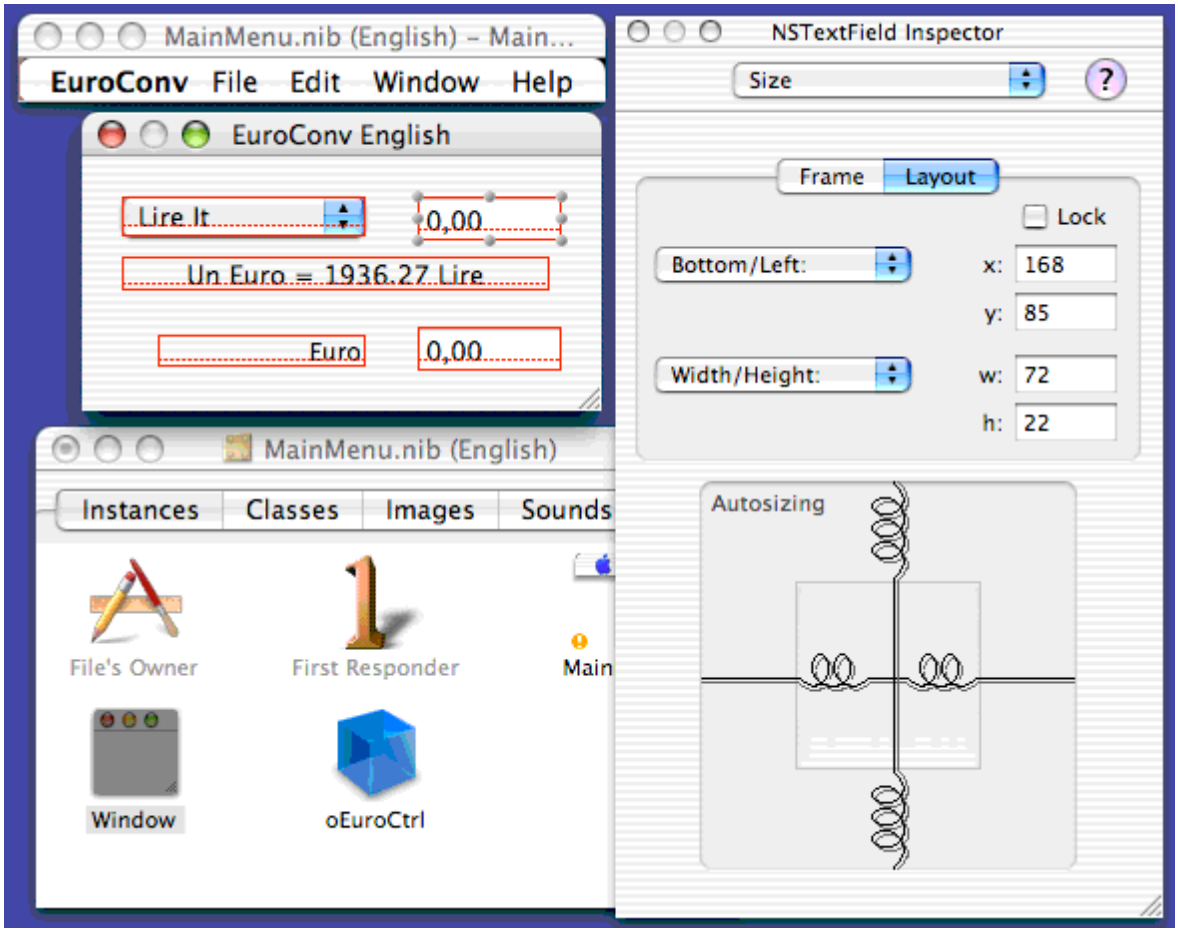
Comincio col disporre accortamente tutti i vari elementi dell'interfaccia. Allo scopo, mi faccio aiutare dalle linee guida, trattate nel capitolo precedente. Dispongo il campo

valuta locale in modo che le due linee guida (azzurre nel mio caso) verticali ed orizzontali scattino nei pressi dell'angolo in alto a sinistra della finestra. Dopo di che, allineo il menu pop-upin modo che, di altezza uguale, corrisponda orizzontalmente. Il campo della valuta in Euro lo piazzo sotto il campo testo della valuta locale, e poi aggiungo due testi non editabili per contenere informazioni (in uno ci ho scritto semplicemente `Euro` e non lo cambierò mai, mentre l'altro campo avrà un contenuto variabile con la voce attiva del menu pop-up... intanto ci metto un testo di default relativo alle lire). Per aiutare ulteriormente il posizionamento degli oggetti, oltre alle linee dinamiche è possibile disegnale delle linee guida aggiuntive (voce `Guide` dal menu `Layout` di IB).

DINAMICA DEGLI OGGETTI

Il passo successivo è decidere come si comportano gli oggetti quando cambia la dimensione della finestra.

Per i ridimensionamenti orizzontali, decido che il menu e la scritta `Euro` rimangono di dimensione fissa ed attaccati al bordo sinistro della finestra, mentre i due campi delle valute si allungano o restringono mantenendo costanti le distanze dai bordi della finestra. Per i dimensionamenti verticali, decido in pratica di riposizionare uniformemente lo tre *righe* nello spazio disponibile, senza modificare le altezze degli oggetti. Nel terzo superiore rimane quindi il menu e il campo valuta locale, nel terzo centrale le informazioni sulla conversione, nel terzo inferiore la scritta `Euro` e la valuta corrispondente.

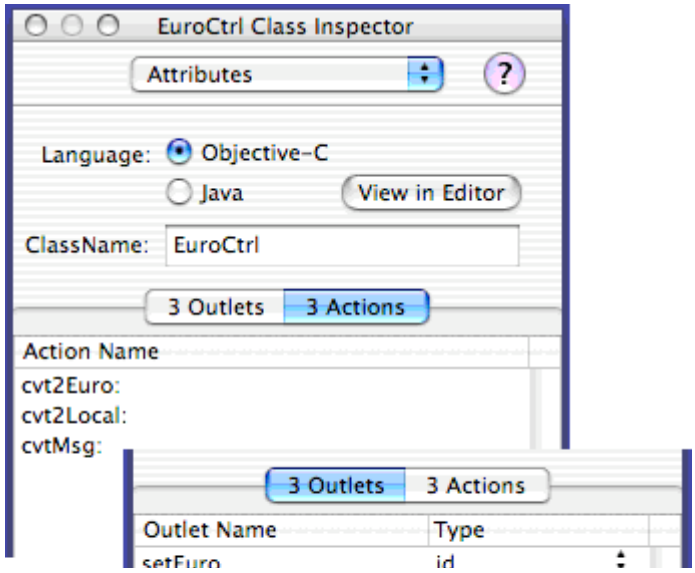


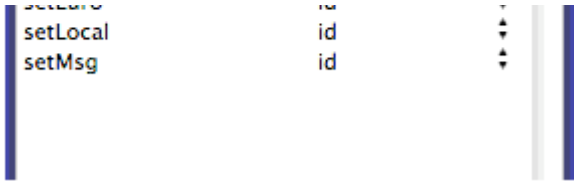
Per fare ciò, ho a disposizione il pannello `Size` della finestra di informazioni (che si ottiene facendo `Show Inspector` dal menu `Tools` di IB). In questo pannello, oltre a decidere numericamente le dimensioni, ho un disegno (il riquadro `Autosizing`) che determina il comportamento dell'oggetto selezionato nei confronti dell'oggetto visuale che lo contiene (nel caso, la finestra). Il comportamento si stabilisce impostando i legami del quadrato interno con il quadrato esterno. Tali legami possono essere di due tipi: fissi, corrispondenti a una doppia barra dritta, o dinamici, corrispondenti ad una connessione a *molla*. È così facile dire come si comportano i vari oggetti. Ad esempio, i campi valuta devono mantenere fisse le distanze dai bordi orizzontali e la loro dimensione verticale, cioè l'altezza. Per fare questo, vi rimando alla figura, cosa che mi risparmia un sacco di parole (ma spiego comunque: occorre mettere molle all'interno in direzione orizzontale, lasciando fisse le barre orizzontali. Viceversa, verticalmente, esternamente si hanno molle ed internamente barre dritte).

È possibile verificare subito cosa succede quando ridimensiono la finestra. Da IB allora scelgo `Test Interface`, e simulo il comportamento del programma. Mi limito a ridimensionare la finestra, e vedo che i vari elementi dell'interfaccia si comportano come desiderato. Più o meno: ci sono il menu pop-up ed il campo valuta locale che fanno degli strani movimenti, per non parlare del campo valuta in Euro che, dopo un po' di movimento, rimane attaccato al bordo superiore e da lì non si sposta... mah. Speriamo che l'applicazione vera e propria si comporti correttamente....

Se uno non vuole complicarsi troppo la vita, può decidere di mantenere la finestra di dimensioni fisse, e quindi eliminare il problema del ridimensionamento alla radice. Allo stato attuale, l'unico metodo che ho trovato è quello di dire che la finestra ha dimensioni massime e minime coincidenti (sono valori che posso assegnare liberamente da IB, basta selezionare la finestra ed andare sullo `Inspector`...). In ogni caso, ne approfitto anche per dire che la mia finestra NON avrà il pulsante di chiusura finestra. L'applicazione quando parte apre la finestra, e non c'è verso di fare scomparire questa finestra, se non uscendo dall'applicazione (si può sempre iconizzare, ovviamente). Così facendo, elimino alla radice il problema di aprire una nuova finestra nel caso l'utente decida di chiuderla.

L'APPLICAZIONE IN QUATTRO ISTRUZIONI





Conclusa la preparazione dell'interfaccia, passiamo velocemente alla realizzazione dell'applicazione. Sappiamo già che dobbiamo costruire una classe Controllore, che battezzo immantinente `EuroCtrl`. Questa classe avrà una unica istanza, un oggetto che chiamo `oEuroCtrl`. Alla classe associo un po' di azioni (metodi) e outlet.

Le azioni sono tre:

- `cvt2Euro`, per convertire in Euro
- `cvt2Local`, per convertire in valuta locale
- `cvtMsg`, per cambiare il messaggio informativo

Gli outlet sono anch'essi tre, e corrispondono ai tre elementi dell'interfaccia con i quali si interagisce, ovvero il campo della valuta locale (`setLocal`), il campo della valuta in Euro (`setEuro`) ed il campo informazioni (`setMsg`). Ci metto più tempo a dirlo che a fare che connessioni.

Sempre velocemente, impostiamo le connessioni: dal menu pop-up verso `oEuroCtrl` con action `cvtMsg`, dal campo valuta locale verso `oEuroCtrl` con action `cvt2Euro`, dal campo valuta in Euro verso `oEuroCtrl` con action `cvt2Local`.

A questo punto generiamo lo scheletro del codice sorgente per la classe EuroCtrl, ed il nostro lavoro con IB è terminato.

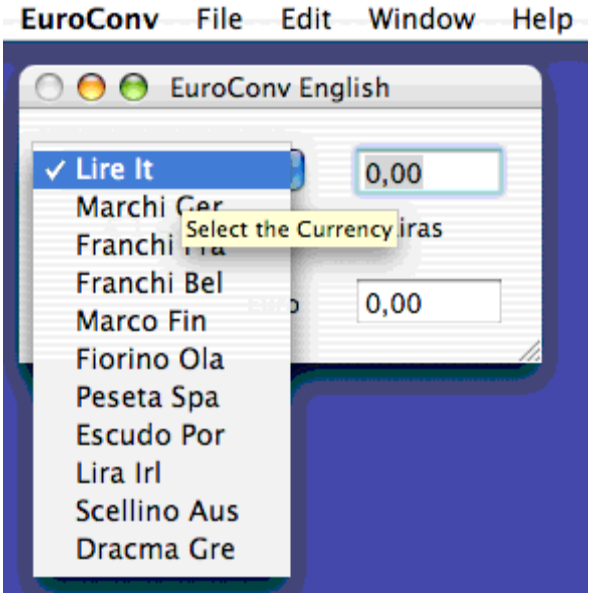
UN PO' DI CODICE

Siamo adesso in XCode. L'applicazione che stiamo facendo è un classico esempio in cui il paradigma M-V-C perde un po' di significato, in quanto la classe modello brilla per la sua assenza (fossimo in SmallTalk, avrei definito una classe Euro come modello, con metodi per conversioni da e verso valute locali, ma sarebbe tirare fuori sangue da una rapa).

Le operazioni da svolgere sono molto banali, anche se occorre scrivere molte righe per organizzarsi. In primo luogo, abbiamo bisogno dei tassi di conversione tra Euro e le altre valute. Utilizzo il buon vecchio `#define` del linguaggio C per costruirmi degli identificatori adatti. Già che ci sono, mi faccio un po' di identificatori per attribuire dei nomi sensati ai vari paesi, piuttosto che avere a che fare con dei numeri. Sono quindi arrivato a questo punto:

```
#define         NUM_EURO_NATIONS    11

#define         EURO_ITALY          0
#define         EURO_GERMANY        1
#define         EURO_FRANCE         2
#define         EURO_BELGIO         3
#define         EURO_FINLAND        4
```

```
#define         EURO_OLANDA          5
#define         EURO_SPAGNA          6
#define         EURO_PORTOGAL        7
#define         EURO_IRLANDA         8
#define         EURO_AUSTRIA         9
#define         EURO_GRECIA         10

#define         EURO_ITALY_CVT        (1936.27)
#define         EURO_GERMANY_CVT      (1.95583)
#define         EURO_FRANCE_CVT       (6.55957)
#define         EURO_BELGIO_CVT       (40.3399)
#define         EURO_FINLAND_CVT      (5.94573)
#define         EURO_OLANDA_CVT       (2.200371)
#define         EURO_SPAGNA_CVT       (166.386)
#define         EURO_PORTOGAL_CVT     (200.482)
#define         EURO_IRLANDA_CVT      (0.78756)
#define         EURO_AUSTRIA_CVT      (13.7603)
#define         EURO_GRECIA_CVT       (340.75)
```



Ho avuto l'accortezza di attribuire i numeri nello stesso ordine con cui ho definito le voci del menu pop-up. Ho infatti sbirciato la definizione dell'oggetto menu pop-up e scoperto che la classe relativa `NSPopUpButton` ha un metodo dal nome `indexOfSelectedItem`, che restituisce appunto un indice che determina quale voce di menu è attualmente selezionata. Ciò mi permette di mettere tutti i rapporti di conversione in un vettore e di utilizzare il rapporto corretto in base all'indice dell'elemento selezionato dal menu. Allo scopo ho dunque dichiarato all'interno della classe `EuroCtrl` due variabili d'istanza di questo tipo:

```
short         currSel ;
float         cvtRate[ NUM_EURO_NATIONS ];
```

La prima mi serve per tenere traccia dell'elemento selezionato (avrei potuto fare altrimenti: aggiungere un outlet ed attribuirgli un collegamento al menu pop-up, ed interrogarlo ogni volta che mi serviva sapere quale fosse la valuta in gioco), la seconda è un vettore che tiene conto di tutti i rapporti di conversione. Questo vettore va ovviamente riempito.

SVEGLIATI!

Nel capitolo precedente, queste operazioni di inizializzazione erano svolte all'interno di un metodo `init` che sovrascriveva quello standard. Ho scoperto un metodo differente e direi migliore (non c'è bisogno di fare l'inizializzazione completa), valido però solo per gli oggetti che si trovano dentro un file nib e che sono creati automaticamente (dall'ambiente Cocoa) al lancio dell'applicazione.

Infatti, quando l'ambiente Cocoa lancia l'applicazione, carica il file nib principale (qui ce ne è uno solo, quindi `MainMenu.nib`) e costruisce tutti gli oggetti presenti in esso. Di seguito, prima di rappresentarli a video, invia a ciascuno di essi il messaggio `awakeFromNib` (senza argomento, e senza attendersi risposta). È questo un ottimo posto per fare un po' di cose. Ad esempio, riempire il vettore con i rapporti di conversione, e poi impostare valori di default per i vari campi (se mai volessi qualcosa diverso dallo zero che ho impostato in IB).

Ecco quindi la mia realizzazione del metodo (un po' noioso, mi rendo conto):

```
- (void) awakeFromNib
{
    cvtRate[ EURO_ITALY ]          = EURO_ITALY_CVT ;
    cvtRate[ EURO_FRANCE ]          = EURO_FRANCE_CVT ;
    cvtRate[ EURO_GERMANY ]      = EURO_GERMANY_CVT ;
    cvtRate[ EURO_BELGIO ]          = EURO_BELGIO_CVT ;
    cvtRate[ EURO_FINLAND ]      = EURO_FINLAND_CVT ;
    cvtRate[ EURO_SPAGNA ]          = EURO_SPAGNA_CVT ;
    cvtRate[ EURO_OLANDA ]          = EURO_OLANDA_CVT ;
    cvtRate[ EURO_PORTOGAL ]      = EURO_PORTOGAL_CVT ;
    cvtRate[ EURO_IRLANDA ]      = EURO_IRLANDA_CVT ;
    cvtRate[ EURO_AUSTRIA ]      = EURO_AUSTRIA_CVT ;
    cvtRate[ EURO_GRECIA ]          = EURO_GRECIA_CVT ;

    currSel = 0 ;
}
```

Passiamo adesso all'azione (metodo) per la conversione da valuta locale in Euro, ovvero `cvt2Euro`. Sono riuscito a farcela in una sola riga:

```
- (IBAction)cvt2Euro:(id)sender
{
    [ setEuro setFloatValue: [ sender floatValue ] / cvtRate[ currSel ] ] ;
}
```

Vediamo bene: `sender` è l'oggetto che ha inviato la richiesta: si tratta del campo che contiene la valuta locale. Allora, lo uso per ricavare il valore lì presente:

```
[ sender floatValue ]
```

il messaggio è parente del metodo visto nel capitolo precedente per impostare un valore intero all'interno di un campo testo. Lì si usava `setIntValue`, qui usiamo `floatValue` se vogliamo estrarre un valore floating point (con la virgola) o anche `intValue` se si intende estrarre un valore intero.

Torniamo all'euroconvertitore. Uso la variabile `currSel` per determinare quale rapporto di conversione utilizzare:

```
cvtRate[ currSel ]
```

Qui le parentesi quadre individuano un vettore, e non sono da confondere con le parentesi quadre che invece (come nel pezzetto precedente) indicano la trasmissione di un messaggio.

Piglio il valore da convertire e lo divido per il rapporto di conversione, in modo da avere l'ammontare in Euro:

```
[ sender floatValue ] / cvtRate[ currSel ]
```

Questa espressione è un numero floating point, e lo uso come argomento del messaggio `setFloatValue` inviato al campo che mostra la valuta in Euro, individuato dall'outlet `setEuro`:

```
[ setEuro setFloatValue: <valore> ]
```

Finito.

Vi risparmio il commento sull'altro metodo che trasforma gli Euro in valuta locale:

```
- (IBAction)cvt2Local:(id)sender
{
    [ setLocal setFloatValue: cvtRate[ currSel ] * [ sender floatValue ] ]
;
}
```

L'ULTIMO METODO

L'ultimo metodo, quello che scatta quando l'utente cambia la voce del menu, comporta molto lavoro aggiuntivo, dovuto essenzialmente al cambiamento del messaggio informativo.

Se infatti lo ignorassimo, il metodo sarebbe molto semplice, due istruzioni:

```
- (IBAction)cvtMsg:(id)sender
{
    currSel = [ sender indexOfSelectedItem ];
    [ setLocal setFloatValue: [ setEuro floatValue ] * cvtRate[ currSel ] ]
;
}
```

Con la prima istruzione, cambio il valore della variabile d'istanza `currSel` per tenere traccia della variazione del menu; per fare questo, basta inviare il già citato messaggio `indexOfSelectedItem` all'oggetto scatenante il metodo, che è proprio il menu pop-up. La seconda istruzione non è altro che un altro modo di attribuire un valore al campo della valuta locale, pigliando il valore direttamente dal campo utilizzando l'outlet che lo identifica (avrei potuto usare la stessa espressione anche nel metodo `cvt2Local`, ma

così avrei *perso* l'argomento `sender`, che sarebbe servito a nulla...).

Ma ci sono le stringhe da modificare. E questo apre un interessante problema di localizzazione e di gestione delle risorse che discuterò nel prossimo capitolo, a cui pieroangelicamente vi rimando.

Capitolo 011 - Localizzazione icone ed altre storie

Il corredo di una applicazione.

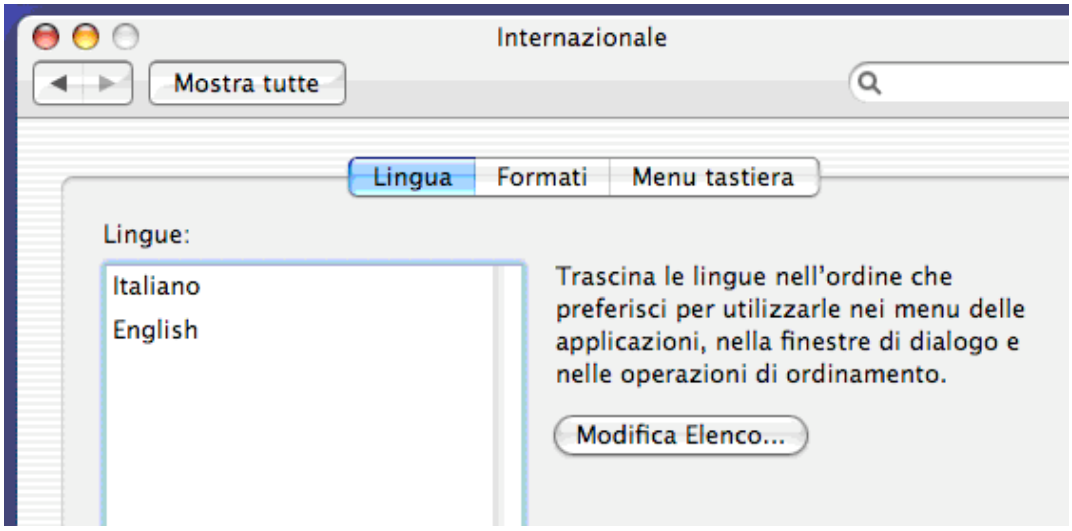
Sorgenti: Manualistica Apple

Primo inserimento: 26 Novembre 2001

Ritocchi per aggiornamento XCode 2.1: 9 Agosto 2005

STRINGHE IN LINGUA

EuroConv è un programma che nel suo piccolo si presta naturalmente ad essere utilizzato in diversi paesi europei. Nel caso di apparecchiature, dispositivi, automobili, elettrodomestici, eccetera, è una legge dello stato fornire le istruzioni ed in generale l'interfaccia utente nella lingua dell'utilizzatore. È buona cosa farlo anche nel caso di programmi per calcolatore, ed è proprio il primo problema che mi pongo in questo capitolo.

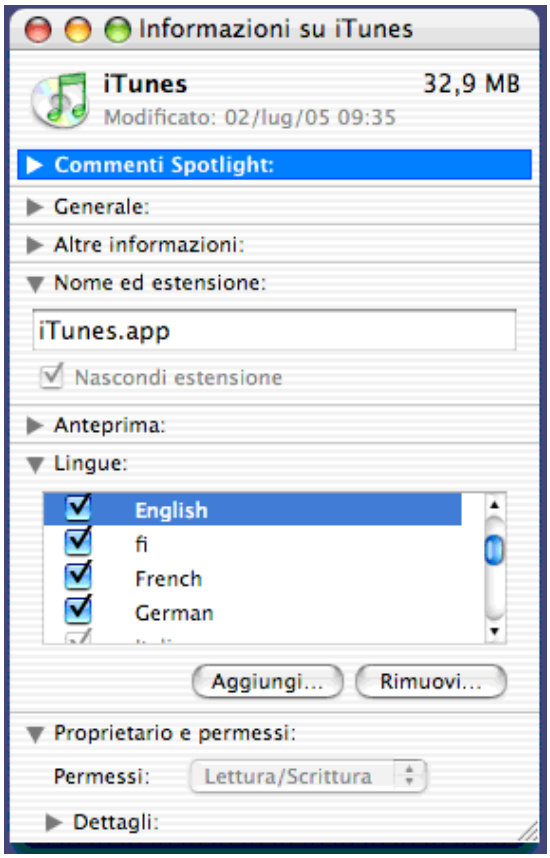


Ho deciso che quando l'utente seleziona una delle voci del menu pop-up, è mostrata al centro della finestra, in una stringa di testo, il rapporto di conversione tra l'Euro e la valuta selezionata. Selezionando quindi `Marchi Ger` deve saltare fuori la stringa '`Euro 1 = 1.95583 Marchi tedeschi`', e sia così per tutte le altre. Questa stringa pone un problema quando EuroConv sarà utilizzato da persone non italiane. Sarebbe quindi giusto che la stringa sia scritta nella lingua scelta nel pannello Preferenze di sistema. Dal punto di vista del programmatore, l'adeguamento alla lingua (operazione nota come **localizzazione** o **internazionalizzazione**) genera tipicamente un sovraccarico di lavoro piuttosto noioso e non focalizzato sull'applicazione vera e propria. Apple opera una sottile distinzione tra i due concetti. Si localizza quando l'intera applicazione è adattata ai vari mercati locali; sono comprese quindi anche le operazioni di traduzione

del manuale, predisposizione delle strutture di supporto, eccetera.

L'internazionalizzazione invece riguarda più strettamente il supporto al programmatore per la stesura di applicazioni nelle diverse lingue. Ogni ambiente di sviluppo fornisce in varia misura un supporto per adeguare l'interfaccia utente nelle varie lingue. L'ambiente Cocoa non è da meno: ereditando alcuni concetti da Unix, unendone altri di originali (derivati in realtà da Next) fornisce una serie di strumenti per la localizzazione delle applicazioni.

LINGUE DELL'APPLICAZIONE



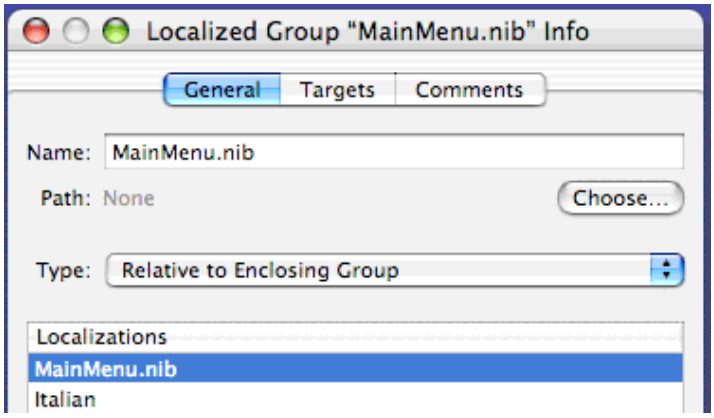
Una parte consistente di localizzazione consiste nell'adeguare le voci dei menu, l'aspetto delle finestre, il testo dei pulsanti e di tutte le stringhe *statiche* dell'interfaccia. Nei precedenti sistemi operativi Macintosh (ma non è l'unico sistema operativo che soffre di questo problema) non c'era altra possibilità che rilasciare diverse applicazioni per le diverse lingue. Esisteva quindi l'applicazione in inglese e l'applicazione in italiano. Chi ha provato a cimentarsi con l'opera di localizzazione, sa che lo strumento principe era l'applicazione ResEdit unito ad una lunga pazienza da certosino. Con Mac Os X le cose sono molto diverse: esiste una unica applicazione che contiene al suo interno tutte le risorse per la localizzazione nelle varie lingue. Questo significa che lo stesso file (che so, `iTunes`) contiene nel suo interno tutte le informazioni per la localizzazione nelle varie lingue (o almeno, nelle lingue previste da chi ha rilasciato l'applicazione). Se infatti seleziono l'icona di iTunes e ne chiedo le Informazioni, tra i pannelli disponibili c'è anche quello relativo alle lingue supportate. Nel mio caso sono disponibili Italiano ed Inglese. Questo significa che, modificando la lingua principale attraverso le preferenze di sistema, è possibile avere l'interfaccia utente in italiano oppure in inglese.

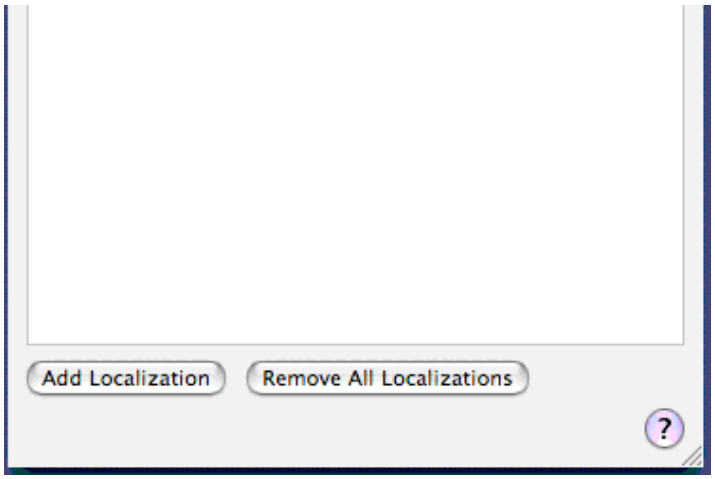
Tale disponibilità di lingua si riflette su come è costruita l'applicazione. Attraverso il Terminale, è possibile esaminare il contenuto dell'applicazione (ecco, questa è una cosa che occorrerà approfondire prima o poi, ovvero, come sono conservate le applicazioni; per ora basti dire che un'applicazione in Mac Os X non è altro che una cartella, di tipo speciale. All'interno di questa cartella trovano posto il vero e proprio codice eseguibile, assieme a tutte le altre risorse necessarie all'applicazione stesse, suoni, immagini e quant'altro...).

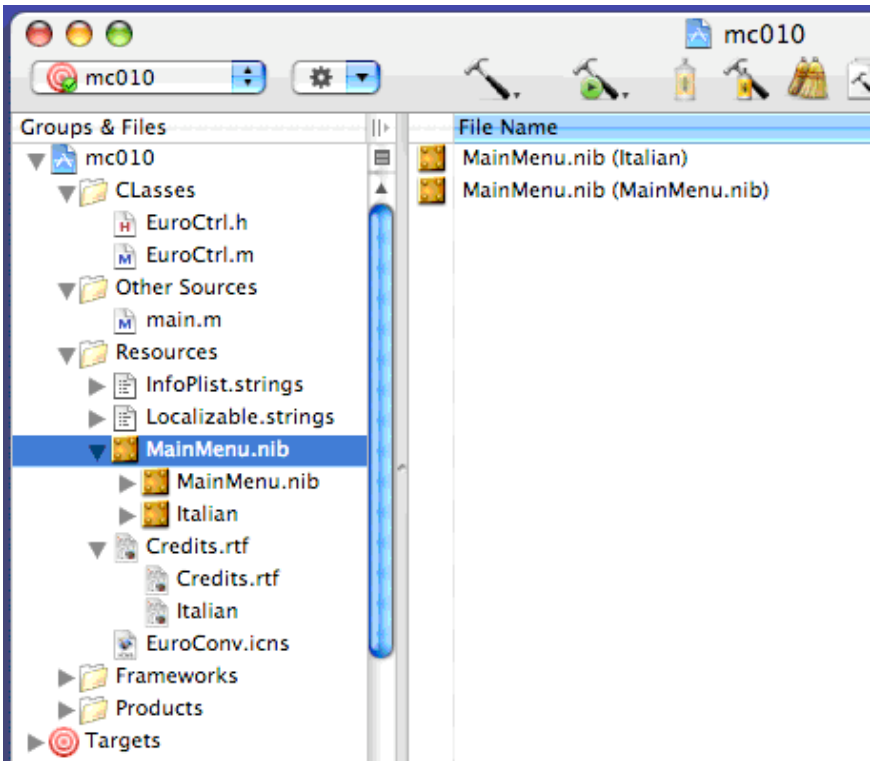
```
PowerDjZero:/Applications/iTunes.app/Contents djzero$ ls -la
total 48
drwxrwxr-x    9 root    admin    306 Aug  9 20:45 .
drwxrwxr-x    3 root    admin    102 Jul  2 09:35 ..
drwxrwxr-x    3 root    admin    102 Mar 28 06:27 Frameworks
-rw-rw-r--    1 root    admin  14938 Jun 23 00:10 Info.plist
drwxrwxr-x    3 root    admin    102 Jul  2 09:34 MacOS
-rw-rw-r--    1 root    admin      8 Jun 16 22:33 PkgInfo
drwxrwxr-x   44 root    admin   1496 Jul  2 09:35 Resources
drwxr-xr-x    2 djzero  admin     68 Aug  9 20:45 Resources Disabled
-rw-rw-r--    1 root    admin    510 Jun 28 23:31 version.plist
PowerDjZero:/Applications/iTunes.app/Contents djzero$ ls -la Resources
total 8808
drwxrwxr-x   44 root  admin     1496 Jul  2 09:35 .
drwxrwxr-x    9 root  admin      306 Aug  9 20:45 ..
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 Dutch.lproj
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 English.lproj
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 French.lproj
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 German.lproj
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 Italian.lproj
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 Japanese.lproj
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 Spanish.lproj
-rw-rw-r--    1 root  admin    36570 Jun 16 22:18 complete.aif
drwxrwxr-x    6 root  admin      204 Jul  2 09:34 da.lproj
        __ altri file che non ci interessano __
```

Il contenuto della cartella `/Application/iTunes.app/Contents/` presenta alcuni file e cartelle; in particolare, scendendo di livello, nella cartella `Resources` sono infine presenti altre due cartelle, `English.lproj` e `Italian.lproj`. Ebbene, queste due cartelle contengono appunto tutte le risorse *localizzate* per la lingua indicata nel nome della cartella. Se iTunes fosse localizzato anche per la lingua aramaica, ci sarebbe stata una cartella `Aramaic.lproj` (o qualcosa del genere), e via di questo passo per le altre lingue.

SUPPORTO ALL'INTERNAZIONALIZZAZIONE

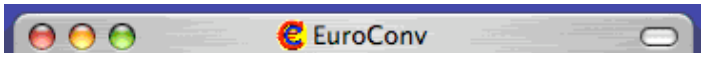


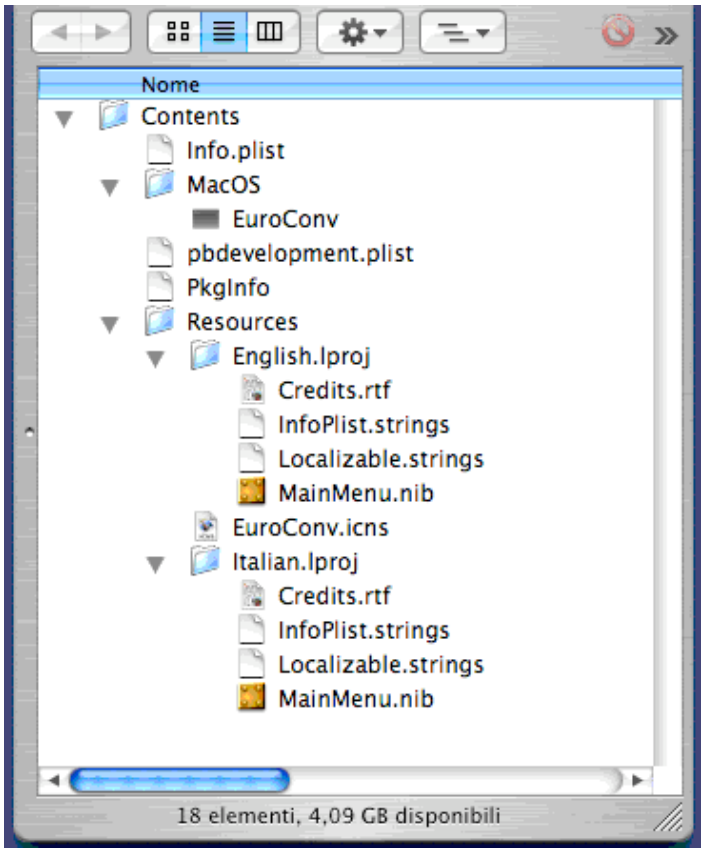




Per realizzare una struttura similare nella applicazione EuroConv, torno nel XCode e seleziono le risorse che voglio far comparire in versione localizzata. Al momento, ho solo il file nib che contiene finestra e menu. Seleziono il file e scelgo di vedere le informazioni relative (menu contestuale e poi `Get Info`). C'è un pulsante che permette l'inserimento una variante locale: `Add Localization`; nel pannello che appare scrivo `Italian`. Di colpo, il file nib si duplica, e ne compaiono due, uno che rimane `MainMenu.nib`, versione di default in inglese, ed un altro che si chiama `Italian`. Su questo file nib posso adesso lavorare con IB modificando tutti gli elementi dell'interfaccia che ritengo opportuno modificare per adeguati alla lingua (in realtà, essendo baldanzosamente partito con l'italiano, devo internazionalizzare il tutto).

E per quanto riguarda l'interfaccia utente, non devo fare altro. È compito dell'ambiente Cocoa e del sistema operativo utilizzare l'uno o l'altro file nib al lancio dell'applicazione per visualizzare l'interfaccia in lingua.





Infatti, compilando il progetto, si genera alla fine l'applicazione. L'esame del contenuto del prodotto si può fare direttamente dal Finder: seleziono da Finder il prodotto della compilazione e, mediante menu contestuale, scelgo `Mostra Contenuto Pacchetto`. Si apre una finestra del Finder con dentro la cartella `Contents`, e da qui, aprendola, si scopre la struttura già descritta, con due cartelle `English` ed `Italian` dense di file di risorse (la figura si riferisce all'applicazione in uno stadio più avanzato di quanto finora ho descritto, ma ci arriviamo presto). Al momento voglio notare il file `mc010` nella cartella `Contents/MacOS`, che contiene il codice eseguibile vero e proprio, e i due file `MainMenu.nib` nelle due cartelle `English.lproj` e `Italian.lproj`, che corrispondono ai due file nib che ho duplicato e manipolato per metterli in lingua. Degli altri file, di qualcuno ne parlo dopo, di altri ignoro totalmente cosa siano e a cosa servono (ma non credano di resistermi a lungo).

STRINGHE LOCALIZZATE

Torno adesso al problema lasciato in sospeso il capitolo precedente, ovvero recuperare le stringhe di informazione da mostrare nella finestra quando l'utente cambia attraverso il menu pop-up la valuta di riferimento. Il problema era appunto di inserire al stringa corretta in dipendenza dalla lingua dell'utente. Ciò si realizza facilmente utilizzando il meccanismo sopra descritto di file localizzato, e la funzione `NSLocalizedString` (la documentazione afferma in realtà trattarsi di una macro, ma la cosa al momento non mi interessa).

In primo luogo, occorre costruire un file di stringhe, ovvero un file dall'estensione `string`, costruito da una serie di righe siffatte:

```
<<stringa chiave>> = <<stringa risultante>> ;
```

La stringa chiave è il *nome* della stringa, il cui contenuto è proprio la stringa risultante. Il file delle stringhe può avere un nome qualsiasi; il nome più comune è `Localizable.string`, utilizzato come default. Questo file di stringhe va poi replicato per tutte le lingue che ci interessa supportare, utilizzando appunto il meccanismo sopra visto di creazione di una versione localizzata del file. Vale a dire, costruisco un nuovo file dove sono contenute le stesse stringhe chiave, ma la stringa risultante è nella lingua desiderata. Nel caso di EuroConv, i miei due file sono così costruiti. Il file corrispondente all'italiano è:

```
/* info conversione da visualizzare */
"it" = "\U20ac 1 = 1936.27 Lire italiane";
"au" = "\U20ac 1 = 13.7603 Scellini austriaci";
"be" = "\U20ac 1 = 40.3399 Franchi belgi";
"de" = "\U20ac 1 = 1.95583 Marchi tedeschi";
"es" = "\U20ac 1 = 166.386 Peseta spagnole";
"fi" = "\U20ac 1 = 5.94573 Marchi finlandesi";
"fr" = "\U20ac 1 = 6.55957 Franchi francesi";
"gr" = "\U20ac 1 = 340.75 Dracme greche";
"ir" = "\U20ac 1 = 0.78756 Lire irlandesi";
"nl" = "\U20ac 1 = 2.200371 Fiorini olandesi";
"por" = "\U20ac 1 = 200.482 Escudo portoghesi";
```

mentre quello per l'inglese è il seguente (e perdonate la mia ignoranza sui nomi dei paesi e delle monete in inglese... anzi, se ne sapete più di me, correggetemi):

```
/* info conversione da visualizzare in inglese */
"it" = "\U20ac 1 = 1936.27 Italian Liras";
"au" = "\U20ac 1 = 13.7603 Austrian shellings";
"be" = "\U20ac 1 = 40.3399 Belgian francs";
"de" = "\U20ac 1 = 1.95583 German marks";
"es" = "\U20ac 1 = 166.386 Spanish pesetas";
"fi" = "\U20ac 1 = 5.94573 Finnish marks";
"fr" = "\U20ac 1 = 6.55957 French francs ";
"gr" = "\U20ac 1 = 340.75 Grecian dracmas";
"ir" = "\U20ac 1 = 0.78756 Irish liras";
"nl" = "\U20ac 1 = 2.200371 Dutch ???";
"por" = "\U20ac 1 = 200.482 Portuguese Escudos";
```

Non vi sarà sfuggita la bizzarra sequenza iniziale di caratteri e numeri. Ebbene, `\U20ac` non è altro che la rappresentazione del simbolo dell'Euro (€) nella rappresentazione Unicode dei caratteri. La codifica Unicode ha sostituito completamente la vecchia codifica ASCII, utilizzata praticamente fin dalla nascita dei calcolatori come equivalente numerico dei caratteri. Unicode è la rappresentazione *nativa* dei caratteri all'interno del Mac OS X.

Adesso entra in gioco la macro `NSLocalizedString`. Supponiamo ad un certo punto di aver bisogno all'interno del nostro codice di una certa stringa. Uso la macro per recuperarne la versione nella lingua:

```
<<stringa risultante>> = NSLocalizedString( <<stringa chiave>>,
<<commento>>);
```

La macro sorella `NSLocalizedSringFromTable` dovrebbe funzionare nei casi in cui

il file dalle stringhe non si chiama `Localizable.string`, ma con qualche altro nome (ad esempio, perché ho costruito diverse file di stringhe per comprensibilità nella costruzione del programma).

EUROCONV EUROPEO

A questo punto, diventa molto semplice localizzare la stringa di informazioni dell'applicazione EuroConv.

In primo luogo, definisco nel file `EutoCtrl.h` le chiavi delle stringhe:

```
#define                 ITALY_STRING        @"it"
#define                 GERMANY_STRING      @"de"
#define                 FRANCE_STRING       @"fr"
#define                 BELGIO_STRING       @"be"
#define                 FINLAND_STRING      @"fi"
#define                 OLANDA_STRING       @"nl"
#define                 SPAGNA_STRING       @"es"
#define                 PORTOGAL_STRING     @"por"
#define                 IRLANDA_STRING      @"ir"
#define                 AUSTRIA_STRING      @"au"
#define                 GRECIA_STRING       @"gr"
```

Aggiungo una variabile d'istanza in grado di conservarle (in realtà, un vettore), utilizzando la stessa idea dei rapporti di conversione tra Euro e valuta locale:

```
NSString        * cvtMsg[ NUM_EURO_NATIONS ];
```

Il vettore va inizializzato all'interno del metodo awakeFromNib:

```
cvtMsg [ EURO_ITALY ]           = ITALY_STRING ;
cvtMsg [ EURO_FRANCE ]          = FRANCE_STRING ;
cvtMsg [ EURO_GERMANY ]         = GERMANY_STRING ;
cvtMsg [ EURO_BELGIO ]          = BELGIO_STRING ;
cvtMsg [ EURO_FINLAND ]         = FINLAND_STRING ;
cvtMsg [ EURO_SPAGNA ]          = SPAGNA_STRING ;
cvtMsg [ EURO_OLANDA ]          = OLANDA_STRING ;
cvtMsg [ EURO_PORTOGAL ]        = PORTOGAL_STRING ;
cvtMsg [ EURO_IRLANDA ]         = IRLANDA_STRING ;
cvtMsg [ EURO_AUSTRIA ]         = AUSTRIA_STRING ;
cvtMsg [ EURO_GRECIA ]          = GRECIA_STRING ;
```

Finalmente all'interno del metodo `cvtMsg` siamo arrivati alla stesura finale:

```
- (IBAction)cvtMsg:(id)sender
{
        currSel = [ sender indexOfSelectedItem ];
           [ setMsg setStringValue: NSLocalizedString( cvtMsg[ currSel ],
@"cheneso" ) ];
         [ setLocal setFloatValue: [ setEuro floatValue ] * cvtRate[ currSel
] ] ;
}
```

La prima istruzione, come detto, recupera l'identificatore della valuta selezionata; utilizzo la chiave corrispondente `cvtMsg[currSel]` per recuperare la stringa dal file

della lingua scelto per me dal sistema operativo e dall'ambiente Cocoa:

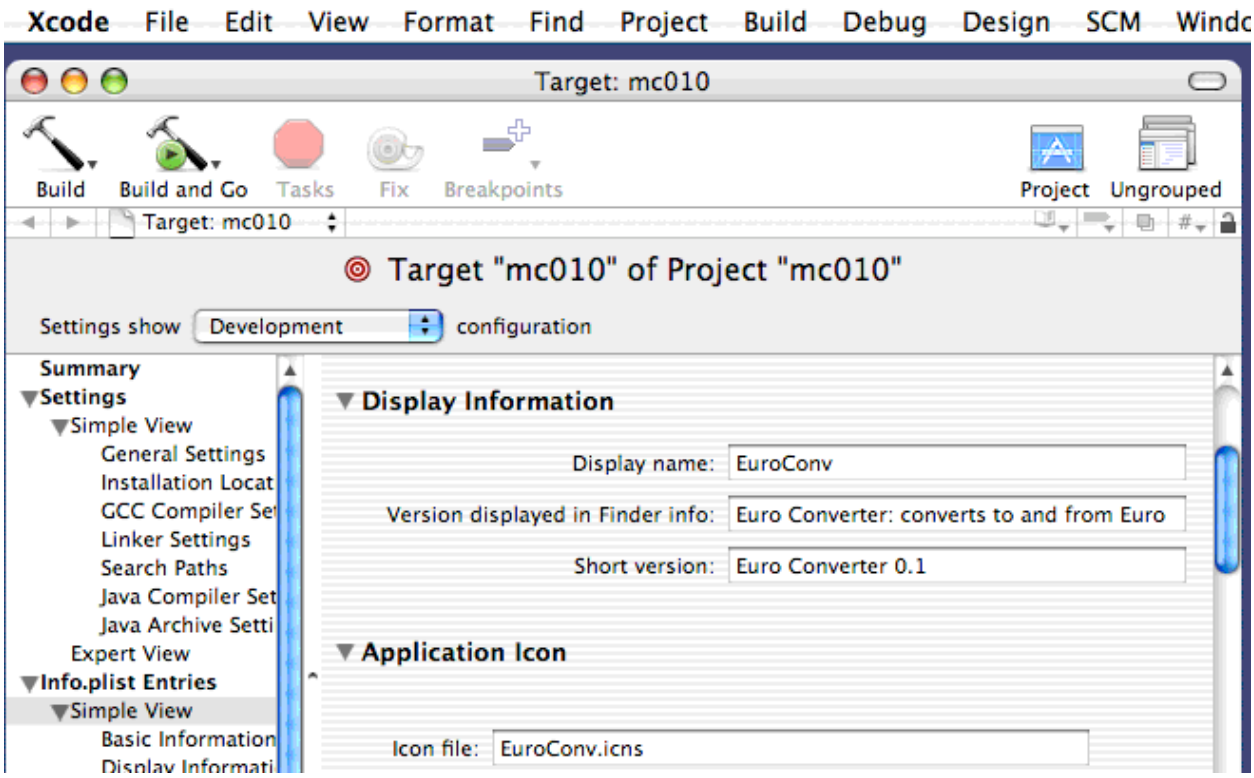
```
NSLocalizedString( cvtMsg[ currSel ], @"cheneso" )
```

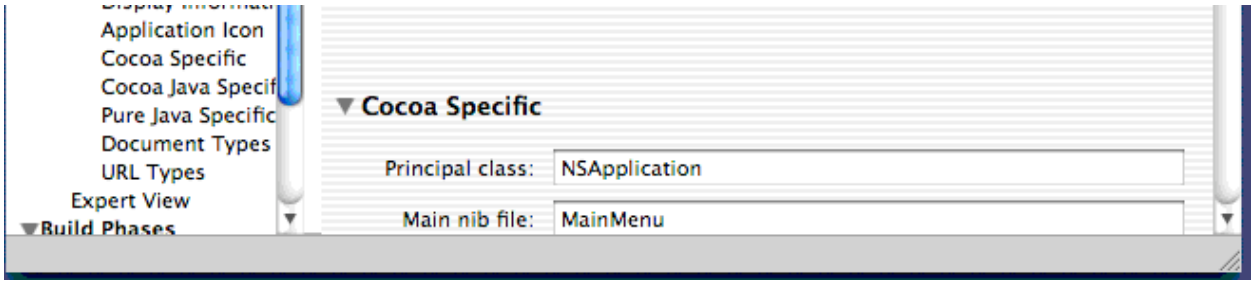
La stringa risultante la uso come argomento del messaggio `setStringValue` che modifica il contenuto del campo informazioni nell'interfaccia. Finito. L'istruzione successiva, come già detto, effettua il cambiamento del valore in Euro nella nuova valuta locale.

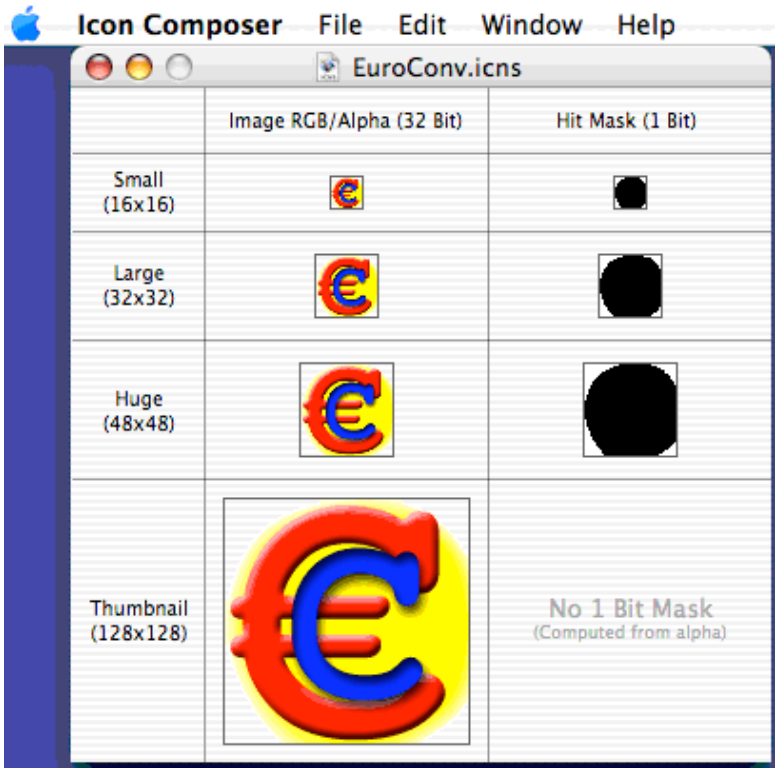
Rimane da chiarire l'uso del commento all'interno della macro `NSLocalizedString`. In effetti, non ha qui alcuna funzione. È però utilizzata come supporto all'internazionalizzazione. Accade che il programmatore non conosca tutte le lingue del mondo, ma solo la sua. Allora, quando ha bisogno di una stringa localizzata, non si preoccupa minimamente della sua rappresentazione, ma usa `NSLocalizedString` con la stringa chiave. Aggiunge poi un commento sul significato effettivo della stringa (la cosa più praticata è di mettere la stringa effettiva nella lingua principale, in inglese) di modo che qualcun altro, linguisticamente più dotato, scriva le stringhe corrette. Questo meccanismo non è proprio di Cocoa, ma lo ricordo adottato (con nomi differenti, ovviamente) all'interno di Unix e sviluppo di interfacce in Xwindow. Lì esisteva un programma (a linea di comando) che esaminava tutti i file sorgenti dell'applicazione e produceva direttamente tutti i file di stringhe necessari, raccogliendo chiavi e stringhe risultanti. In Mac Os X, esiste una funzione simile, ma non riesco più a trovare la pagina del manuale dove ne avevo appreso l'esistenza.

Daniele Baracchi mi suggerisce che il programma si chiama `genstrings`. Trovate le informazioni relative attraverso il terminale e digitando `man genstrings` oppure nella sezione 'Internationalizing Your Software' nella documentazione dell'ambiente di sviluppo.

ICONE



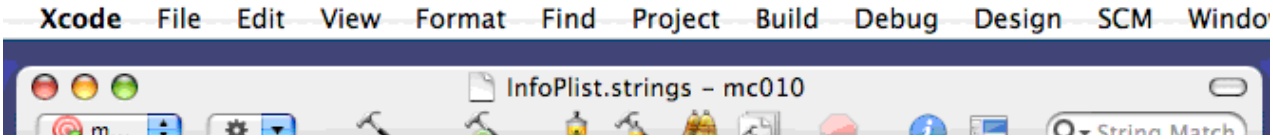


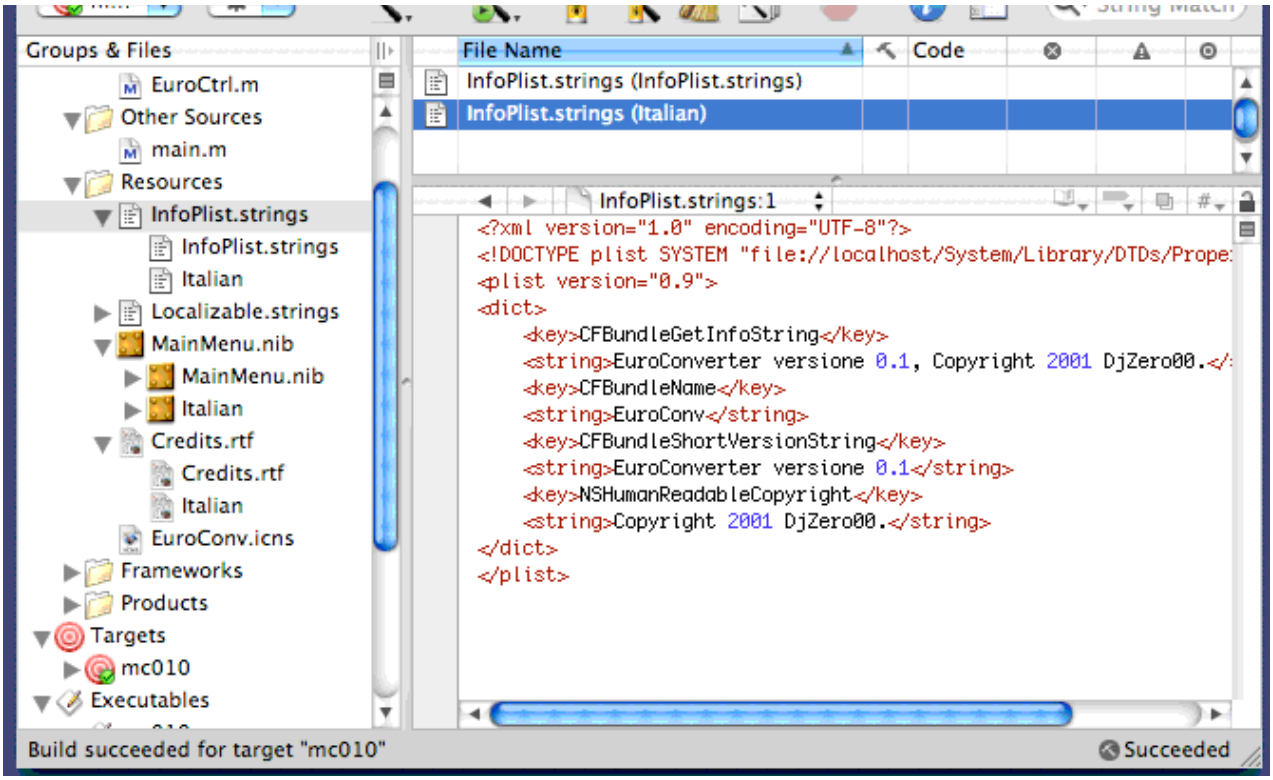


Aggiungere l'icona all'applicazione è estremamente semplice. Ovviamente, la prima cosa da fare è disegnare l'icona. Occorre una immagine di partenza costituita da un quadrato di 128 pixel di lato; io ho prodotto una cosa orribile a vedersi. Apro l'applicazione IconComposer nei Developer Tools; con la tecnica del drag and drop trascino il file sulle quattro dimensioni previste di icona, e lascio all'applicazione il compito di ridimensionarla e di calcolarne la maschera ad un bit. Salvo il risultato in un file che chiamo fantasiosamente `Euroconv.icns`; ripasso in XCode e lo aggiungo al progetto, nella sezione risorse. Il passo successivo è associare questo file alle icone del programma; basta andare nel pannello delle informazioni del `Target` sotto esame e scrivere il nome del file nell'apposito spazio.

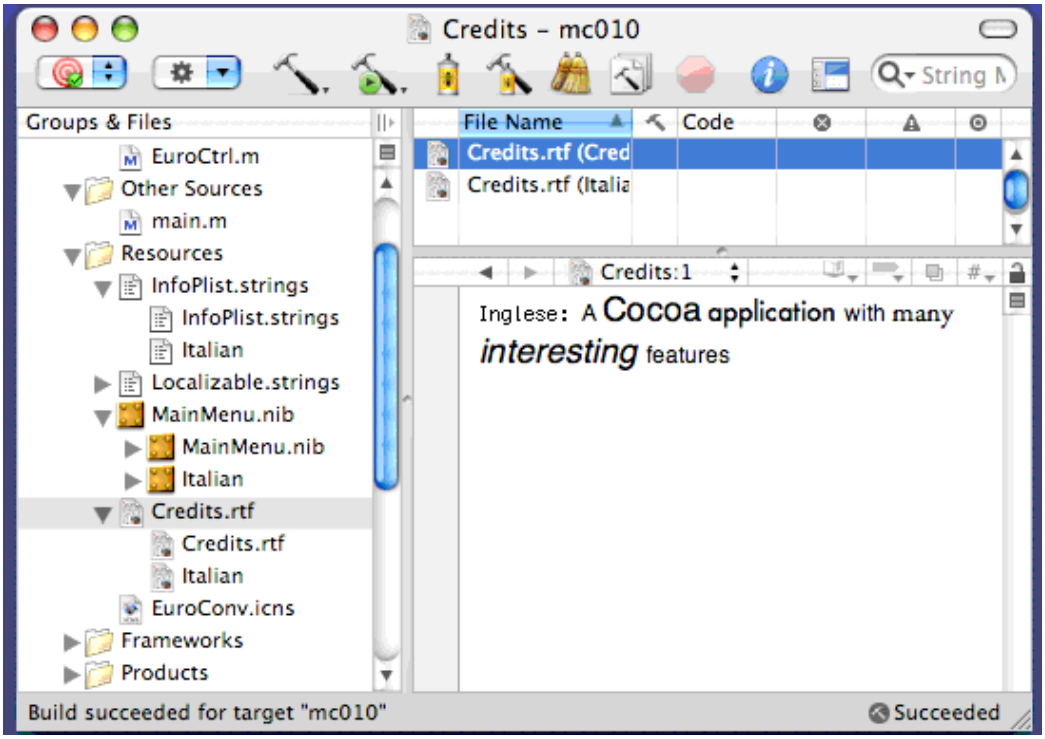
Già che sono ho riempito un altro paio di campi, utili per cambiare il nome del file eseguibile (altrimenti XCode usa di default il nome del progetto), informazioni da mostrare nella finestra del Finder, eccetera.

ABOUT





Per chiudere, personalizzo la finestra di About dell'applicazione. In primo luogo utilizzo il file `InfoPlist.string`, costruito da XCode nel momento di creazione del progetto. Lo divido subito in due versioni locali (una per l'inglese e l'altra per l'italiano), e modifico le stringhe proposte da XCode con le informazioni che mi piacciono.



Poi, costruisco, ad esempio con TextEdit, un file in formato RTF (che per altro è il formato nativo di TextEdit) dal nome `Credits.rft`. Ebbene, questo file, una volta inserito nel progetto, è mostrato dalla finestra di About tale e quale (anche fosse piuttosto corposo...). Ciò permette di personalizzare velocemente e senza fatica la finestra di About. Per avere qualcosa di completamente diverso (che so, credits scorrevoli, animazioni e quant'altro), credo che si debba lavorare un bel po'. Non è questo il momento.

Per ora mi accontento di aver costruito una applicazione funzionante, bilingue, magari di non troppa utilità pratica, ma abbastanza completa. Fra un po', potrei anche cominciare a spacciarne in giro come freeware...

Capitolo 012 - Questioni di memoria

È stato un capitolo sofferto. Ero partito baldanzoso con l'idea del capitolo bene in testa, mi sono lanciato allegramente nella codifica e già stavo preparando le parole per scrivere il capitolo, quando è successo l'imprevisto.

L'applicazione non funzionava. Si suicidava velocemente dopo l'apertura, oppure reggeva qualche secondo, poi moriva ingloriosamente, oppure rimaneva su un bel po', per poi morire improvvisamente. Ho cominciato allora a lavorare di debugger, e niente, non riuscivo assolutamente a capire cosa stava succedendo.

Poi, l'illuminazione. Mi sono dato dello stupido (succede spesso: quando capisco dove ho sbagliato, mi do dello stupido), in quanto non avevo tenuto presente la prima causa di errori inafferrabili: l'allocazione della memoria. E quindi, in questo capitolo si parla della genesi ed apocalisse di oggetti.

Sorgenti: Documentazione Apple, varie qui e lì

Primo inserimento: 3 Dicembre 2001

Genesi: Alloc ed Init

Ho già parlato di come nasce un oggetto. Si parte dalla Classe, modello o stampo per costruire oggetti, e gli si invia il messaggio `alloc`. In risposta a questo messaggio, l'ambiente operativo costruisce un nuovo oggetto della classe richiesta. La costruzione di un oggetto è molto semplice: il sistema operativo individua una zona di memoria di dimensione adatta a contenerlo e mette a zero ogni locazione di memoria che ne fa parte. Dopo di che, mette a posto alcune variabili interne nascoste ai comuni mortali (la variabile d'istanza `isa` che collega l'oggetto alla sua classe) e restituisce un puntatore a questa zona di memoria. È questo meccanismo che spiega le due istruzioni che definiscono un nuovo oggetto:

```
LSMioOggetto        * ilMioOggetto ;
ilMioOggetto = [ LSMioOggetto alloc ];
```

È chiaro che avere tutte le variabili d'istanza a zero è molto poco utile, per cui in genere si effettua una procedura di inizializzazione, inviando un messaggio di `init`. Il metodo associato a questo messaggio effettua l'inizializzazione delle variabili d'istanza, assegnando loro dei valori di default. Tipicamente, quando si definisce una nuova classe, occorre scrivere un nuovo metodo `init` per effettuare l'inizializzazione delle nuove variabili d'istanza (ho già discusso l'argomento in uno dei capitoli precedenti).

Non è sempre necessario usare il messaggio `init`, ma si possono utilizzare altri messaggi con degli argomenti; l'importante è in ogni caso chiamare all'interno del metodo `init` il metodo `init` della superclasse, in modo da effettuare correttamente le inizializzazioni della gerarchia delle classi.

APOCALISSE: RELEASE

Quando un oggetto non serve più, lo si butta via. Per aprire il sacco della spazzatura, si invia all'oggetto il messaggio di `release`. Ecco quindi la corretta sequenza d'uso di un oggetto:

```
LSMioOggetto        * ilMioOggetto ;
ilMioOggetto = [[ LSMioOggetto alloc ] init ];
// uso e abuso dell'oggetto
[ ilMioOggetto release ];
```

E fin qui, è tutto molto semplice e chiaro. La regola base da seguire è che chi costruisce un oggetto, è responsabile della sua vita e della sua eliminazione quando non è più necessario.

Ma supponiamo adesso che un dato oggetto sia utilizzato da due diversi enti software (altri due oggetti, ad esempio), ognuno ignaro dell'altro. Ad esempio pensiamo ad un database che contiene informazioni su dei file. Una di queste informazioni è l'icona del file, che è un'oggetto esso stesso. Creando un elemento del database principale, devo creare non solo l'oggetto che conserva le informazioni del file, ma anche l'oggetto immagine che contiene l'icona. Se adesso inserisco un nuovo file con la stessa icona, verosimilmente inserisco un nuovo oggetto file, ma utilizzo l'oggetto immagine precedente per l'icona. Ora, se cancello le informazioni (l'oggetto) del primo file, non posso buttare via anche l'immagine, che è utilizzato dall'altro oggetto. D'altra parte, questo secondo oggetto non sa che il primo oggetto non c'è più, quindi, quando cancellato, non può cancellare l'immagine, non sapendo se qualcun altro la usa.

Il metodo per risolvere questo problema è molto semplice: si tratta di contare quanti stanno usando un determinato oggetto. Esiste allo scopo una variabile d'istanza di `NSObject` (che è quindi ereditata da tutti gli oggetti Cocoa), della quale, in piena logica Object-Oriented, non è noto il nome ma solo come si usa. Con il messaggio `retainCount` è quindi possibile leggere tale variabile (la chiamo io *contavita*), mentre per modificarne il valore esistono diversi meccanismi.

LA VITA IN UN CONTATORE

L'uso della variabile contavita semplifica la gestione degli oggetti; a questo punto non ci si deve più preoccupare di chi gestisce gli oggetti condivisi, ma si lascia fare al sistema operativo. All'interno dell'ambiente operativo c'è un meccanismo che va in giro con una falce: di ogni oggetto che incontra chiede il valore del contatore interno. Se la variabile contavita è zero, nessuno sta usando quell'oggetto. La sua vita non ha più significato, e quindi verrà eliminato brutalmente inviandogli un messaggio di `dealloc` (che è il contrario di `alloc`).

Per evitare che la triste signora con la falce ci tolga da sotto il naso gli oggetti in uso, è quindi bene gestire correttamente la variabile contavita.

In primo luogo, contavita è incrementata di uno quando l'oggetto è creato dal messaggio `alloc`, oppure attraverso un'operazione di copia.

Esiste poi il messaggio `retain`, che incrementa di uno contavita. Userò `retain` ogni volta che l'oggetto che ho sottomano deve continuare a vivere almeno finché ne ho bisogno. Chiamo l'operazione **ritenuta**: inviare il messaggio `retain` significa ritenere l'oggetto.

Quando ho terminato di usarlo, invio il messaggio di `release`, che diminuisce di uno la variabile contavita. Chiamo l'operazione **rilascio**: inviare il messaggio `release` significa rilasciare l'oggetto. All'interno del metodo che realizza `release`, dopo l'operazione di decremento si controlla il valore della variabile. Se è zero, invia all'oggetto il messaggio di `dealloc`.

Nella gestione di questa variabile sta il problema che mi ha fatto perdere un bel po' di tempo. Ecco il pezzo di codice incriminato.

Ho una classe che ha come variabile d'istanza un oggetto (non preoccupatevi dei nomi che non sapete, non sono essenziali alla comprensione); il file .h mostra le righe:

```
@interface LSDataSource : NSObject
{
        NSMutableArray        *listaFile ;
}
```

Poiché bisogna inizializzare la variabile `listaFile` alla creazione dell'oggetto `LSDataSource`, ho riscritto il metodo `init` come segue:

```
- (id ) init
{
        self = [ super init ] ;
        listaFile = [ NSMutableArray array ] ;
        return self ;
}
```

Semplice e lineare, ma soprattutto sbagliato; lancio infatti l'applicazione in esecuzione, e quello che ottengo è un errore tragico:

```
mc012.app has exited due to signal 10 (SIGBUS).
```

Dove sta il problema?

Sta nel fatto che l'oggetto restituito dall'espressione

```
[ NSMutableArray array ]
```

rimane vivo finche si trova all'interno del metodo `init`. Appena si esce dal metodo, l'oggetto puntato da `listaFile` perde ogni significato, e la triste signora con la falce lo ha eliminato dall'esistenza. Ma quell'oggetto in realtà serve al resto dell'applicazione, e serve vivo. Ecco quindi che aggiungendo un messaggio di `retain`, tutto funziona meravigliosamente:

```
- (id ) init
{
```

```
        self = [ super init ] ;
        listaFile = [ NSMutableArray array ] ;
        [ listaFile retain ];
        return self ;
}
```

Questo meccanismo funziona, ma non è corretto. C'è un metodo migliore per scrivere il tutto, e che risolve anche un ulteriore problema, che illustro con un esempio.

Devo creare un oggetto, diciamo una stringa per fissare le idee: ho un metodo che restituisce un puntatore alla stringa appena creata.

```
- (NSString *) faiStringa
{
        NSString * str ;
        str = [ [ NSMutableString alloc] init ] ;
        // altre operazioni sulla stringa
        return str ;
}
```

Semplice e lineare, ed ancora una volta sbagliato. La stringa è contata in uso una volta di troppo, in quanto il contavita della stringa esce da qui col valore di uno. Dovrebbe invece uscire col valore nullo, perché nessuno (all'uscita) la sta (ancora) usando. Inviare un messaggio esplicito di `release` prima dell'istruzione di return è addirittura peggio, perché al `release` l'oggetto raggiunge un contavita nullo, e viene immediatamente ucciso. Il metodo restituisce un puntatore ad una zona di memoria che contiene spazzatura; l'applicazione, qualche millisecondo dopo, morirà per certo inviando qualche messaggio strano.

Esiste quindi un ulteriore messaggio, chiamato `autorelease`, che ha gli stessi effetti di `release` (diminuisce di uno il contavita), ma lo fa più tardi, quando tutti coloro che avevano bisogno di quell'oggetto hanno finito.

Ora, vi ricordo che un'applicazione con interfaccia utente funziona più o meno così: si parte, si inizializzano le variabili interne, poi si aspetta che succeda qualcosa. Quando succede qualcosa (un evento), si gestisce l'evento, poi si torna ad aspettare. Questo infinito attendere un evento è chiamato il *loop degli eventi*. Il meccanismo degli `autorelease` funziona come un bidone della spazzatura, e la triste signora con la falce diventa più prosaicamente l'addetto della nettezza urbana che passa periodicamente a portare via la spazzatura. All'inizio del loop degli eventi si crea un nuovo bidone vuoto. Poi l'applicazione attende un evento. Quando arriva un evento, sarà eseguita una operazione più o meno corposa di codice, con creazione e distruzione di oggetti. Gli oggetti che devono essere distrutti non lo sono subito, ma sono buttati dentro il bidone, e lì rimangono, ancora vivi, fino alla fine delle operazioni conseguenti all'evento. Alla fine, quando l'evento è stato processato, passa il netturbino e svuota il bidone. Poi, si ricomincia con un nuovo bidone.

ASSEGNARE UNA VARIABILE

Voglio concludere il capitolo con i metodi chiamati **accessor**, ovvero i metodi che permettono di accedere alle variabili d'istanza. Consideriamo un metodo per assegnare il valore ad una variabile d'istanza che sia essa stessa un oggetto. La prima realizzazione

è sbagliata:

```
- (void)setListaFile:(NSMutableArray*)newListaFile
{
        [listaFile release];
        listaFile = [newListaFile retain];
}
```

Eppure, sembra tutto a posto: butto via il vecchio oggetto e mantengo un puntatore al nuovo oggetto. A questo oggetto, per di più, gli dico di sopravvivere perché ne ho appunto bisogno. A prima vista, è corretto; ma consideriamo cosa succede se gli oggetti `listaFile` e `newListaFile` sono in realtà lo stesso oggetto. Prima butto via l'oggetto, poi faccio per prendere quello nuovo, e non c'è più.

Ecco la versione corretta:

```
- (void)setListaFile:(NSMutableArray*)newListaFile
{
        if (listaFile != newListaFile) {
                [listaFile release];
                listaFile = [newListaFile retain];
        }
}
```

Prima di fare qualsiasi cosa, verifico se gli oggetti incriminati sono differenti. Se lo sono, procedo come descritto, altrimenti, non ho nulla da fare!

Utilizzare il metodo accessor è proprio il metodo migliore per gestire l'assegnazione di una variabile d'istanza. Il metodo ha da sé i pregi dell'incapsulamento (non mi preoccupo della struttura dati nemmeno all'interno dell'oggetto stesso!), della comprensione e delle facilità di manutenzione. Ecco quindi la versione definitiva del metodo init:

```
- (id ) init
{
        self = [ super init ] ;
        [self setListaFile: [ NSMutableArray array ]];
        return self ;
}
```

RIASSUNTO FINALE

Riassumendo e ricapitolando questo capitolo piuttosto tecnico ma fondamentale:

- gli oggetti creati tramite alloc o copia sono ritenuti.
- tutti gli altri oggetti sono considerati autorilasciati; scompariranno quindi dall'esistenza al termine del loop degli eventi;
- se un oggetto deve sopravvivere tra due loop degli eventi successivi (se insomma, serve a qualcosa in maniera più o meno permanente), bisogna ritenerlo (si invia un messaggio di `retain`);
- se un oggetto non serve proprio, lo si rilascia esplicitamente con il messaggio di `release`. L'oggetto scompare subito alla vista, senza aspettare la fine del loop degli eventi;

- non occorre mai inviare il messaggio di `dealloc`, in quanto è inviato automaticamente. Occorre però farne una versione specifica per ogni oggetto che abbia costruito oggetti come variabile d'istanza (cioè per ogni oggetto dove il metodo `init` contiene degli `alloc` verso altre classi), per poter rilasciare anche questi oggetti;
- non si deve rilasciare un oggetto se prima non si è ritenuto;
- se un oggetto deve essere utilizzato come valore di ritorno di una funzione o metodo, lo si autorilascia prima di restituirlo. È compito del chiamante ritenerlo (o anche no, se gli serve solo temporaneamente).

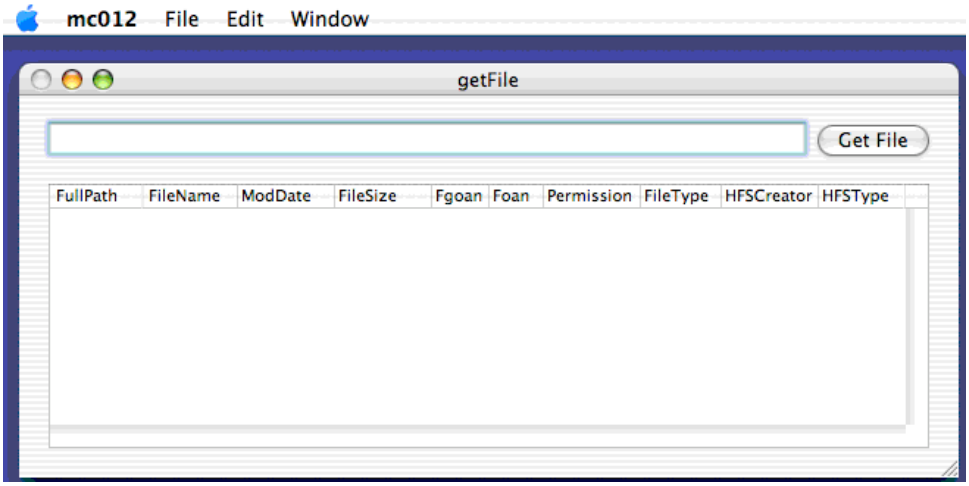
Capitolo 013 - Una tabella di file

Scopo del capitolo è di selezionare un file o una directory, recuperare un po' di informazioni sul file stesso (o sulla directory) ed inserire tutto quanto all'interno di una tabella. Semplice a dirsi e, a parte qualche normale difficoltà, anche a farsi. Cominciamo.

Sorgenti: Documentazione classi

Primo inserimento: 5 Dicembre 2001

L'INTERFACCIA



Velocemente costruisco una interfaccia per fare quanto detto sopra. Sarà scarna e per nulla bella, ma al momento non ci interessa.

Mi occorre un pulsante per far partire le attività, un campo dove mettere il percorso completo del file (serve solo per verifica, visto che so già come lavorare con questo tipo di elementi) ed una tabella. La tabella è un oggetto della classe `NSTableView`, che si può inserire nell'interfaccia trascinando uno degli elementi della palette di IB. Costruisco subito una classe che funzioni da Controllore, la chiamo `FileInfoCtrl`, e genero una istanza dal nome `oFileInfoCtrl`. Collego con una action (`lsGetFile`) il pulsante alla classe controllore, e collego questa al campo di testo (outlet `lastFilePath`). Lascio temporaneamente da parte la tabella.

PESCARE IL NOME DI UN FILE

Per il momento, mi accontento di selezionare un file o una directory con il dialogo standard di apertura file, quello che normalmente si usa quando, dall'interno di una applicazione, si vuole aprire un file.

La classe Cocoa che mi interessa si chiama `NSOpenPanel`. A meno di fare cose piuttosto esotiche, la modalità standard sembra piuttosto semplice. In primo luogo, occorre recuperare un oggetto della classe `NSOpenPanel`. Con l'oggetto sotto mano si possono predisporre alcune caratteristiche d'uso del dialogo (quali file selezionare, quanti, nome

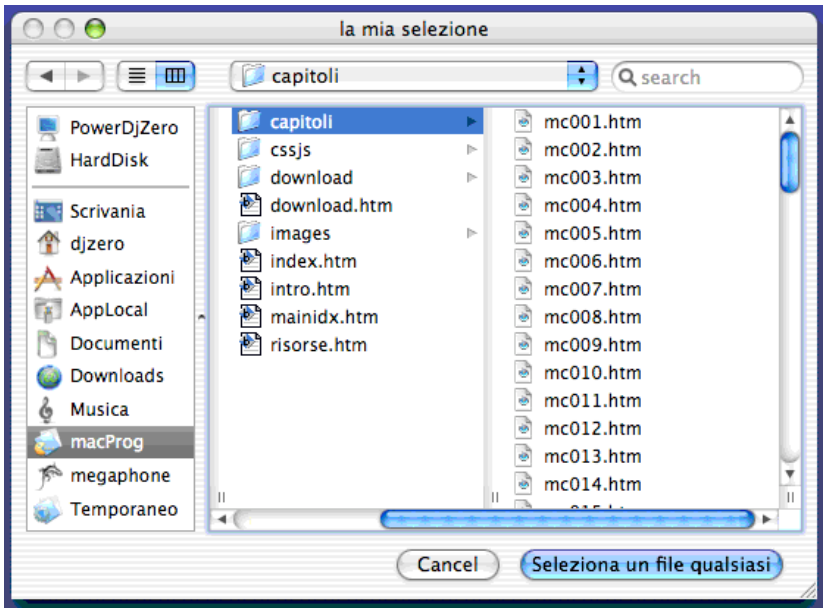
del dialogo, cose del genere), per poi finalmente mostrare all'utente il dialogo, modale. Modale significa che l'utente non può proseguire fino a che non ha effettuato una scelta, o ha rinunciato alla scelta stessa; in altre parole, finché il dialogo è presente a video, l'utente non può interagire con le altre finestre dell'applicazione.

Dalla documentazione della classe `NSOpenPanel` vedo che posso modificare solo quattro proprietà: la possibilità o meno di selezionare file, di selezionare directory, di risolvere o meno gli alias e di permettere o meno la selezione multipla dei file. Non è finita qui: vado a vedere la documentazione della superclasse, che è `NSSavePanel` (bizzarro, ma non troppo: anche il salvataggio di un file è in effetti la selezione di un nome del file, solo che nel salvataggio posso anche scegliere il nome del file...).

Dalla documentazione di `NSSavePanel` trovo interessante la possibilità di modificare altre caratteristiche: il titolo della finestra, il testo del pulsante (il prompt), come trattare i *file packages* (tipo di elementi che ad esempio esemplificano le applicazioni), eccetera.

Potrei risalire la gerarchia delle classi, ma mi limito a capire come fare a modificare le dimensioni della finestra. All'interno di `NSWindow` trovo il metodo `setFrame:display:` che consente di impostare posizione e dimensione della finestra con un `NSRect`. Mi pare bizzarro che Cocoa mi lasci scegliere dove posizionare la finestra: una buona interfaccia utente posiziona la finestra di apertura file al centro della finestra da cui in qualche modo è richiesta. Infatti, a cose fatte, vedrò che non c'è verso di posizionare la finestra come mi piace, anche se per la dimensione qualcosa si può fare.

C'è un ultimo filtro da impostare, ovvero i tipi di file che si possono selezionare, scelti in base alla loro estensione. Poiché al momento non so che file pigliare, decido di poter selezionare qualsiasi file.



Non rimane altro che aprire la finestra di dialogo usando uno dei metodi disponibili. Al mio caso è adatto `runModalForTypes:`, che appunto richiede l'elenco delle estensioni dei file da filtrare: uso la parola chiave `nil` (nulla) che appunto imposta nessun filtro.

Il metodo restituisce un codice, che indica se l'utente ha selezionato il pulsante di scelta predefinito oppure il pulsante di cancellazione della selezione. Nel primo caso si possono

recuperare il file o i file selezionati estraendoli con metodo `filenames`. Il tutto porta alla seguente realizzazione del metodo `lsGetFile`:

```
- (IBAction)lsGetFile:(id)sender
{
    int result;
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];
    /* NSRect NSMakeRect(float x, float y, float w, float h) */
    [oPanel setFrame: NSMakeRect(0, 0, 500, 200) display: NO];
    [oPanel setTitle:@"la mia selezione"];
    [oPanel setPrompt:@"Seleziona un file qualsiasi"];
    [oPanel setCanChooseDirectories:YES];
    [oPanel setCanChooseFiles:YES];
    [oPanel setAllowsMultipleSelection:NO];
    [oPanel setResolvesAliases:NO];

    result = [oPanel runModalForTypes: nil];

    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        NSString *aFile = [filesToOpen objectAtIndex:0];
        [ lastFilePath setStringValue: aFile ];
    }
}
```

Alcune osservazioni.

- Per impostare la dimensione e la posizione della finestra, devo usare un `NSRect`; documentazione al proposito non esiste, allora guardo dei file .h che costituiscono il Foundation Kit, e trovo il file `NSGeometry.h`. Dopo un po' di esplorazioni si trova una funzione di comodo `NSMakeRect` che produce appunto un `NSRect` specificando quattro numeri floating point. I primi due danno le coordinate del punto in alto a sinistra del dialogo, gli altri due la larghezza e l'altezza. Come sospettato, i primi due numeri sono ignorati (io li cambio, e nulla succede all'applicazione). Viceversa, posso modificare larghezza e altezza, anche se rimane fissata la minima e massima dimensione della finestra (ma solo perché sono soggette a modifica attraverso un altro metodo la storia ci porterebbe troppo lontano e al momento non ci interessa).
- il metodo filenames restituisce un oggetto di tipo `NSArray`; quest'oggetto è molto semplicemente un contenitore ordinato di oggetti; gli oggetti si possono recuperare con il metodo `objectAtIndex:`, partendo da Zero per il primo elemento. So già che il file è restituito sotto forma di stringa (anzi `NSString`) completa del path completo, e che c'è un solo elemento in `NSArray` in quanto ho predisposto la selezione singola con il metodo `setAllowsMultipleSelection:`.

L'ultima istruzione del metodo `lsGetFile` si limita a inserire la stringa nel campo testo dell'interfaccia.

INFORMAZIONI DEL FILE

Ora che ho il nome completo del file, voglio recuperare un po' di informazioni sullo stesso (che so, data di modifica, tipo, cose del genere), e metterle da qualche parte.

Cerco nella documentazione, ed il primo oggetto che mi capita sottomano e che fa al mio caso si chiama `NSFileManager`. Proprio nella documentazione c'è un esempio calzante e che ricopio più o meno fedelmente.

Utilizzando il metodo `fileAttributesAtPath:traverseLink:` si possono recuperare, tre le altre, le seguenti informazioni:

- il nome del proprietario e del gruppo di appartenenza;
- la data dell'ultima modifica;
- i permessi relativi al file (lettura, scrittura, modifica);
- la dimensione del file
- il tipo del file

Per conservare tutte queste informazioni relative al file definisco una nuova classe, adatta a contenerle. Ecco il file di intestazione `LSFileInfo.h`:

```
@interface LSFileInfo : NSObject {
    NSString     * fileFullPath ;
    NSString     * fileName ;
    NSDate        * modDate ;
    long         fileSize;
    NSString     * fgoan ;
    NSString     * foan ;
    long         filePosixPerm ;
    NSString     * fileType ;
    long         creatorCode ;
    long         typeCode ;
}

- (void) initWithPath: (NSString*) fullPath ;
- (void) dealloc;
- (NSString*)fileFullPath;
- (void)setFileFullPath:(NSString*)inFileFullPath;

@end
```

Ho aggiunto due variabili, una per contenere il path completo del file, ed una per conservare il solo nome del file. Poi ci sono tutte le variabili che contengono le informazioni sopra citate.

Ho dichiarato anche una lunga teoria di metodi. Il primo metodo serve ad inizializzare le variabili d'istanza con i valori desunti dal file puntato dal path passato come argomento. Ho dichiarato esplicitamente (e quindi lo sovrascrivo) il metodo `dealloc`, che non chiamerò mai esplicitamente, ma che è bene scrivere per eliminare tutti gli oggetti che dipendono da quest'oggetto (tutti gli `NSString` utilizzati come variabili d'istanza) quando si deve eliminare un oggetto di classe `LSFileInfo`.

Si trovano poi elencati a coppie tutta una serie di metodi (e qui ho solo riportato la prima coppia) chiamati *accessor* per accedere alle variabili d'istanza, in lettura e scrittura. Da notare la realizzazione di questi metodi quando la variabile non è un tipo semplice, ma un oggetto; ad esempio nel caso seguente di un oggetto di tipo `NSDate` (destinato a contenere la data di modifica del file):

```
- (void)setModDate:(NSDate*)newModDate
{
    if (modDate != newModDate) {
                [modDate release];
                modDate = [newModDate retain];
        }
}
```

Il metodo tiene conto delle avvertenze sviluppate nel capitolo precedente relativo alla questione di release, retain, eccetera.

Il metodo dealloc merita qualche commento aggiuntivo:

```
- (void) dealloc
{
    [ fileFullPath release ] ;
    [ fileName release ] ;
    [ modDate release ] ;
    [ fgoan release ] ;
    [ foan release ] ;
    [ fileType release ] ;
    [ super dealloc ] ;
}
```

Con questo metodo si rilasciano tutti gli oggetti ritenuti all'interno di `initWithPath:`. Non basta: al termine delle operazioni (e solo al termine!) occorre invocare anche il metodo `dealloc` della superclasse, utilizzando appunto `super`.

Finalmente arriviamo al più interessante metodo di `initWithPath:`, che intende rimpiazzare in tutto e per tutto il metodo standard `init`. Con questo intento, `initWithPath:` è chiamato **inizializzatore designato**, ed è bene chiamarlo ogni volta che si costruisce un oggetto della classe LSFileInfo. La struttura del metodo è molto semplice. La prima cosa da fare è chiamare il metodo `init` della superclasse (in realtà l'operazione è svolta dopo, ma tanto nessuna inizializzazione propria della classe è eseguita fino a quel momento). Poi si costruisce un oggetto della classe `NSFileManager`, che ci serve per l'istruzione successive, che recupera le informazioni del file indicato dal path. Ho passato `NO` come valore del parametro `traverseLink:`; questo significa che se il file puntato è in realtà un link, le informazioni recuperate sono proprio quelle del link e non quelle del file cui il link si riferisce (perché? Perché così mi andava...nessuna ragione precisa). Dopo di che, si assegnano ordinatamente i valori alle variabili d'istanza. Ci sono quelle di facile assegnamento (il path completo, ed il nome del path), quelle che si ricavano direttamente dall'elenco degli attributi del file (la data di modifica, il nome del proprietario), altri infine che bisogna elaborare prima di assegnare il valore alla variabile (tipo e creatore del file secondo la nomenclatura HFS cara al vecchio sistema operativo). Come si può notare, in ogni caso (anche quelli più banali, che meno richiedono l'accorgimento) utilizzo il metodo accessor piuttosto che assegnare direttamente il valore alla variabile d'istanza (lo avessi fatto, avrei dovuto distinguere la variabili oggetto dalle altre, dal momento che per le prime dovevo anche fare una ritenuta dell'oggetto stesso).

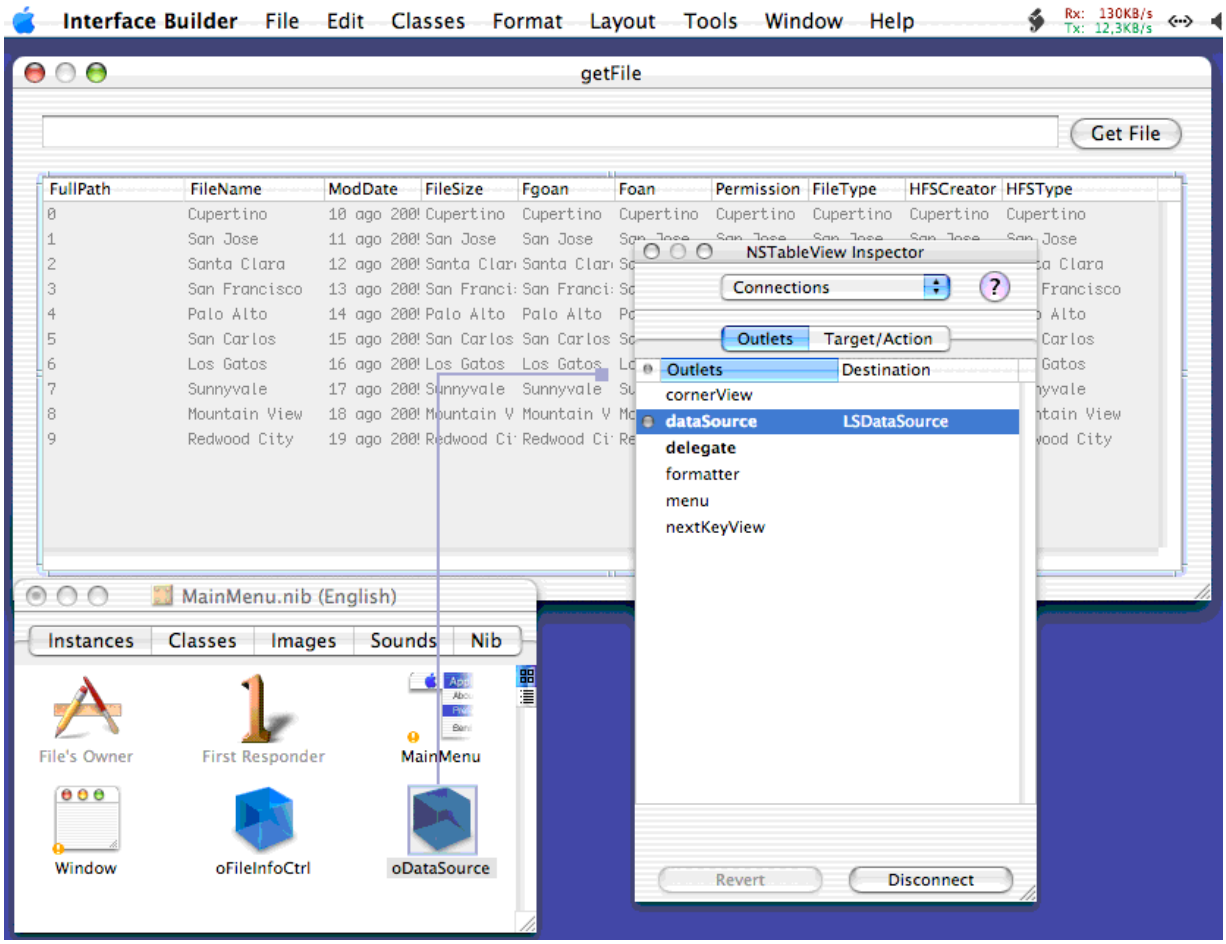
```
- (id) initWithPath: (NSString*) aFile
{
        NSFileManager *manager = [NSFileManager defaultManager];
        NSDictionary *fattrs = [manager fileAttributesAtPath: aFile traverseLink:NO];
        NSNumber *num ;

        [ super init ];
        [ self setFileFullPath: aFile ];
        [ self setFileName: [aFile lastPathComponent]];
        [ self setModDate: [fattrs fileModificationDate] ];
        [ self setFileSize: [ fattrs fileSize ] ];
        [ self setFgoan: [ fattrs fileGroupOwnerAccountName ] ];
        [ self setFoan:[ fattrs fileOwnerAccountName ] ];
        [ self setFilePosixPerm: [ fattrs filePosixPermissions ] ] ;
        [ self setFileType: [ fattrs fileType ] ] ;
        num = [ fattrs objectForKey: NSFileHFSCreatorCode ] ;
```

```
        [ self setCreatorCode: [ num longValue] ];
        num = [ fattrs objectForKey: NSFileHFSTypeCode ] ;
        [ self setTypeCode: [ num longValue] ];
        return ( self );
}
```

Ora che ho capito come costruire un nuovo oggetto con le informazioni relative ad un dato file, mi segno l'istruzione per fare l'intera operazione

```
LSFileInfo * fInfo = [[LSFileInfo alloc] initWithPath: aFile];
```

e mi pongo il problema di come rappresentare l'elenco dei file.



Ho già stabilito di utilizzare un oggetto della classe `NSTableView`. Dalla documentazione leggo che devo utilizzare una classe controllore ausiliaria che fornisca i dati. Torno velocemente su IB, dichiaro una nuova classe (*LSDataSource*), ne faccio una istanza, e la collego alla `NSTableView`. Meglio: collego l'oggetto `NSTableView` dell'interfaccia all'oggetto *oDataSource* (istanza di LSDataSource) utilizzando la connessione `dataSource` che mi si presenta nel dialogo relativo.

Perché l'oggetto controllore `LSDataSource` possa funzionare da sorgente di dati per `NSTableView`, deve realizzare due metodi. Il primo metodo deve dire semplicemente di quante righe è composto l'insieme dei dati da visualizzare. Il secondo metodo è più complesso e merita considerazioni aggiuntive.

Un oggetto `NSTableView` mostra una serie di dati. I dati sono rappresentati uno per

riga, mentre le colonne mostrano i vari attributi di questo dato. Nel nostro caso, il dato è rappresentato dalle informazioni relative ad un file, e le varie colonne contengono i vari campi di informazione relativi.Dal punto di vista costruttivo, una `NSTableView` è in realtà un insieme cooperante di oggetti: c'è un oggetto header (la riga di intestazione) e ci sono tanti oggetti colonna `NSTableColumn`. Selezionando infatti ad una ad una le varie colonne, si possono inserire due informazioni cruciali: il nome della colonna, che sarà mostrato a video, e l'identificatore della colonna, utile per capire quale attributo del dato la colonna contiene (perché si usa un identificatore invece che un più semplice indice numerico? Ma perché le colonne possono essere spostate a piacimento dall'utente, con le solite tecniche del trascinamento... ed allora l'indice cambia, mentre la colonna mantiene il proprio identificatore; ovviamente, si possono utilizzare identificatori numerici, ma tanto sono trattati comunque come stringa...). Lascio che il contenuto della cella abbia una rappresentazione standard, e impostando sempre vari attributi tramite IB, impedisco che l'utente possa fare modifiche al contenuto della cella stessa.

A questo punto, torno ai due metodi che si diceva; il primo metodo da definire è il seguente:

```
- (int) numberOfRowsInTableView: (NSTableView*) tableView
```

che dice quante sono le righe della tabella. Il secondo metodo è più lungo da scrivere:

```
- (id) tableView: (NSTableView*) tableView objectValueForTableColumn: (NSTableColumn *)
tableColumn row: (int) row
```

Questo metodo ha come argomenti una riga (`row`) ed una colonna (appunto identificata come un oggetto `NSTableColumn`) e deve restituire un oggetto che rappresenti il contenuto della cella così individuata.

Bene. La classe `LSDataSource` deve rappresentare un elenco di file; quindi, ha come variabile d'istanza un vettore (di dimensioni variabili) di oggetti. Ecco il file `LSDataSource.h`:

```
@interface LSDataSource : NSObject
{
        NSMutableArray        *listaFile ;
}
- (id) init ;

- (NSMutableArray *) listaFile ;
- (void) setListaFile: (NSMutableArray *) newListaFile ;
- (void) addFileEntry: (id) newEntry ;
- (int) numberOfRowsInTableView: (NSTableView*) tableView ;
- (id) tableView: (NSTableView*) tableView objectValueForTableColumn: (NSTableColumn *)
tableColumn row: (int) row ;
@end
```

La variabile d'istanza è il vettore di oggetti; c'è il metodo `init` (devo inizializzare il vettore) e `dealloc` (devo disfarmi del vettore allocato da `init`), i due metodi sopra discussi e un nuovo metodo, `addFileEntry`, per aggiungere un file nella base di dati. Non parlo dei metodi accessor, che sono banali.

Passiamo alla realizzazione. Il metodo `init` ormai è un classico:

```
- (id ) init
{
    self = [ super init ] ;
    listaFile = [ NSMutableArray array ] ;
    [ listaFile retain ];
    return self ;
}
```

Anche `dealloc` si fa facile:

```
- (void) dealloc
{
    [ listaFile release ] ;
    [ super dealloc ] ;
}
```

Per aggiungere un elemento ad un vettore `NSMutableArray` c'è un metodo apposito, `addObject`, che sfrutto in `addFileEntry:`

```
- (void) addFileEntry: (id) newEntry
{
    [listaFile addObject: newEntry ];
}
```

Sapere quante righe ci sono è facile; basta contare quanti elementi ci sono nel vettore:

```
- (int) numberOfRowsInTableView: (NSTableView*) tableView
{
    return [ listaFile count ];
}
```

Ed adesso, la cosa più difficile: restituire il contenuto di una cella:

```
- (id) tableView: (NSTableView*) tableView objectValueForTableColumn: (NSTableColumn *)
tableColumn row: (int) row
{
    NSString * colId = [ tableColumn identifier] ;
    LSFileInfo *fInfo = [ listaFile objectAtIndex: row ];
    return ( [ fInfo valueForKey: colId ] ) ;
}
```

Tre istruzioni: la prima recupera l'identificatore della colonna. La seconda, l'oggetto corrispondente alla riga indicata. La terza fa tutto il lavoro, ma solo perché sono stato furbo (sì, proprio: ho copiato l'idea dal libro *Learning Cocoa*).

Ho infatti impostato gli identificatori di ogni colonna con lo stesso nome delle variabili d'istanza corrispondenti. In realtà, posso impostare il nome che voglio, ma poi dovrei scrivere una cosa del tipo:

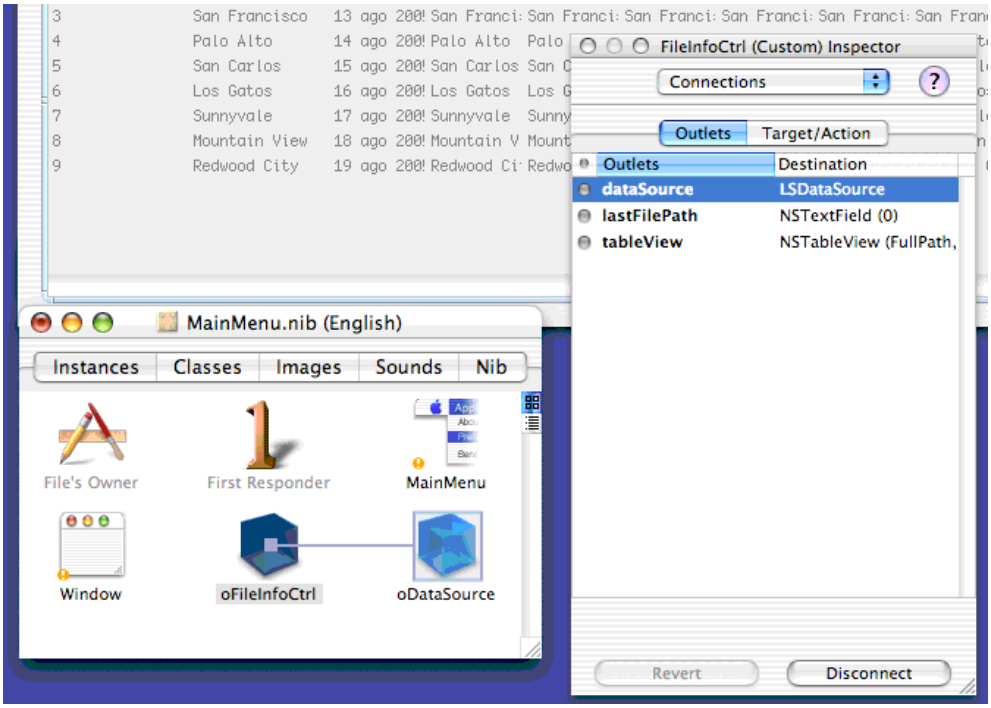
```
if ( [ colId isEqual: @"nome prima colonna")
    valore = [ fInfo <estraggo primo attributo>];
else if ( [ colId isEqual: @"nome seconda colonna")
    valore = [ fInfo <estraggo secondo attributo>];
```

e via così per tutte le possibili colonne. Invece, c'è un trucco. All'interno di un oggetto posso recuperare il valore di una variabile d'istanza se ne conosco il nome (non è certo il metodo più efficiente per farlo, però è mooolto comodo). Ciò si realizza con il metodo `valueForKey:`.

Con questo trucco, sporco e veloce, la terza istruzione completa il metodo.

C'è un ultimo passo. Mettere tutto assieme all'interno del metodo `lsGetfile` nell'oggetto `FileInfoCtrl`. Tralascio la parte di recupero del nome del file, e parto

```
- (IBAction)lsGetFile:(id)sender
{
    ## parte già trattata ##

    result = [oPanel runModalForTypes: nil];

    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        NSString *aFile = [filesToOpen objectAtIndex:0];
        LSFileInfo * fInfo = [[LSFileInfo alloc] initWithPath: aFile];

        [ dataSource addFileEntry: fInfo ];
        [ lastFilePath setStringValue: aFile ];
    }
}
```

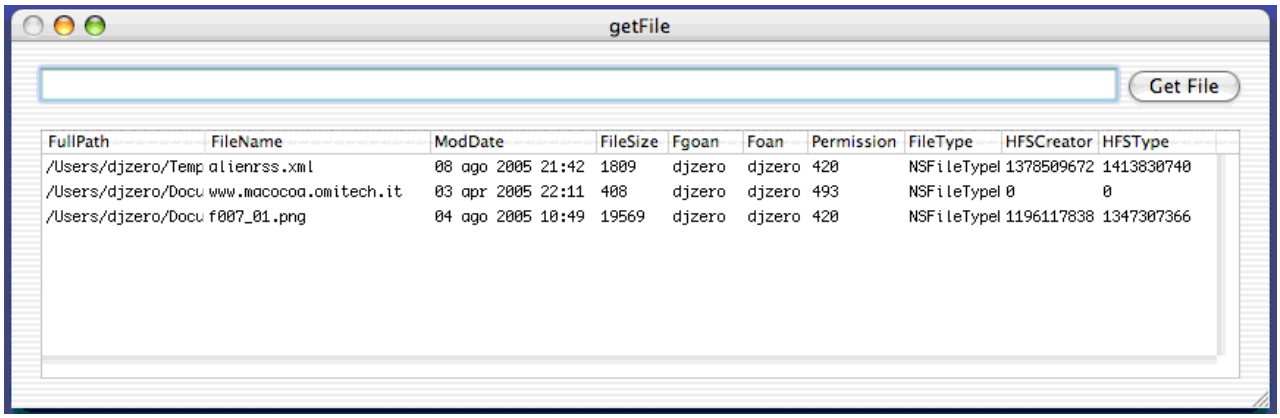


Nel caso ci sia una selezione effettiva (l'utente ha fatto clic sul pulsante di OK), recupero il nome del file, costruisco un nuovo oggetto `fInfo` con tutte le informazioni relative al file selezionato ed aggiungo tale oggetto alla struttura dati della classe `LSDataSource` invocando il metodo `addFileEntry:` sull'outlet `dataSource` che mi sono premunito, ancora in IB, di collegare.

La cosa buffa e divertente è che non funziona. O meglio, funziona, ma bisogna aiutare la finestra. Infatti, nessuno ha detto alla `NSTableView` che i dati sono cambiati, e quindi il contenuto della finestra non viene aggiornato. Non appena sposto o ridimensiono la finestra, ecco che compare il nuovo file aggiunto... .

Rimane quindi da aggiungere in IB un altro outlet che connetta la classe controllore con la `NSTableView`, ed inviare un messaggio di rinfresco dati subito dopo aver inserito un nuovo oggetto:

```
...
[ dataSource addFileEntry: fInfo ];
[ tableView reloadData ];
[ lastFilePath setStringValue: aFile ];
```



Adesso il tutto funziona, anche se sono pieno di dubbi sulle informazioni relative al file (ad esempio, la dimensione del file non ci somiglia per niente...).

Capitolo 014 - L'inizio del Catalogo

Partendo dalla base del capitolo precedente, lo rifaccio cambiando un importante elemento dell'interfaccia grafica, per avvicinarmi di un passo all'idea del catalogatore di Volumi, come annunciato all'inizio di questo percorso. L'occasione è utilizzata anche per focalizzare alcuni concetti passati un po' in sordina nei capitoli precedenti.

Sorgenti: Faccio mio un esempio di Apple, /Developer/Examples/AppKit/OutlineView

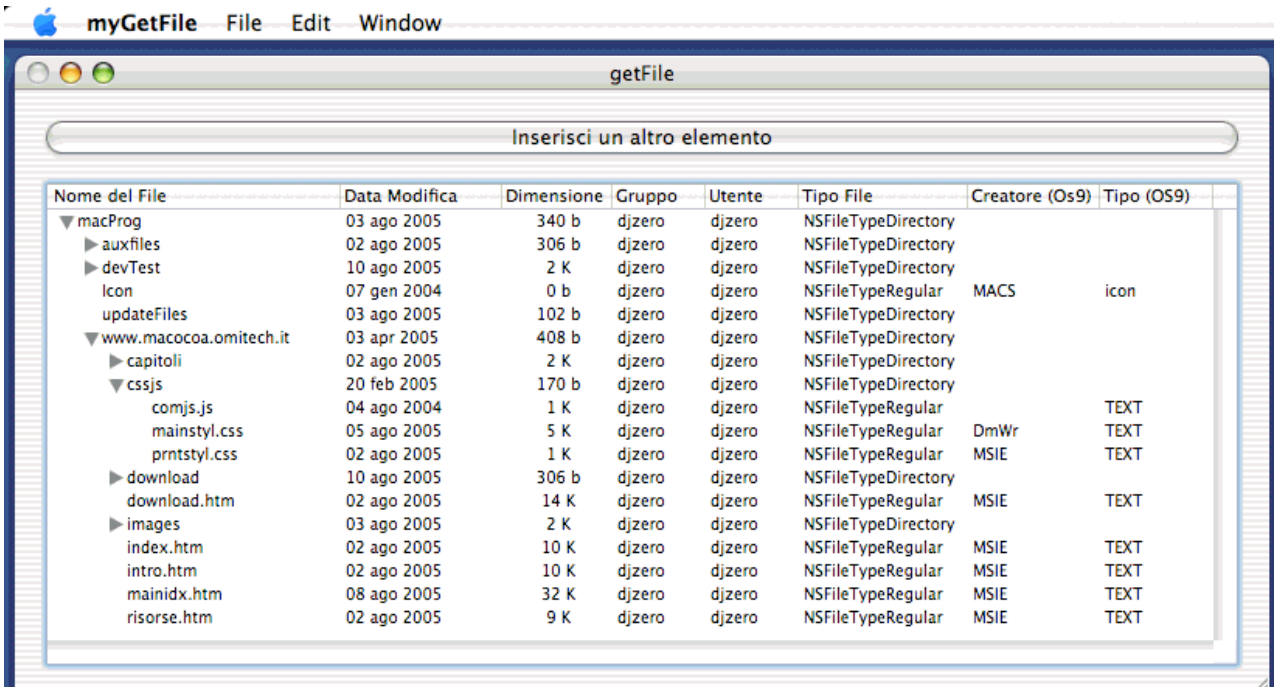
Primo inserimento: 17 Dicembre 2001

L'INTERFACCIA GRAFICA

L'idea di partenza è sempre la stessa: abbiamo una finestra con un pulsante per selezionare un file o una directory, ed un elemento dell'interfaccia dove mostrare le informazioni relative. Ora, esiste un oggetto dell'interfaccia, `NSOutlineView`, che è molto adatto per presentare informazioni in maniera gerarchica. Infatti, `NSOutlineView` è ancora una volta un metodo per esplorare dati in maniera tabellare, ma questa volta è possibile raggruppare una serie di elementi sotto un unico genitore, e contrarre ed espandere questo genitore in modo che mostri o tenga nascosti i propri figli. Ho descritto in modo complicato la vista come lista del Finder...

Insomma, cerco di rappresentare all'interno della mia finestra una collezione di file e directory, dove le directory si possono espandere a mostrare i file in essa contenuti, e così via per le directory contenute nella directory, eccetera. Se la directory è un disco, praticamente è possibile visualizzare l'intero contenuto del disco.

L'utente deve essere in grado di indicare una serie di *punti di partenza*, siano essi file o directory, da inserire all'interno di un *catalogo*. Aggiungendo dischi su dischi, ecco che il catalogatore di Cd è pronto (si fa per dire).



L'interfaccia che disegno in IB è estremamente semplice; consiste nel solito pulsante che scatena le operazioni (l'ho solo allungato a coprire tutta la parte superiore della finestra), e nel campo `NSOutlineView`. Visto che ormai ho capito come gestire le colonne, ho pensato di ridurre la confusione lasciando solo una parte delle colonne definite nel capitolo precedente.

RICORSIONE SULLE DIRECTORY

Torno adesso in XCode e mi occupo di un problema strettamente informatico. Nel capitolo precedente, il dialogo standard di apertura file permetteva di scegliere un file o una directory, e poi le classi sottostanti si limitavano a leggere le informazioni relative all'elemento selezionato. Adesso faccio un passo oltre. Se l'elemento selezionato è una directory, esploro il contenuto della directory stessa. Se uno degli elementi è esso stesso una directory, continuo ad esplorare i file contenuti, e via cosi, fino ad arrivare fino in fondo all'albero delle directory.

Per fare tutto ciò, occorre estendere l'oggetto `LSFileInfo` per aggiungere una serie di informazioni che permettano la costruzione dell'albero. Visto che parlo di oggetti e di classi, sono piacevolmente obbligato a costruire una sottoclasse della classe `LSFileInfo`, che chiamerò `FileStruct`. In questo modo sfrutto le caratteristiche premianti della OOP, ovvero la riusabilità e l'estendibilità dei costrutti pre-esistenti senza richiederne la completa disponibilità (va da sé che adesso ho la piena disponibilità della classe `LSFileInfo`, visto che scrivo tutto io, ma pensate ad un ambiente in cui si lavora in più d'uno...).

Bene, ecco il frammento del file `FileStruct.h` dove dichiaro la nuova classe:

```
#import "LSFileInfo.h"

@interface FileStruct : LSFileInfo {
    FileStruct      * parentDir ;
    NSMutableArray    * fileList ;
}

- (id) initTreeFromPath: (NSString*) fullPath parent: (FileStruct*) parentDir;
- (void) dealloc ;
```

Devo ovviamente includere la dichiarazione della superclasse, poi dichiaro la classe `FileStruct` come sottoclasse di `LSFileInfo`, ed aggiungo un paio di variabili d'istanza. La prima variabile serve a contenere un puntatore alla directory in cui l'elemento è contenuto. Da notare come posso utilizzare subito la dichiarazione di `FileStruct`, appena dichiarata. La seconda variabile serve a contenere, nel caso in cui l'oggetto sia esso stesso una directory, i puntatori ai file contenuti.

Ricordo che, pur non comparendo, sono comunque disponibili tutte le variabili d'istanza della superclasse `LSFileInfo`, anche se, da buon programmatore OO, vi accederò comunque tramite i metodi accessor standard.

Infine, sono dichiarati due metodi. Il primo è l'inizializzatore designato, e serve a costruire l'alberatura delle directory a partire da un path completo. Il secondo è la ridefinizione del metodo `dealloc`, in quanto nell'inizializzazione occorre allocare

oggetti, che dealloc distrugge una volta inutili.

Tutta la difficoltà sta nel metodo `initTreeFromPath: parent:`. Allora lo scrivo qui e poi lo commento:

```
- (id) initTreeFromPath: (NSString*) fullPath parent: (FileStruct*) myParentDir ;
{
    NSFileManager *fileManager = [NSFileManager defaultManager];
    BOOL         isAdir, fileOK ;
    int            i ;

    [ super initWithPath: fullPath ] ;
    [ self setParentDir: myParentDir ] ;
    fileOK = [fileManager fileExistsAtPath:fullPath isDirectory: &isAdir];
    if (fileOK && isAdir)
    {
        NSArray *dirContent = [fileManager directoryContentsAtPath:fullPath];
        int numFile = [dirContent count];
        fileList = [[NSMutableArray alloc] initWithCapacity:numFile];
        for ( i = 0; i < numFile; i++)
        {
            [fileList addObject:[
                [FileStruct alloc]
                    initTreeFromPath:[ fullPath stringByAppendingPathComponent:
                        [dirContent objectAtIndex:i] ]
                    parent:self
                ]
            ];
        }
    }
    else
    {
            fileList = (id) -1 ;
    }
    return ( self );
}
```

La prima cosa che faccio è chiamare l'inizializzatore designato della superclasse, come è giusto. Già che ci sono, metto a posto la variabile d'istanza `parentDir` in cui inserisco appunto il parametro con cui il metodo è stato invocato. Ho già preventivamente definito un `NSFileManager` per accedere al file e capire se è una directory o meno.

Parentesi: mi scapperà più di qualche volta dire file anche dove dovrei dire directory. Non è sbagliato. In Unix qualsiasi cosa è un file. In particolare anche una directory è un file, di tipo speciale, ma un file. Chiusa parentesi.

Se il file non è una directory (attenti, sono nella parte else dell'if), assegno -1 alla variabile `fileList`. Questo è il solito sporco trucco dei programmatori, che utilizzano dei valori speciali all'interno delle variabili per indicare casi strani. Se infatti il file è un vero e proprio file, non si deve fare altro. Però, per indicare che l'oggetto è un file, invece di utilizzare un'altra variabile, o chiedere esplicitamente all'oggetto se è un file o una directory (da qualche parte `LSFileInfo` aveva una variabile che conteneva il tipo del file), metto -1 nel vettore destinato a contenere l'elenco dei file; una volta noto, si fa prima che con altri metodi...

La parte che ragiona con una directory è quella più interessante. Con il metodo `directoryContentsAtPath:` recupero in un vettore tutti i file contenuti nella directory. Conto subito quanti sono con il metodo count e costruisco un vettore (`NSMutableArray`) destinato a contenerli. Poi, iterando sul numero dei file, eseguo

un'unica istruzione che in realtà consiste di una successione di messaggi.

Per prima cosa, recupero il nome del file: come

```
[dirContent objectAtIndex:i]
```

l'oggetto al posto i-esimo nel vettore.

Di questo file mi serve il path completo, che ottengo giustapponendo il path completo della directory che lo contiene con il nome stesso. Questo path

```
[ fullPath stringByAppendingPathComponent: <nome-del-file>]
```

è utilizzato come argomento del metodo di costruzione dell'oggetto FileStruct che alloco e quindi inizializzo:

```
[[FileStruct alloc] initTreeFromPath: <path-completo> parent:self]
```

ottenendo finalmente come risultato un oggetto della classe `FileStruct` che contiene tutte le informazioni necessarie (devo anche passare l'oggetto `self` come argomento).

L'oggetto `FileStruct` risultante è finalmente aggiunto al vettore `fileList` che avevo in precedenza definito.

```
[fileList addObject: <nuovo oggetto>]
```

La struttura ad albero in questo modo si costruisce da sé. Se infatti il messaggio `initTreeFromPath:` è inviato ad una directory, il metodo è *ricorsivamente* chiamato al momento della costruzione della directory stessa, per cui il procedimento si espande da solo fino a coprire tutti i file contenuti all'interno della directory di partenza ed in tutte le directory contenute.

NSOutlineView

Ora che sono in grado di costruire un'alberatura completa a partire da una directory, provo ad inserire il tutto all'interno di una `NSOutlineView`. Il concetto di base è lo stesso visto per la `NSTableView` del passato capitolo. L'oggetto `NSOutlineView` non contiene al suo interno i dati, ma delega tale operazione ad un altro oggetto, definito dall'utente, che deve fornire all'oggetto `NSOutlineView` un insieme minimo di *servizi* per il riempimento della vista.

Nel caso della tabella `NSTableView`, il discorso era piuttosto semplice, in quanto si trattava di definire due soli metodi; il primo per dire quanti elementi erano presenti nella tabella, il secondo per individuare, date riga e colonna della tabella, cosa doveva essere mostrato nella cella della tabella. Qui le cose sono un po' più complicate (ed infatti tra vicissitudini varie ci ho messo parecchio per venirne a capo, o meglio, per avere qualcosa che funziona... non sono tanto sicuro di aver capito tutto), anche a causa del fatto che il numero di righe non è predeterminato (gli elementi si possono espandere e contrarre).

I metodi appunto che una classe adatta a funzionare come sorgente di dati per una `NSOutlineView` sono i seguenti:

```
- (int) outlineView:(NSOutlineView *)outlineView numberOfChildrenOfItem:(id)item;
- (BOOL)outlineView:(NSOutlineView *)outlineView isItemExpandable:(id)item ;
- (id) outlineView:(NSOutlineView *)outlineView child:(int)index ofItem:(id)item ;
-  (id)  outlineView:(NSOutlineView  *)outlineView  objectValueForTableColumn:(NSTableColumn
*)tableColumn byItem:(id)item;
```

Il punto di partenza per capire il funzionamento del meccanismo è il parametro `item`. Questo argomento è l'oggetto vero e proprio che deve essere mostrato in una data riga della tabella. Infatti l'ultimo metodo sopra elencato è l'equivalente di quello utilizzato da `NSTableView`: la differenza sta nel fatto che lì era passato il numero della riga, qui invece, in assenza di un affidabile numero di riga, si dà il riferimento all'oggetto stesso.

Gli altri metodi partono sempre da questo item, e permettono di gestire con comodità il meccanismo di espansione, contrazione, eccetera. Infatti, con il metodo `outlineView:numberOfChildrenOfItem:` l'oggetto `NSOutlineView` è in grado di capire quanti sono i *figli* dell'elemento, e quindi, utilizzando il metodo `outlineView:child:ofItem:` è in grado di esaminarli uno ad uno, e di chiedere direttamente a loro cosa visualizzare in occorrenza di una data colonna (con l'ultimo metodo sopra citato). Per la gestione dell'espansione/contrazione usa il metodo `outlineView:isItemExpandable:`, attraverso il quale si può capire se l'elemento può essere espanso o meno. Nel primo caso, visualizza a lato dell'elemento il classico triangolino dall'orientamento variabile, a seconda se l'elemento è aperto e mostra i figli, oppure è ancora chiuso.

Ora, tutto questo meccanismo è chiaro ed anche piuttosto semplice, ma rimane il problema di far partire tutto il meccanismo (che è appunto il punto dove mi sono bloccato a lungo). L'esempio fornito da Apple funziona, ma sorvola allegramente sulla questione. A differenza dell'esempio, la mia applicazione ha due problemi aggiuntivi. Questi fatti mi complicano la faccenda

- il punto di partenza non è definito all'inizio (per l'esempio è sempre lo stesso);
- non esiste un solo punto di partenza, ma anzi, ne esistono diversi, e selezionabili a piacere da parte dell'utente (ogni volta che l'utente aggiunge un file, si aggiunge un nuovo punto di partenza.

e ne sono venuto fuori con la seguente ipotesi di lavoro (non necessariamente vera, ma visto che funziona...).

`NSOutlineView`, per cominciare, non sa che pesci pigliare, nel senso che non conosce quanti e quali item sono presenti al suo interno. Invia quindi il messaggio corrispondente al metodo `outlineView:numberOfChildrenOfItem:`, dove però l'argomento `item` non è definito, o meglio, è nullo. L'oggetto sorgente di dati interpreta il tutto come la richiesta di quanti elementi di primo livello (i miei punti di partenza) sono contenuti al suo interno. Saputo questo, passa ordinatamente a chiedere gli elementi, individuandoli con il metodo `outlineView:child:ofItem:`; ancora una volta, l'argomento `item` è nullo, per indicare che sta cercando gli elementi di primo livello. A questo punto, la prima visualizzazione della `NSOutlineView` è completata mostrando gli elementi delle varie colonne...

La gestione dell'espansione e contrazione dei vari elementi procede di qui con facilità, visto che adesso è possibile dare dei valori sensati all'argomento `item`. È la stessa `NSOutlineView` a tenere traccia di quali elementi sono espansi, quali contratti, in modo da sgravare il resto del software da questo compito noioso. Quindi, quando è richiesto un'aggiornamento della finestra, `NSOutlineView` ricomincia daccapo (dapprima con gli elementi di primo livello, che ci sono sempre) e poi scendendo man mano sui livelli inferiori, in base alla condizioni di espanso/contratto dei vari elementi.

IL CODICE

Questo discorso si riflette su come sono realizzato i metodi sopra descritti nel mio caso. Ho quindi definito una classe `LSDataSource` (per pigrizia mentale ho utilizzato lo stesso nome della classe del capitolo precedente, ma sono classi del tutto diverse...), che ho collegato all'outlet `dataSource` della `NSOutlineView` direttamente da IB. Il file `LSDataSource.h` risulta qualcosa del genere:

```
@interface LSDataSource : NSObject
{
    NSMutableArray    *startPoint ;
}
- (id) init ;
- (void) dealloc ;

- (NSMutableArray*)startPoint;
- (void)setStartPoint:(NSMutableArray*)newStartPoint;

- (void) addFileEntry: (id) newEntry ;

- (int)outlineView:(NSOutlineView *)outlineView numberOfChildrenOfItem:(id)item;
- (BOOL)outlineView:(NSOutlineView *)outlineView isItemExpandable:(id)item ;
- (id)outlineView:(NSOutlineView *)outlineView child:(int)index ofItem:(id)item ;
-   (id)outlineView:(NSOutlineView   *)outlineView   objectValueForTableColumn:(NSTableColumn
*)tableColumn byItem:(id)item;
-    (BOOL)outlineView:(NSOutlineView    *)outlineView    shouldEditTableColumn:(NSTableColumn
*)tableColumn item:(id)item ;

@end
```

C'è una variabile d'istanza `startPoint` (un vettore) utilizzato per contenere gli oggetti di primo livello; un metodo di `init` e uno di `dealloc`, ai soli scopi della gestione della memoria necessaria al vettore `startPoint`, i metodi accessor per la variabile, un metodo per aggiungere un elemento di primo livello, i quattro metodi che forniscono i dati alla `NSOutlineView`, ed un ultimo metodo di cui parlerò verso la fine del capitolo.

Lascio perdere la definizione (ovvia) dei primi cinque metodi, e mi concentro sui quattro cruciali.

```
// con questo metodo dico quanti figli ha l'item
- (int)outlineView:(NSOutlineView *)outlineView numberOfChildrenOfItem:(id)item
{
    if (item == nil)
        return( [ [self startPoint] count ]);
    return ( [item numOfFiles] ) ;
}
```

Il metodo si commenta da sé: se l'argomento `item` è nullo (si usa la parola chiave `nil` per indicare che si tratta di un puntatore o un oggetto nullo), `NSOutlineView` sta

ragionando sugli elementi di primo livello, e quindi si risponde con il numero di elementi del vettore `startPoint`.

Se invece `item` è un oggetto effettivo, non può che trattarsi di un oggetto della classe `FileStruct`, al quale chiedo di dirmi quanti figli ha; allo scopo ho infatti aggiunto all'interno della classe `FileStruct` un nuovo metodo che aiuta l'incapsulamento dei dati:

```
- (int)numOfFiles
{
    id tmp = [self fileList];
    if (tmp == (id)(-1))
        return ( 0 );
    else return ( [tmp count] ) ;
}
```

Ancora una volta, è molto semplice: recupero l'elenco dei file contenuti. Se il valore è -1, il file con il quale stiamo lavorando è un file vero e proprio, e quindi non ha figli. Restituisco Zero come numero di figli. Se invece è una directory, basta contare quanti elementi sono contenuti nel vettore.

Il passo successivo è il metodo che restituisce i figli degli elementi:

```
// con questo metodo dico qual e' il figlio 'index-esimo' di item
- (id)outlineView:(NSOutlineView *)outlineView child:(int)index ofItem:(id)item
{
    if (item == nil)
        return( [ [self startPoint] objectAtIndex: index ]);
    return ([item getFileAtIndex:index]);
}
```

Se stiamo parlando dell'elemento `nil`, allora si ragiona sugli elementi di primo livello, e quindi si deve restituire l'elemento index-esimo del vettore `startPoint`. Se invece l'elemento è un file (una directory, a questo punto), ho definito un altro metodo di `FileStruct` per facilitarmi il compito, ma che fa essenzialmente la stessa cosa, ma su un altro vettore:

```
- (FileStruct *)getFileAtIndex:(int) elem
{
    return [[self fileList] objectAtIndex:elem];
}
```

Per capire se un elemento è espandibile, ho l'apposito metodo:

```
// con questo metodo dico se item e' espandibile
- (BOOL)outlineView:(NSOutlineView *)outlineView isItemExpandable:(id)item
{
    if (item == nil)
        return ( YES ) ;
    return ([item numOfFiles] != 0);
}
```

Se stiamo parlando dell'elemento `nil`, la risposta non ha importanza; dico di sì giusto pro forma. Altrimenti, nel caso di un elemento file o directory, dico che è espandibile se il numero di figli dell'elemento è diverso da Zero.

Rimane da dire cosa mostrare nelle varie colonne: quasi più semplice del corrispondente metodo del capitolo precedente:

```
// con questo metodo dico cosa deve mostrare item nella colonna indicata
-   (id)outlineView:(NSOutlineView   *)outlineView   objectValueForTableColumn:(NSTableColumn
*)tableColumn byItem:(id)item
{
    NSString * colId ;

    if (item == nil)
        return ( @"???" ) ;
    // recupero l'identificatore della colonna
    colId = [ tableColumn identifier] ;
    return ( [ item valueForKey: colId ] ) ;
}
```

Se l'elemento non esiste, restituisco una stringa a caso. Altrimenti, ricorro allo stesso giochetto dell'identificatore di colonna e `valueForKey:` già visto.

DELEGHE

Rimane da discutere l'ultimo metodo della classe: `outlineView:shouldEditTableColumn:item:`, attraverso il quale si comunica alla `NSOutlineView` se il campo dell'elemento indicato dall'accoppiata item e `NSTableColumn` può essere *editato* oppure no. In effetti, potrei decidere che alcuni *attributi* del file possano essere modificati ed altri no. Per non complicarmi la vita, dico che non si può toccare nulla di quello mostrano dentro la tabella:

```
-    (BOOL)outlineView:(NSOutlineView    *)outlineView    shouldEditTableColumn:(NSTableColumn
*)tableColumn item:(id)item
{
    return NO;
}
```

La questione potrebbe chiudersi qui se non fosse che con questo metodo si introduce il concetto di **classe delegata**. Ci sono infatti alcuni oggetti che, per poter svolgere completamente le proprie funzioni, debbano delegare a qualche altro oggetto alcune funzioni, in quanto non sono in grado di svolgere da soli le operazioni. Decidere se una data cella della tabella possa essere editata o meno non è una questione facilmente risolvibile dalla tabella stessa (`NSOutlineView` è essenzialmente una classe per la realizzazione dell'interfaccia grafica,). Abbiamo cioè un caso speciale del paradigma Model-View-Controller, in cui un oggetto di tipo View richiede esplicitamente un oggetto di tipo Controller per la gestione di alcune funzioni. Si dice allora che l'oggetto View delega una serie di operazioni ad un oggetto delegato. Nel mio caso, dico che l'oggetto delegato (controller) della `NSOutlineView` è lo stesso oggetto che fa da sorgente di dati, ma avrei potuto decidere diversamente. Il fatto che il metodo delegato richiesto sia molto semplice mi ha fatto decidere per una economicità degli oggetti, e di attribuire all'oggetto della classe `LSDataSource` la doppia funzione. Il fatto che normalmente la sorgente di dati e l'oggetto delegato siano concettualmente diversi si può vedere in IB, in cui occorre assegnare esplicitamente, nel modo classico di tracciare un percorso tra i due oggetti tenendo premuto il tasto Control, un collegamento per l'outlet DataSource ed un collegamento per l'outlet Delegate.

COME SI PARTE

Per chiudere, rimane solo da dire come si fa ad aggiungere un elemento di primo livello, ovvero cosa succede quando l'utente fa clic sull'unico pulsante dell'applicazione. Ecco il frammento di codice che riporta le poche modifiche del metodo `lsGetFile`:

```
if (result == NSOKButton) {
    NSArray *filesToOpen = [oPanel filenames];
    NSString *aFile = [filesToOpen objectAtIndex:0];
    FileStruct * fInfo = [[FileStruct alloc] initTreeFromPath: aFile parent: nil];

    [ dataSource addFileEntry: fInfo ];
    [ outlineView reloadData ];
}
```

In risposta ad una scelta dell'utente, si recupera il nome del file come percorso completo, si costruisce un oggetto della classe `FileStruct`, lo si aggiunge alla base di dati e si dice all'oggetto `NSOutlineView` che sono cambiati i dati, e che bisogna quindi rinfrescare la finestra.

Da notare come la costruzione dell'oggetto `fInfo` scateni anche la creazione di tutta l'alberatura delle directory e la raccolta delle informazioni di tutti i file contenuti.

Capitolo 015 - Formattatori

Un capitolo un po' discorsivo all'inizio, ma poi ci sarà un po' d'azione....

Primo inserimento: 19 Dicembre 2001

Celle e Formattatori

Molti degli elementi dell'interfaccia disponibile in IB appartengono alla famiglia dei **Controlli**, sottoclassi della classe `NSControl`. Un Controllo si chiama così perché attraverso di esso l'utente *controlla* l'applicazione: è proprio attraverso i Controlli che si eseguono le operazioni richieste. Un pulsante è un ovvio Controllo, ma anche un campo di testo è tutto sommato un Controllo. In particolare, anche gli elementi argomento dei due capitoli precedenti, `NSTableView` e `NSOutlineView` sono in fin dei conti un Controllo.

Una caratteristica interessante dei controlli è che sono *visibili* all'interno di una finestra; del resto, se non fossero visibili, non si potrebbero utilizzare per interagire con l'applicazione.

Ogni oggetto Controllo utilizza un compagno oscuro per svolgere il lavoro sporco: questo oggetto è una cella, o meglio, è un oggetto di una delle varie sottoclassi di `NSCell`. Sono gli oggetti di tipo `NSCell` i veri responsabili della visualizzazione del contenuto dei vari controlli. Abbiamo, in piccolo, l'applicazione del paradigma Model-View-Controller, dove `NSControl` è appunto la classe Controller, `NSCell` la classe View, mentre la classe Model è compito del programmatore...

Quindi, quanto si interagisce con un pulsante, in realtà le operazioni tipo target/action sono svolte dall'oggetto `NSControl`, mentre la visualizzazione dello stesso (compresi brillamenti ed altre facezie grafiche) è compito della `NSCell`.

Controlli sofisticati contengono al loro interno diverse categorie di celle, adatta a rappresentare diverse unità di informazioni; ad esempio `NSTableView` contiene un elemento cella per ogni colonna, ed ogni cella può essere configurata in modo differente dalle altre. In questo è possibile visualizzare all'interno della `NSTableView` non solo semplici stringhe, ma anche numeri, date, immagini, insomma, quello che si vuole.

Formattatori

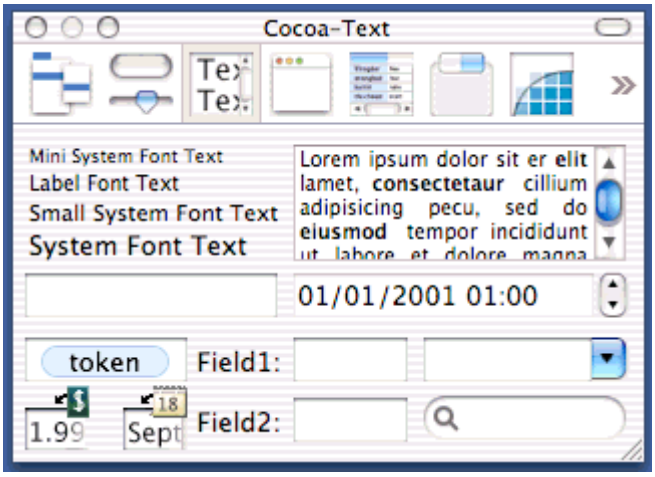
Nella rappresentazione delle informazioni di un file presentata nelle applicazioni dei capitoli precedenti, è piuttosto brutto vedere dimensioni di file espressi in numeri un po' bizzarri, date espresse con formati strani, eccetera. In altre applicazioni potrebbe ad esempio essere necessario visualizzare numeri di telefono, ma non esiste alcuna cella in grado di visualizzare in bella forma questa informazione.

La soluzione a questo problema è la classe `NSFormatter`, una classe astratta che intende raccogliere dei meccanismi di presentazione delle informazioni in bella copia. I

formattatori sono oggetti che traducono il valore di altri oggetti in una rappresentazione più adatta alla visualizzazione; e viceversa, devono cioè essere in grado di passare dalla forma visualizzata in bella copia nella rappresentazione interna.

Esistono una coppia di classi formattatrici direttamente accessibili da IB. Il primo formattatore è adatto alla conversione di numeri, in vari formati; è così possibile rappresentare numeri come dollari, eventualmente in rosso se fossero negativi, trasformare numeri automaticamente in percentuali, cose del genere.

Il secondo formattatore invece consente la rappresentazione di date secondo diverse modalità, scrivendo il giorno della settimana, oppure no, eccetera. In entrambi i casi, esiste anche la possibilità di inventarsi il formato specificandolo attraverso un formato di esempio.



Per utilizzare queste classi formattatici standard, c'è un metodo pratico, direttamente da IB. Si tratta di eseguire l'operazione di drag and drop del formattatore prescelto dalla palette di IB dove sono presenti tutti gli elementi dell'interfaccia, sopra la cella cui il formattatore si applica. Pigliando ad esempio l'applicazione del capitolo precedente, ho utilizzato il formattatore di data per meglio rappresentare la data di modifica del file, trascinando appunto il simbolo della classe `NSDateFormatter` sopra la corretta colonna della `NSOutlineView`.

FARSI UN FORMATTATORE

Ma c'è di più, ovvero è possibile realizzare formattatori a piacimento, semplicemente facendo una sottoclasse della classe madre di tutti i formattatori, ovvero `NSFormatter`. È quello che ho pensato di fare per meglio visualizzare una serie di informazioni riguardanti il file.

In particolare mi interessa rappresentare in maniera più intelligibile la dimensione del file e le due stringhe di **Tipo** e **Creatore** tipiche di ogni file dei sistemi Os9 e precedenti.

Per quanto riguarda la dimensione del file, si tratta semplicemente di trasformare il numero crudo di byte in una stringa più simpatica a vedersi, in cui siano rappresentati K o anche Mega, per maggior leggibilità. Per il tipo ed il creatore, faccio finta che non ci siano funzioni di Carbon/Cocoa che eseguano automaticamente la traduzione (in realtà,

lo ignoro, ma secondo me da qualche parte ci sono), ma mi ricordo che un tipo o creatore è una stringa di quattro caratteri di otto bit, considerata come fosse un intero a 32 bit (sono un po' oscuro, ma forse l'esempio successivo chiarisce tutto).

Ora, per farsi in casa un formattatore, bisogna definire tre metodi: uno per convertire dalla rappresentazione interna alla stringa che sarà visualizzata, uno per ritornare indietro, ed un terzo, il cui uso mi è poco chiaro, ma che dovrebbe servire a capire se sono applicate al testo strane formattazioni (del tipo: il numero è rosso se negativo).

In realtà, poiché intendo solo visualizzare i dati in bella copia, il secondo ed il terzo metodo li realizzo solo per accontentare Cocoa, ma li lascerò bellamente vuoti.

Parto quindi col formattatore per la dimensione del file:

```
@interface FileSizeForm : NSFormatter {

}
- (NSString *)stringForObjectValue:(id)anObject ;
-    (BOOL)getObjectValue:(id    *)anObject    forString:(NSString    *)string
errorDescription:(NSString **)error ;
-      (NSAttributedString      *)attributedStringForObjectValue:(id)anObject
withDefaultAttributes:(NSDictionary *)attributes ;

@end
```

Ho costruito un sottoclasse di `NSFormatter`, non ci sono variabili d'istanza, ma solo i tre metodi citati sopra. Passo alla realizzazione.

```
@implementation FileSizeForm

- (NSString *)stringForObjectValue:(id)anObject
{
    long    fSize, ff ;
    float     fff ;

    if (![anObject isKindOfClass:[NSNumber class]]) {
        return nil;
    }
    fSize = [ anObject longValue ] ;
    // se il file e' piccolo, mostro byte
    if ( fSize < 1024 )
        return ( [ NSString stringWithFormat: @" %8d b", fSize] );
    // la dimensione e' in byte, divido per 1024 ed arrotondo, ottengo K
    ff = (long) (( fSize / 1024.0 ) + 0.5 );
    // se il file e' medio, mostro K
    if ( ff < 1024 )
        return ( [ NSString stringWithFormat: @" %8d K", ff] );
    //in tutti gli altri casi, ritorno Mega, con due cifre decimali
    fff = ( ff / 1024.0 ) ;
    return ( [ NSString stringWithFormat: @" %6.2f M", fff] );
}

-    (BOOL)getObjectValue:(id    *)anObject    forString:(NSString    *)string
errorDescription:(NSString **)error
{
    return ( YES );
}

-      (NSAttributedString      *)attributedStringForObjectValue:(id)anObject
```

```
withDefaultAttributes:(NSDictionary *)attributes
{
    return ( nil);
}
@end
```

Il secondo ed il terzo metodo, come si può vedere, fanno proprio nulla (si limitano a restituire qualcosa di sensato). Il primo metodo è quello interessante. Da notare subito la prima istruzione.

```
if (![anObject isKindOfClass:[NSNumber class]]) {
```

Mi chiedo se l'oggetto che mi è stato chiesto di convertire è un numero. Del resto, la dimensione del file (andate a vedere la classe `LSFileInfo`) è dichiarata essere un long. Mi aspetto quindi che l'oggetto da convertire sia un numero. In ObjectiveC, quando si parla di numeri puri, sono rappresentati come oggetti della classe `NSNumber`. Il numero ricordo che salta fuori dal metodo `outlineView:objectValueForTableColumn:byItem:`, ed in particolare è il risultato dell'istruzione

```
[ item valueForKey: colId ]
```

È ben vero che stiamo parlando di un numero intero, ma questo, in un ambiente OOP, è comunque incapsulato all'interno di un oggetto.

Noto per inciso che l'istruzione `if` mi ha fatto penare non poco. Infatti, senza di essa, l'applicazione muore in maniera spettacolare. Questo è ciò che mi viene risposto:

```
2001-12-17 23:05:59.777 myGetFile[966] *** -[NSCFString longValue]:
selector not recognized
```

Questa criptica frase afferma che nell'esecuzione dell'istruzione

```
fSize = [ anObject longValue ] ;
```

è capitato che `anObject` fosse un oggetto della classe `NSCFString` (credo una stringa). Ignoro chi o cosa sia, e soprattutto da dove salta fuori. Una indagine più approfondita mi ha fatto scoprire che in effetti il formattatore viene invocato una prima volta passando una `NSString` come argomento `anObject`, il cui valore è la misteriosa stringa `Field`. Ancora una volta, ignoro da dove salti fuori questa stringa.

Torniamo a bomba, cioè al metodo convertitore. Una volta deciso che l'oggetto passato parametro è effettivamente un numero, ne ricavo il valore con il metodo `longValue`; a questo punto, è solo C. I commenti incorporati nel codice dovrebbero essere sufficienti a spiegare il procedimento. L'unica cosa da notare è la costruzione *al volo* di una `NSString` utilizzando il metodo `stringWithFormat:`, cosa che permette l'uso, per me molto familiare, della sintassi della funzione `printf` del C standard.

Il passo successivo è il formattatore per il tipo/creatore. Riporto solo il metodo più interessante, per il resto è tutto simile a quanto visto sopra.

```
- (NSString *)stringForObjectValue:(id)anObject
{
    char    ss[20] ;
    union {
        long cc ;
        char ccc[4];
    }    ccu ;

    if (![anObject isKindOfClass:[NSNumber class]]) {
        return nil;
    }

    ccu.cc = [ anObject longValue ] ;
         sprintf(  ss,  "%c%c%c%c",  ccu.ccc[0],  ccu.ccc[1],  ccu.ccc[2],
ccu.ccc[3]);
    return ( [ NSString stringWithCString: ss ] );
}
```

Quando dicevo che il tipo ed il creatore del file sono quattro caratteri rappresentati tramite un intero, ho semplicemente spiegato malamente la union che compare all'inizio. Dopo di ché, è tutto semplice e lineare.

L'ultimo compito da eseguire è di appiccicare il formattatore alla cella che ci interessa.

Ho vagabondato a lungo attraverso la documentazione della varie classi, fino a scoprire quanto ho detto all'inizio. Queste semplici affermazioni si tramutano nelle istruzioni seguenti:

- `NSOutlineView` è un tipo speciale di `NSTableView` (cioè, una sottoclasse).
- `NSTableView` è costruita dall'unione di diverse `NSTableColumn`.
- `NSTableColumn` è solo la classe Controller; c'è bisogno di cella interna per la visualizzazione del tutto.
- i formattatori vanno attaccati alla cella (così che tutta la colonna è rappresenta utilizzando lo stesso formattatore) usando il metodo `setFormatter:`

Queste semplici affermazioni si tramutano nelle istruzioni seguenti:

```
FileSizeForm     *myDF2 = [[ [ FileSizeForm alloc] init ] autorelease ];
[   [[   outlineView   tableColumnWithIdentifier:   @"fileSize"]   dataCell]
setFormatter: myDF2 ];
```

Nella prima istruzione, alloco ed inizializzo un formattatore e lo chiamo `myDF2`. C'è da spiegare il metodo `autorelease`. Con questa istruzione dico che, per quanto riguarda la funzione in cui si trovano queste istruzioni, l'oggetto `myDF2` può essere buttato via quando è finito il loop degli eventi. Infatti, il metodo `setFormatter` esegue un `retain` dell'oggetto formattatore. Quando verrà distrutta (se mai lo sarà) la cella, al formattatore sarà inviato il messaggio release. La famigerata variabile contavita torna a zero, l'oggetto formattatore è distrutto. Se non avessi messo `autorelease`, contavita sarebbe stata già 1 dopo la prima istruzione, diventa 2 per l'incremento causato da `setFormatter:`, ritorna 1 alla distruzione della cella. L'oggetto rimane in memoria, ma nessuno lo sta usando; dovrei esplicitamente distruggerlo contestualmente alla distruzione della cella.

Passiamo alla seconda istruzione, che come al solito è piuttosto pasticciata e la divido in vari pezzi. Per prima cosa, devo recuperare la `NSTableColumn` che mi interessa:

```
[ outlineView tableColumnWithIdentifier: @"fileSize"]
```

Della `NSTableColumn` in realtà mi interessa la cella che lo rappresenta:

```
[ <colonna> dataCell]
```

è proprio a questa cella che appiccico il formattatore:

```
[ <cella> setFormatter: myDF2 ];
```

Sembra complicato, ma poi guardando bene non lo è.

Dove inserire queste istruzioni? Il momento adatto è dopo il caricamento dell'interfaccia utente (o meglio, del file nib che contiene tutti gli elementi dell'interfaccia), ma prima della sua visualizzazione. Sto ovviamente parlando del metodo `awakeFromNib` della classe che controlla l'intera applicazione (`FileInfoCtrl`), che è qui presentato completo dell'installazione di un altro formattatore, condiviso da due colonne (e qui è veramente essenziale `autorelease`...):

```
- (void) awakeFromNib
{
    TOS9TCForm     *myDF1 = [[ [ TOS9TCForm alloc] init ] autorelease ];
    FileSizeForm      *myDF2 = [[ [ FileSizeForm alloc] init ] autorelease
];
      [ [[ outlineView tableColumnWithIdentifier: @"typeCode"] dataCell]
setFormatter: myDF1 ];
     [ [[ outlineView tableColumnWithIdentifier: @"creatorCode"] dataCell]
setFormatter: myDF1 ];
      [ [[ outlineView tableColumnWithIdentifier: @"fileSize"] dataCell]
setFormatter: myDF2 ];
}
```

FILTRO SUI FILE

Per concludere l'intero progetto, ho inserito all'interno della classe `FileStruct` due funzioni aggiuntive scritte in C (funzioni, non metodi! E sì, si possono usare le classiche funzioni in C all'interno di una classe) per evitare di esplorare directory e file non interessanti.

Ho definito due funzioni che rispondono Vero o Falso quando le si sottopone il nome di un file.

Se il file ha un nome che comincia per *punto*, allora è un file che non ci interessa. I file punto in Unix sono normalmente tenuti nascosti e servono per immagazzinare dati anche essenziali ma che non è bello mostrare ad un utente. Per rendervi conto della cosa, aprire un Terminale e digitate `ls -a` o `ls -la` al posto del classico `ls`. Compare in genere qualche file in più il cui nome comincia appunto con un punto.

Inoltre, se il file finisce con `.app`, allora è una applicazione, e non si deve esplorare il suo contenuto. Come dovrebbe essere noto, una applicazione in Mac Os X è in realtà una cartella, la cui struttura è ben definita. All'interno della cartella-applicazione sono

contenute tutte le risorse necessarie al funzionamento dell'applicazione, *help* e *readme* compresi. Dico che nel leggere una directory, se questa è una applicazione, lascio perdere.

Le due funzioni descritte si realizzano facilmente utilizzando due metodi della classe `NSString`:

```
BOOL myFileFilterInsert( NSString * fileName )
{
    if ( [ fileName hasPrefix: @"."] )
        return ( NO );
    return (YES );
}

BOOL myFileFilterExpand( NSString * fileName )
{
    if ( [ fileName hasSuffix: @".app"] )
        return ( NO );
    return (YES );
}
```

per cui il metodo `initTreeFromPath` diventa qualcosa del tipo:

```
- (id) initTreeFromPath: (NSString*) fullPath parent: (FileStruct*)
myParentDir
{
    // mi metto via un file manager
    NSFileManager *fileManager = [NSFileManager defaultManager];
    BOOL    isAdir, fileOK, expandOK ;
    int    i ;

    // per prima cosa, inizializzo super
    [ super initWithPath: fullPath ] ;
    // assegno la directory geniotrice
    [ self setParentDir: myParentDir ] ;
    // e adesso, se il file puntato e' una directory, espando
    fileOK = [fileManager fileExistsAtPath:fullPath isDirectory:&isAdir];
    // ma solo se lo voglio veramente
    expandOK = myFileFilterExpand( [ fullPath lastPathComponent] );
    if (expandOK && fileOK && isAdir)
    {
        // ok, e' una directory, recupero l'elenco dei file
                                NSArray    *dirContent    =    [fileManager
directoryContentsAtPath:fullPath];
        // vedo quanti file ci sono all'interno
        int numFile = [dirContent count];
        // costruisco un vettore destinato a contenere i file e lo assegno
        fileList = [[NSMutableArray alloc] initWithCapacity:numFile];
        // adesso, per ogni elemento, lo aggiungo e costruisco l'albero
        for ( i = 0; i < numFile; i++)
        {
            NSString *myfile = [dirContent objectAtIndex:i] ;
            // ma solo se lo voglio veramente
            if ( myFileFilterInsert( myfile) )
                [fileList addObject:[
                    [FileStruct alloc]
                                                initTreeFromPath:[  fullPath
stringByAppendingPathComponent:
                            myfile] parent:self
                ]
```

```
                ];
        }
    }
    else
    {
        // va beh, e' un file normale
        fileList = (id) -1 ;
    }
    return ( self );
}
```

il metodo contiene ovviamente un errore (logico, e che non inficia il funzionamento dell'applicazione), che al momento non ho idea di come risolvere. Bisognerebbe leggere la documentazione...

Capitolo 016 - Codifica ed Archiviazione

Il passo successivo del progetto di catalogo è di capire come fare a salvare il contenuto della finestra in un file, e come successivamente recuperare le informazioni salvate per inserirle nuovamente all'interno della finestra.

Scopo di questo capitolo è quindi di capire le funzionalità di salvataggio e recupero dati su file. Si parte ancora una volta dall'esempio sviluppato nel capitolo precedente, e lo si modifica per quanto riguarda l'interfaccia utente e le classi interessate.

Sorgenti: Dalla documentazione Apple.

Primo inserimento: 26 Dicembre 2001

ARCHIVIAZIONE

Praticamente ogni applicazione ha bisogno di rendere persistente una serie di dati relativi alle proprie funzionalità. Nel caso del catalogo, è necessario salvare all'interno di un file tutte le informazioni relative ai file che l'utente ha deciso di inserire all'interno della finestra. Se infatti si esce dall'applicazione, al successivo lancio ogni informazione è stata persa.

Cocoa fornisce un meccanismo chiamato **Coding** e **Archiving** per congelare all'interno di un file una rete di oggetti tra loro interagenti, senza per questo perdere le relazioni reciproche. La codifica e l'archiviazione dei dati sono quindi alla base di ogni applicazione che abbia bisogno di immagazzinare da qualche parte i dati di lavoro o altre informazioni necessarie alla propria esecuzione.

Codifica ed archiviazione non sono solo utilizzate nei processi di salvataggio su disco, ma anche, ad esempio, nella trasmissione su rete o su linee seriali in generale (in Unix, ogni cosa è un file: la rete, le linea seriale, eccetera).

La codifica di un oggetto non è altro che la trasformazione di questo oggetto in una rappresentazione seriale, adatta cioè ad essere salvata su file o inviata su un canale seriale. Nell'operazione di codifica sono mantenute le strutture dati, le relazioni con altri oggetti e le informazioni relative alla classe dell'oggetto. L'archiviazione di un oggetto non è altro che la codifica applicata ad un mezzo che possa mantenere nel tempo le informazioni codifica, come ad esempio un file su di un disco rigido.

In Cocoa, si parla di due classi, `NSCoder`, che realizza le operazioni di codifica, e `NSArchiver`, sottoclasse di `NSCoder`, che esegue le operazioni di archiviazione.

Ad esempio, `NSCoder` possiede il metodo `encoderRect:` con il quale è possibile codificare un oggetto della classe `Rect`. In altre parole, si manda ad un oggetto `NSCoder` un messaggio dicendogli: eccoti un rettangolo, vedi di infilarlo nella struttura dati codificati. Ovviamente, `NSCoder` deve eseguire operazioni reversibili, cioè deve possedere anche il metodo opposto `decodeRect:` attraverso il quale è possibile estrarre

dalla struttura dati codificata un rettangolo bello e pronto.

Come si fa a codificare ed archiviare un oggetto di una classe definita dal programmatore? La classe deve aderire al protocollo `NSCoding`.

E per spiegare quest'ultima frase, apro una parentesi.

PROTOCOLLI

Ho a suo tempo discusso come funziona la gerarchia di classi, come si generano nuove classi ereditando variabili e metodi di una superclasse, cose del genere; in particolare, se si vuole aggiungere un metodo ad una classe di cui non si è proprietari, occorre definire una sottoclasse.

I protocolli, formali o informali, servono a dichiarare metodi aggiuntivi ad una classe, metodi che però non sono tipici della classe, ma sono condivisi tra diverse classi.

Un protocollo non è altro che una serie di metodi che una classe deve definire per poter aderire a tale protocollo. Ho già utilizzato, senza dichiararlo esplicitamente, un meccanismo del genere. L'elenco dei metodi che una classe deve fornire perché possa funzionare come sorgente di dati per una `NSOutlineView` è in effetti un protocollo.

NSCODING

Il protocollo `NSCoding` deve essere adattato da tutte quelle classi che intendono avvalersi dei servizi automatici di codifica ed archiviazione. In altre parole, se si vogliono utilizzare i servizi standard di salvataggio degli oggetti su file, ogni oggetto che deve essere salvato deve realizzare due metodi specifici:

```
- (void)encodeWithCoder:(NSCoder *)encoder ;
- (id)initWithCoder:(NSCoder *)decoder ;
```

Il primo metodo serve per codificare ed archiviare l'oggetto (scriverlo su una base di dati serializzata), il secondo per decodificare l'oggetto (crearlo in base ai dati estratti).

Quando si esegue `encodeWithCoder:`, l'oggetto deve salvare le proprie variabili d'istanza, codificandole attraverso l'oggetto encoder della classe `NSCoder`. Pigliamo ad esempio un oggetto `LSFileInfo`; la definizione del metodo è la seguente:

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    [ encoder encodeObject: [ self fileFullPath ] ] ;
    [ encoder encodeObject: [ self fileName ] ] ;
    [ encoder encodeObject: [ self modDate ] ] ;
    [ encoder encodeObject: [ self fgoan ] ] ;
    [ encoder encodeObject: [ self foan ] ] ;
    [ encoder encodeObject: [ self fileType ] ] ;
    [ encoder encodeValueOfObjCType: @encode(long) at: & fileSize];
    [ encoder encodeValueOfObjCType: @encode(long) at: & filePosixPerm];
    [ encoder encodeValueOfObjCType: @encode(long) at: & creatorCode];
    [ encoder encodeValueOfObjCType: @encode(long) at: & typeCode];
}
```

Normalmente, ogni oggetto comincia il metodo invocando `encodeWithCoder:` per la superclasse. Qui non accade perché `LSFileInfo` è un discendente diretto di `NSObject`, e quindi non ci sono altre variabili d'istanza da salvare. Il metodo poi procede a codificare (in un certo ordine, arbitrario ma determinato) tutte le variabili d'istanza (almeno, quello che sono necessarie alla piena funzionalità dell'oggetto). Poiché molte delle variabili d'istanza sono a loro volta degli oggetti, le prime sei righe codificano direttamente questi oggetti invocando il metodo `encodeObject:`; il polimorfismo degli oggetti stessi fa poi in modo che per ogni oggetto sia utilizzato il codificatore più appropriato per la classe.

Le ultime quattro righe sono invece un accorgimento per salvare in maniera uniforme anche le variabili d'istanza che oggetti non sono. Allo scopo si usa il metodo `encodeValueOfObjCType:at:`. Il primo argomento del metodo deve specificare il tipo del dato che si intende codificare. Utilizzo la direttiva al compilatore `@encode(.)` perché si preoccupi lui stesso di produrre la rappresentazione appropriata. Il secondo argomento è un puntatore al dato che interessa codificare.

Questo metodo deve andare di pari passo con il metodo estrattore. Anzi, se i due metodi non sono uno lo specchio dell'altro, il meccanismo non funziona. Qui è importante che l'ordine di estrazione dei dati sia lo stesso utilizzato per la codifica:

```
- (id)initWithCoder:(NSCoder *)decoder
{
    [ self setFileFullPath: [ decoder decodeObject ] ];
    [ self setFileName: [ decoder decodeObject ] ];
    [ self setModDate: [ decoder decodeObject ] ];
    [ self setFgoan: [ decoder decodeObject ] ];
    [ self setFoan: [ decoder decodeObject ] ];
    [ self setFileType: [ decoder decodeObject ] ];
    [ decoder decodeValueOfObjCType: @encode(long) at: & fileSize];
    [ decoder decodeValueOfObjCType: @encode(long) at: & filePosixPerm];
    [ decoder decodeValueOfObjCType: @encode(long) at: & creatorCode];
    [ decoder decodeValueOfObjCType: @encode(long) at: & typeCode];
    return self ;
}
```

Come si può vedere, i due metodi sono molto simili, esattamente speculari.

In realtà, gli oggetti da archiviare all'interno del catalogo non sono di tipo `LSFileInfo`, ma di tipo `FileStruct`. Ecco quindi la realizzazione del protocollo per questa classe:

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    int dim ;
    // salvo la variabili ereditate
    [ super encodeWithCoder: encoder ];
    // vedo quanti file ci sono
    dim = [ self numOfFiles] ;
    // metto da parte questo numero
    [ encoder encodeValueOfObjCType: @encode(int) at: & dim];
    // se ci sono file dipendenti, li salvo tutti
    if ( dim > 0 )
        [ encoder encodeObject: [ self fileList ]];
}

- (id)initWithCoder:(NSCoder *)decoder
```

```
{
    int dim ;
    // recupero le variabili ereditate
    [ super initWithCoder: decoder ];
    // vedo se ci sono file dipendenti
    [ decoder decodeValueOfObjCType: @encode(int) at: & dim];
    if ( dim <= 0 )
        [ self setFileList: nil ]; // non ce ne sono
        // recupero le info di tutti i file contenuti
    else [ self setFileList: [ decoder decodeObject ]];
    return self ;
}
```

Qui c'è un problema aggiuntivo, dovuto al fatto che un oggetto `FileStruct` contiene al suo interno un vettore (`fileList`) di altri oggetti `FileStruct`. Ora, la variabile `fileList` è un oggetto della classe `NSMutableArray`; basta dare un'occhiata alla documentazione relativa per notare che è conforme al protocollo `NSCoding`. Questo significa che non devo essere io direttamente a preoccuparmi di esaminare tutti gli oggetti presenti nel vettore e dire a ciascuno di archiviarsi, ma è sufficiente invocare il messaggio `encodeObject:` con argomento il vettore stesso. Poiché la classe aderisce al protocollo, sia occupa lei stessa di salvare il contenuto del vettore, in maniera molto semplice ed efficace per il programmatore. Ho in ogni caso evitato di codificare il vettore se il vettore in realtà non è presente; è questo il motivo del calcolo della dimensione del vettore stesso prima di effettuarne la codifica (perché tutto ciò funzioni, ho dovuto modificare la classe `FileStruct`, ma ne parlerò dopo...).

Punto di partenza

Rimane da discutere come scatenare tutto il meccanismo di salvataggio/ripristino dei dati. `NSCoder` è una classe astratta; per fare qualcosa di serio, occorre rivolgersi ad una sua sottoclasse. Per quanto riguarda i file su disco, le classi che mi interessano si chiamano `NSArchiver` e `NSUnarchiver`.

L'idea di base è che si archivia un solo oggetto (root) per file. Tuttavia, oggetti dipendenti da questo oggetto root (perché immagazzinati come variabili d'istanza) sono ugualmente salvati all'interno del file. In questo modo, è possibile salvare in un file un intero grafo di oggetti interconnessi. Nel caso del catalogo, la cosa è semplice: l'oggetto root che mi interessa è un `NSMutableArray`, (è la variabile `startPoint` dell'oggetto `LSDataSource`), che, per sua natura, da luogo a tanti alberi di file quanti sono i punti di partenza inserito dall'utente.

Questo fatto mi risparmia un problema: non sempre nel grafo degli oggetti interconnessi gli oggetti sono presenti una volta sola; ad esempio, potrebbero esserci oggetti riferiti in altri oggetti, che a loro volta tornano indietro...cose del genere (in effetti, i più accorti di voi si saranno accorti che nel codice sopra presentato è scomparsa la variabile d'istanza `parentDir` della classe `FileStruct`, che puntava proprio all'indietro...ma anche di questo ne parlo poi). La cosa è risolta direttamente da `NSArchiver`: questa classe tiene conto di tutti gli oggetti incontrati nel processo di attraversamento del grafo di oggetti. La prima volta che un oggetto viene incontrato, è codificato normalmente; incontri successivi con lo stesso oggetto portano non ad una nuova codifica, ma all'inserimento di un riferimento alla codifica precedente.

A questo punto, per salvare un grafo basta archiviare un oggetto root:

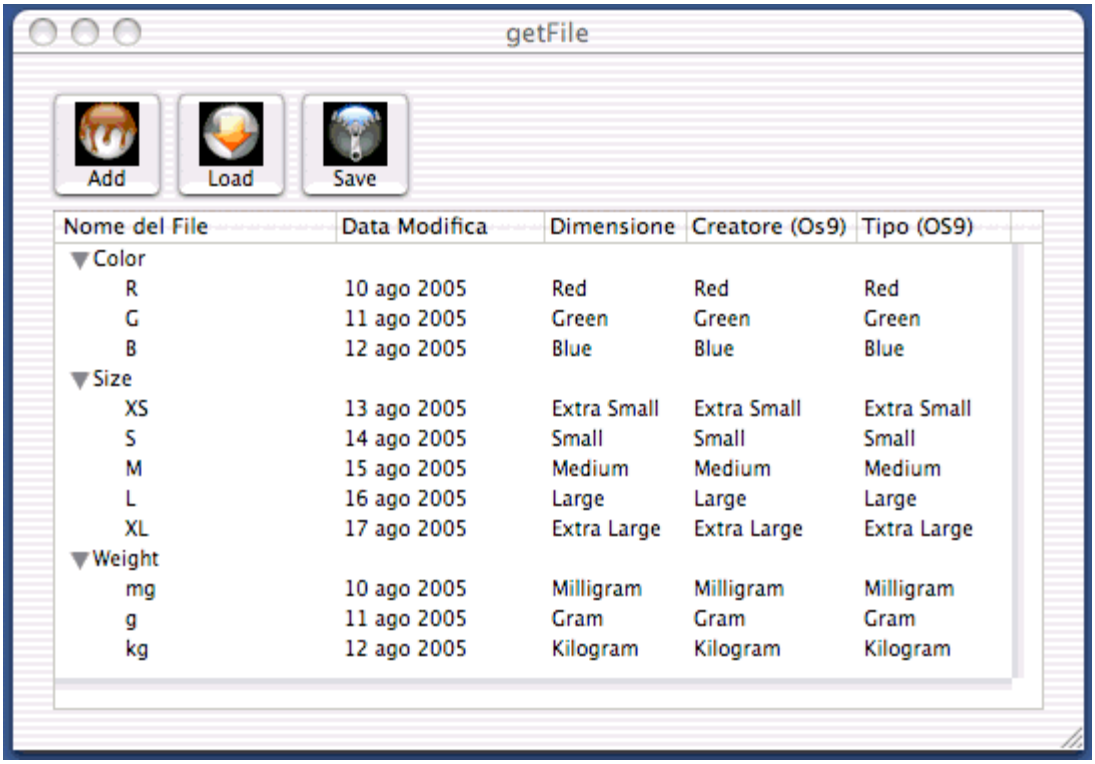
```
[NSArchiver archiveRootObject: mioOggetto toFile: miofile];
```

e per recuperarlo dal file basta assegnare un oggetto root:

```
mioNuovoOggetto = [ NSUnarchiver unarchiveObjectWithFile: miofile];
```

Prima di attaccare queste istruzioni nel punto giusto dell'applicazione, bisogna modificare l'interfaccia.

L'INTERFACCIA



Ho modificato l'interfaccia dell'applicazione per tenere conto delle nuove possibilità. Via il pulsantone per aggiungere un file, ho deciso per tre pulsanti quadrati, uno per aggiungere file al catalogo, uno per caricare il catalogo salvato su disco, uno per salvare il catalogo su disco. Devo aggiungere una coppia di target/action, ovvero un paio di metodi per la classe controllore, una per effettuare il salvataggio e l'altra per il recupero. Semplifico ulteriormente la `NSOutlineView` riducendo ancora una volta il numero delle colonne. Da apprezzare il fatto che non cambia la classe `LSDataSource`.

SALVATAGGIO E RECUPERO

Diventa così possibile concludere la realizzazione delle procedure di salvataggio e di recupero da file dei dati. Per salvare su file, in risposta al clic sul pulsante Save il metodo è il seguente:

```
- (IBAction)save2File:(id)sender
{
    int        risposta ;
    NSSavePanel    *sPanel = [[ NSSavePanel savePanel ]autorelease ];
    // personalizzo il pannello di salvataggio
    [ sPanel setTitle:@"Salva Catalogo" ];
    [ sPanel setPrompt:@"Determina un file di catalogo" ];
    // i file salvati avranno l'estensione lscat
    [ sPanel setRequiredFileType: @"lscat" ];

    risposta = [ sPanel runModal ] ;
    if ( risposta == NSOKButton )
    {
        // recupero il nome del file
        NSString *aFile = [ sPanel filename ] ;
        // archivio l'intera struttura dati
            [NSArchiver archiveRootObject: [dataSource startPoint] toFile:
aFile ];
    }
}
```

Qui, oltre alla chiamata finale a `NSArchiver`, c'è l'apertura di un dialogo modale di salvataggio file. C'è poca differenza rispetto a quanto visto nei capitoli precedenti relativi all'apertura di un file. Da notare il metodo `setRequiredFileType:` che gestisce l'estensione dei file prodotti. Se l'estensione non è presente o è diversa da `lscat`, aggiunge `.lscat` al nome del file. Inoltre, poiché c'è un solo nome di file possibile come risposta (quello immesso dall'utente), il metodo utilizzato è `filename`, e restituisce direttamente il path del file.

Per recuperare i dati dal file, bisogna prima richiedere il file, e poi procedere all'operazione:

```
- (IBAction)loadFromFile:(id)sender
{
    NSOpenPanel    *oPanel = [ NSOpenPanel openPanel ] ;
    int        risposta ;
    // imposto un po' di caratteristiche del dialogo
    [ oPanel setTitle:@"Apri Catalogo" ];
    [ oPanel setPrompt:@"Seleziona un file di catalogo" ];
    [ oPanel setCanChooseDirectories:NO ];
    [ oPanel setCanChooseFiles:YES ];
    [ oPanel setAllowsMultipleSelection:NO ];
    [ oPanel setResolvesAliases:YES ];
    // posso selezionare solo i file che hanno estensione lscat
      risposta = [ oPanel runModalForTypes: [ NSArray arrayWithObjects:
@"lscat" ] ];

    if ( risposta == NSOKButton )
    {
        // recupero il nome del file selezionato
        NSArray *filesToOpen = [ oPanel filenames ];
        NSString *aFile = [ filesToOpen objectAtIndex: 0 ];
        // recupero l'intera struttura dati dal file
           NSMutableArray * data = [NSUnarchiver unarchiveObjectWithFile:
aFile ];
        // assegno il tutto alla sorgente di dati
        [ dataSource setStartPoint: data ] ;
        // dico che sono cambiate le cose
        [ outlineView reloadData ];
```

```
    }
}
```

Oltre ad un po' di istruzioni per la personalizzazione del dialogo, da notare come adesso accetto di aprire solo i file aventi l'estensione `lscat`. Alla fine poi, dopo aver recuperato i dati con `NSUnarchiver`, assegno i dati come `startPoint` dell'oggetto `dataSource`, senza dimenticare di inviare l'invito di rinfrescare i dati a `NSOutlineView`.

MODIFICHE E CAMBIAMENTI

Ho eseguito alcune modifiche sul vecchio codice, in modo da renderlo più rispondente alle nuove esigenze. La modifica più importante riguarda il codice sentinella (-1) che avevo assegnato alla variabile d'istanza `fileList` degli oggetti `FileStruct`, per indicare che il file era effettivo e non una directory. Dopo aver penato un intero pomeriggio con applicazioni che morivano allegramente con errori vari, mi sono reso conto di inviare messaggi a `fileList` senza controllare che la variabile fosse un effettivo oggetto e non il numero -1. Piuttosto che filtrare questa circostanza (cosa piuttosto noiosa), approfitto del fatto che si possono inviare tutti i messaggi del mondo ad un oggetto di tipo nil (cioè, assolutamente vuoto e nullo), senza che questo si arrabbi. Contestualmente, ho anche modificato il metodo `initTreeFromPath:` per correggere l'errore segnalato alla fine del capitolo precedente. Il problema è che la variabile `fileList` è inizializzata subito con una dimensione data dal numero di file contenuti nella directory; in realtà, nel ciclo for successivo, alcuni file sono esclusi da conteggio, per cui spesso il vettore non è completamente riempito. Ciò non inficia la funzionalità dell'applicazione (il metodo `count` continua a funzionare correttamente), ma c'è un certo spreco di spazio. Ho deciso quindi di mettere i dati in una variabile temporanea, e solo alla fine assegnare il vettore della giusta dimensione a `fileList`. Il codice spiega tutto:

```
- (id) initTreeFromPath: (NSString*) fullPath
{
    ++++tutto come prima++++
    if (expandOK && fileOK && isAdir)
    {
        // ok, e' una directory, recupero l'elenco dei file
            NSArray *dirContent = [ fileManager directoryContentsAtPath:
fullPath ];
        // vedo quanti file ci sono all'interno
        int numFile = [ dirContent count ];
        // costruisco un vettore destinato a contenere i file e lo assegno
              NSMutableArray *tmpfileList = [[ NSMutableArray alloc ]
initWithCapacity: numFile ];
        // adesso, per ogni elemento, lo aggiungo e costruisco l'albero
        for ( i = 0; i < numFile; i++)
        {
            NSString *myfile = [ dirContent objectAtIndex: i ] ;
            // ma solo se lo voglio veramente
            if ( myFileFilterInsert( myfile) )
                [tmpfileList addObject:[
                    [FileStruct alloc]
                                                initTreeFromPath:[ fullPath
stringByAppendingPathComponent:
                                myfile] ]
                ];
```

```
        }
        // adesso copio il vettore con la giusta dimensione
              fileList = [[ NSMutableArray alloc ] initWithCapacity: [
tmpfileList count] ];
        [ fileList setArray: tmpfileList ];
    }
    else
    {
        // va beh, e' un file normale
        fileList = nil ;
    }
    return ( self );
}
```

La modifica invece del valore sentinella da -1 a nil ha generato altre modifiche; la più importante riguarda il metodo `numOfFiles`, adesso molto più semplice:

```
- (int)numOfFiles
{
    return ( [[self fileList] count] ) ;
}
```

Qui, anche se `fileList` è `nil`, il metodo restituisce correttamente 0 come dimensione del vettore.

Infine, ho eliminato la variabile d'istanza `parentDir`. Ciò è avvenuto in un punto oscuro della storia di questo capitolo, quando non funzionava nulla. Ho pensato che i miei problemi derivassero dal fatto che archiviare l'oggetto `parentDir` una seconda volta fosse un problema; in realtà, come poi ho scoperto leggendo la documentazione, non c'è problema. Tuttavia, ho notato che la variabile non serviva ad alcuno scopo, ed infatti tutto funziona anche senza di essa. E questo la condanna all'eliminazione (salvo poi scoprire, come credo succederà tra poco, che servirà a qualcosa di utile e fondamentale... è la storia della mia vita, fare, disfare, rifare, mi sento un po' Penelope).

Capitolo 017 - Documenti

Adesso che so come leggere ed aggiungere file ad un controllo `NSOutlineView`, leggere e salvare i dati su di un file, mi accingo a compiere un passo molto importante.

L'argomento di questo capitolo è infatti l'architettura di una applicazione basata su documenti, e quindi in grado di visualizzare il contenuto di uno o più documenti all'interno di una serie di finestre, tutti assieme all'interno della stessa applicazione.

Tre sono le Classi con cui ho a che fare: `NSDocument`, `NSDocumentController` e `NSWindowController`. Ma comincio dall'inizio, ovvero dal misterioso e inutile file `main.m`.

Sorgenti: Sfrutto il capitolo 11 del libro *Learning Cocoa*.

Primo inserimento: Gennaio 2002 (?)

NSApplication

Il file contiene una unica istruzione che si limita a chiamare una funzione.

```
return NSApplicationMain(argc, argv);
```

`NSApplicationMain` è una funzione di utilità fornita dal framework, che costruisce un oggetto della classe `NSApplication`, carica il file nib indicato come principale (lo si indica nel pannello `Application Setting` quando si selezionano le opzioni dei `Targets` dentro XCode), tipicamente chiamato `mainMenu.nib`, e fa partire il loop, teoricamente infinito, degli eventi.

Ogni applicazione contiene uno ed un solo oggetto della classe `NSApplication`. L'oggetto incapsula l'intera applicazione (anche dal punto di vista del Finder e delle altre applicazioni) e funziona come unico punto di interfaccia verso il mondo esterno. Alcuni attributi (le variabili d'istanza) interessanti dell'oggetto `NSApplication` sono ad esempio l'icona, il già citato file nib principale, l'elenco delle finestre delle applicazioni, quale delle finestre è quella principale (davanti a tutte le altre), quale finestra riceve i caratteri inviati dalla tastiera (che può essere diversa dalla principale). Una variabile d'istanza importante è la classe delegata. Ricordo che una classe delegata aiuta la classe principale quando devono essere prese alcune decisioni sulle quali classe principale ha delle difficoltà. Un esempio è la decisione di uscire dall'applicazione. L'oggetto `NSApplication`, quando l'utente seleziona la voce di menu `Esci` o `Quit` non può decidere da sola se terminare; passa la palla alla classe delegate, che si preoccupa di verificare se tutti i documenti sono stati salvati, o se tale salvataggio interessa l'utente. La classe delegata può addirittura sospendere il procedimento di terminazione (ad esempio, perché l'utente ha deciso di rispondere `Annulla` ad uno dei dialoghi di salvataggio dei file).

Architettura

La struttura a documenti di una applicazione nasce a partire da un controllore di documenti. C'è un unico (credo...) oggetto della classe `NSDocumentController` che funziona da gestore dei documenti. È questo oggetto che risponde in prima istanza ai menu `Nuovo`, `Apri`, `Salva`. L'oggetto `NSDocumentController` gestisce quindi uno o più oggetti `NSDocument`, uno per ogni documento aperto o creato. Gli oggetti `NSDocument` sono responsabili della gestione del contenuto del documento, e di come questo contenuto è visualizzato dall'applicazione. Quindi, ogni oggetto `NSDocument` deve gestire una o più finestre in cui il documento è rappresentato a video. Comanda quindi su una serie di oggetti `NSWindowController`, uno per ogni finestra appartenente al documento. `NSWindowController` è la classe controllore della finestra, che contiene una serie di controlli, viste, eccetera.

NSDocumentController

Il compito principale dell'oggetto `NSDocumentController` è di creare o aprire documenti e di gestirli all'interno dell'applicazione. Possiede una lista di documenti aperti, e sorveglia particolarmente il documento attivo, quello relativo alla finestra principale.

È questo oggetto a rispondere ai comandi fondamentali da menu: apri, nuovo, salva, stampa, cose del genere. Ad esempio, quando l'utente seleziona `Nuovo` (o simile) da menu, è inviato un messaggio a `NSDocumentController`. Questo messaggio fa sì che venga costruito un nuovo documento (un oggetto `NSDocument` o sottoclasse) del tipo gestito dall'applicazione (l'elenco dei tipi si può dare sempre tramite il pannello `Application Setting` in XCode), e gli manda il messaggio `init` perché esegua le inizializzazioni base. Se invece la voce di menu scelta è `Apri`, si preoccupa di aprire il solito panello di apertura file, filtrando opportunamente i file secondo i tipi indicati (quelli di prima...). In base al tipo di file scelto, costruisce l'appropriato oggetto `NSDocument`; questa volta, invece di `init` gli manda il messaggio `initWithContentOfFile`, in modo da costringere la lettura del contenuto del file. Questo meccanismo ha una implicazione nella procedura di salvataggio e apertura file, molto più facile della realizzazione precedente.

NSDocument

`NSDocument` è una classe astratta, ovvero, così com'è non funziona. Devo sempre fare una sottoclasse di `NSDocument` perché funzioni come modello del mio tipo di documento. Questo significa che devo scrivere (anzi, sovrascrivere) una serie di metodi che facciamo il lavoro base di gestione dei documenti.

Il compito principale di un oggetto `NSDocument` è di rappresentare, manipolare, immagazzinare e caricare i dati persistenti associati ad un documento (insomma, gestisce un file su disco). Le operazioni che deve essere in grado di svolgere sono quindi essenzialmente di fornire i dati contenuti nel documento in una delle varie rappresentazioni richieste dagli altri oggetti dell'applicazione (in particolare, agli oggetti view contenuti nelle finestre), caricare i dati dal file in strutture interne e mostrarle a video, immagazzinare le informazioni in un file specificato. L'oggetto è il destinatario ultimo per salvare, stampare e chiudere documenti. È qui che si realizzano le operazioni

di stampa, di Undo, è qui dove si traccia lo stato del documento (se è stato modificato o meno).

NSWindowController

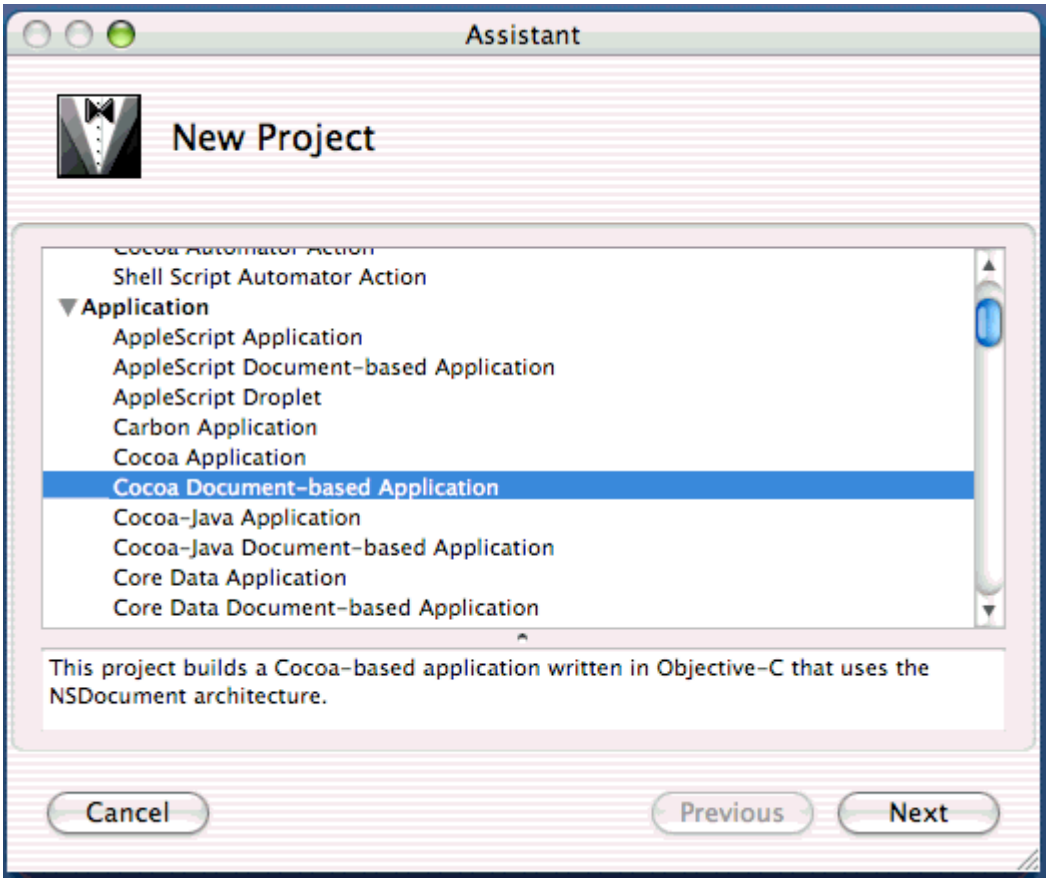
Un oggetto `NSWindowController` gestisce una finestra associata ad un documento. Un documento può avere diverse finestre associate, ciascuna con il suo `NSWindowController`. Questi oggetti non sono molto interessanti, nel senso che il comportamento di default fornisce a `NSDocument` sufficienti servizi per compiere tutte le operazioni più comuni. In effetti, nell'esempio successivo, non mi sono nemmeno accorto della loro esistenza.

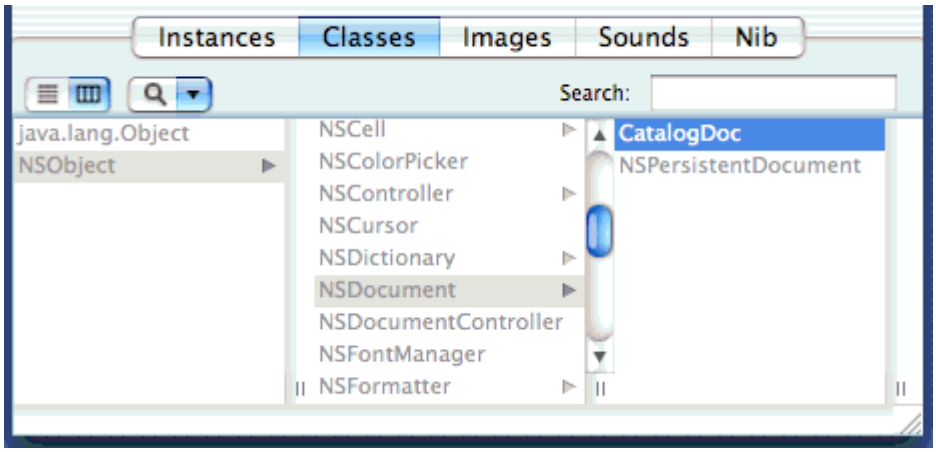
In definitiva, per fare una applicazione in grado di gestire più documenti, devo:

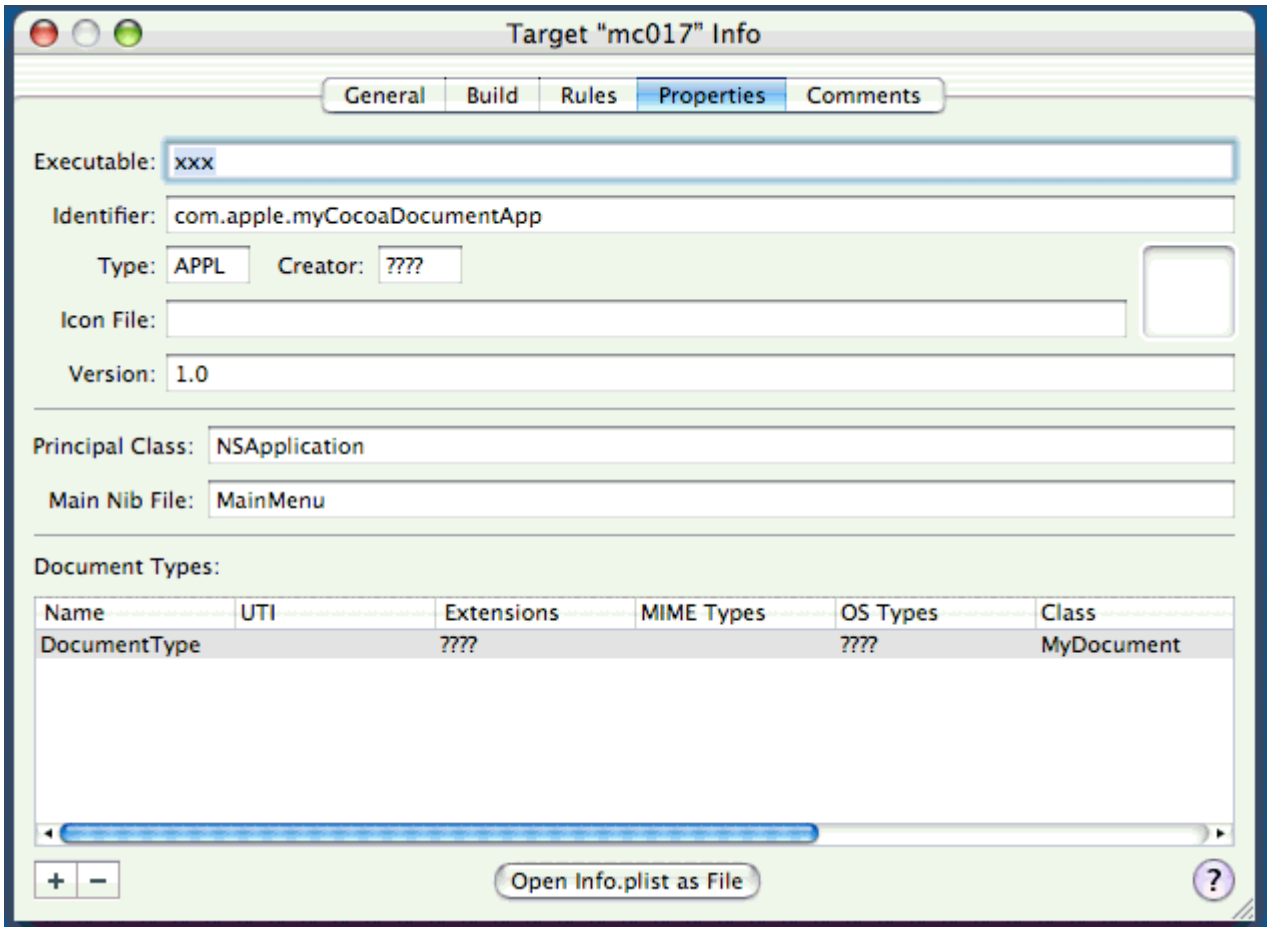
- costruire in IB la finestra; dovrò farlo in un file nib separato da quello principale, per poter caricare la finestra solo quando creo/apro un documento del tipo richiesto.
- fare un sottoclasse di `NSDocument`, e scrivere un po' di metodi che eseguano le operazioni necessarie.

Sembra facile. Parto.

Comincio







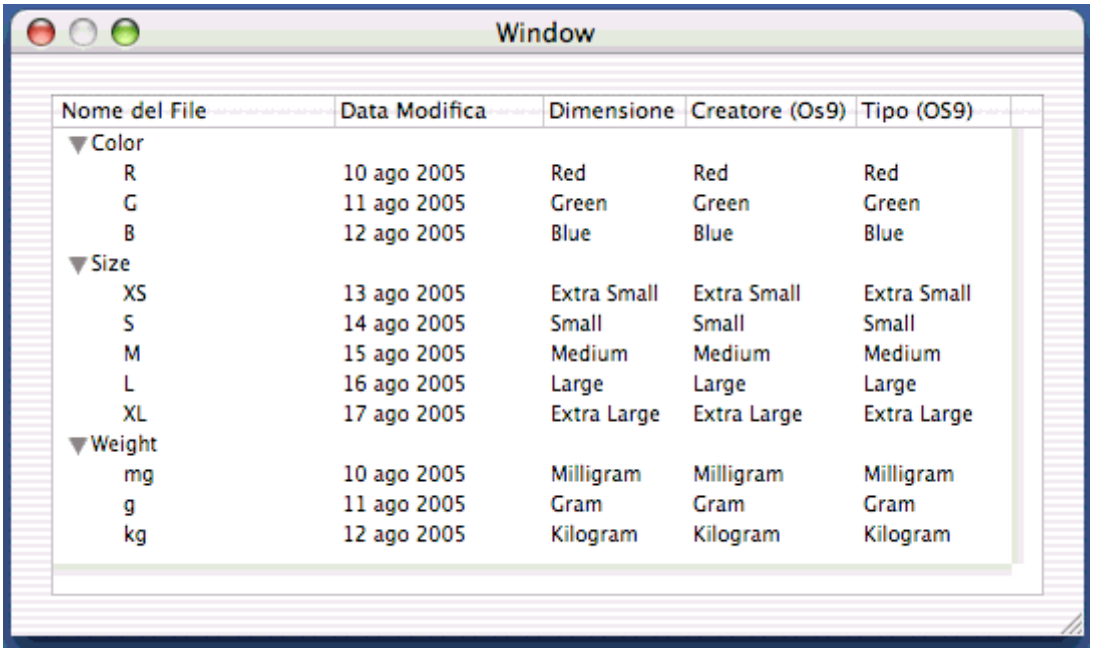
Faccio un nuovo progetto, dal pregnante nome `mc017`, e dico di farlo `document-based`. XCode mi viene incontro e produce di suo già alcuni file; intanto, due file nib separati, uno con il menu dell'applicazione (è il file nib principale), ed uno con dentro la finestra dell'applicazione, per ora vuota. Poi, ha già definito per me una sottoclasse di `NSDocument`, dall'orribile nome di `MyDocument`. Ovviamente, il nome non mi va bene, e comincio a cambiare tutto. Il file nib lo passo da `MyDocument.nib` a `CatalogWin.nib`. I file `MyDocument.h` e m sono rinominati come `CatalogDoc.h/m`. Devo anche intervenire nel codice sorgente per cambiare il nome della sottoclasse (uso la comoda funzione `Find` estesa a tutti i file del progetto). Poi, dopo una serie di penosi tentativi, scopro altre cose da cambiare. Il file `CatalogWin.nib`, facendo riferimento ad una classe di `NSDocument`, utilizza ancora il nome `MyDocument`. Vado in IB, nel

pannello `Classes` e rinomino la sottoclasse `MyDocument` di `NSDocument` in `CatalogDoc`. Non è finita. Devo specificare anche qual è il tipo di documento prediletto dall'applicazione (che è rimasto, nemmeno a dirlo, `MyDocument`). Si piglia ancora una volta la finestra Inspector del Target in questione; nei parametri della sezione *Properties*, sostituisco ancora una volta le occorrenze di MyDocument con `CatalogDoc`. E questa è finita.

Compilo, eseguo. Ho già una applicazione multi-documento funzionante (poco significativa, è vero, ma funzionante; dopotutto, cosa vi aspettate senza avere ancora scritto una riga di codice...).

Inserisco poi dall'esempio del capitolo precedente le classi che mi servono per sviluppare l'esempio corrente: `LSFormatter`, `FileStruct`, `LSFileInfo`, `LSDataSource`. Le userò tali e quali, senza alcuna modifica (che bello che bello che bello).

INTERFACCIA



Ricomincio da IB, e costruisco la finestra; butto via quella predefinita; dalle mie applicazioni precedenti riciclo la `NSOutlineView` (la copio dal precedente nib e la incollo qui, così evito di dover modificare le colonne e tutto il resto... si suppone che nel mondo object-oriented si cerchi di riciclare il più possibile). Già che ci sono, apro anche il nib principale con il menu, ed aggiungo un nuovo menu. Ci metto due voci, `Add...` e `Delete`, dove la prima voce intende rimpiazzare uno dei pulsanti che avevo nelle vecchie realizzazione dentro la finestra. I pulsanti `Load` e `Save` sono invece sostituiti dalle voci di menu `Open` e `Save`, appunto.

Ora, la mia domanda è la seguente. Chi è che svolge le funzioni della classe controllore (negli esempi fin qui, `FileInfoCtrl`)? In altre parole, adesso ho il problema di collegare la `NSOutlineView` con la sua sorgente di dati; negli esempi precedenti, il collegamento era automatico, in quanto attribuivo un dato valore ad un outlet in IB. Poi,

il meccanismo di caricamento automatico del file nib faceva il resto, ricreando il collegamento all'apertura del file nib. Qui invece ho una `NSOutlineView` (almeno) per ogni documento aperto, ed il file nib (e quindi ogni outlet presente) è condiviso da tutti i documenti.

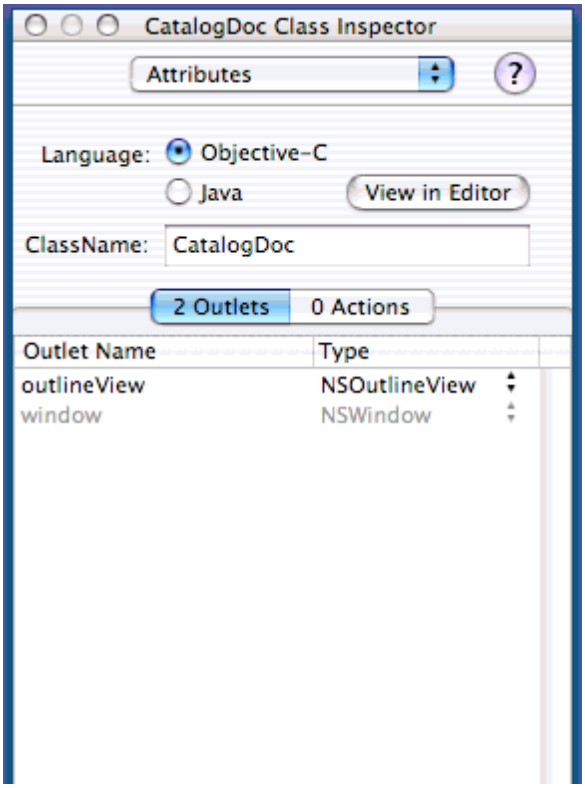
PROXY

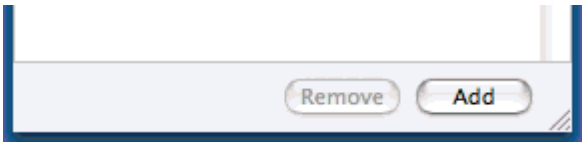
La cosa è risolta sfruttando il (fino ad ora) misterioso oggetto `File's Owner`. In IB esiste per ogni file nib una icona `File's Owner` (proprietario del file). Il proprietario è un oggetto, esterno al file nib, che funziona da intermediario tra gli oggetti estratti dal nib al momento del caricamento dello stesso all'interno di una applicazione e gli altri oggetti presenti nell'applicazione. Quest'oggetto è spesso detto essere un oggetto *proxy*, con una terminologia utilizzata anche nelle connessioni di rete. L'oggetto proxy per un file nib funziona come referente di ogni messaggio da e verso il mondo esterno. Quando un oggetto definito in un file nib deve inviare dei messaggi al resto dell'applicazione, lo invia al proxy. È poi compito di questo oggetto proxy farlo arrivare. Un proxy è quindi un altro metodo per incapsulare i dati e le procedure, e facilitare la modularità delle applicazioni. Quando il file nib è caricato, il proxy è associato automaticamente a chi ha richiesto il caricamento. Quindi, nel file nib principale, quello che contiene il menu dell'applicazione, il proxy sta al posto dell'oggetto `NSApplication`. Nel file nib che contiene gli oggetti relativi alla rappresentazione di un dato documento, il proxy è associato a `NSDocument`.

Tutta questa storia serve a giustificare le seguenti due operazioni.

La prima cosa è aggiungere un outlet alla dichiarazione della classe `CatalogDoc`:

```
IBOutlet NSOutlineView        * outlineView ;
```





Già che ci sono, ho esplicitamente dichiarato l'outlet del tipo corretto, in modo da facilitare il lavoro al compilatore e all'applicazione. Ora, bisogna che questa dichiarazione di classe sia comprensibile al file nib. Dall'interno di IB bisogna selezionare il pannello `Classes` e la voce `Read Files...` dal menu `Classes`. Da qui, occorre leggere appunto il file `CatalogDoc.h`; in questo modo l'oggetto `File's Owner`, che ho tipizzato essere un `CatalogDoc`, presenta nella palette di `Info`, lo outlet che ho dichiarato. La seconda operazione è collegare a questo outlet la outlineView contenuta nella finestra, col solito meccanismo del control-drag. Noto che connetto direttamente il `File's Owner`, non devo generare alcuna istanza della classe `CatalogDoc`, come avevo fatto negli esempi precedenti.

L'interfaccia è terminata, torno in XCode.

QUATTRO METODI

Devo completare la dichiarazione della classe `CatalogDoc`. Mi rifaccio alla vecchia classe `FileInfoCtrl`, e vedo che mi manca in pratica solo l'oggetto di classe `LSDataSource`. Anticipo anche la dichiarazione di un oggetto della classe `NSData`, che servirà a contenere i dati contenuti nella finestra.

```
NSData          * dataFromFile ;
LSDataSource * dataSource;
```

Per completare la realizzazione della classe `CatalogDoc`, si devono (come stabilito da documentazione) definire quattro metodi, che vado nell'ordine ad elencare:

```
- (NSString *)windowNibName ;
- (void)windowControllerDidLoadNib:(NSWindowController *) aController ;
- (NSData *)dataRepresentationOfType:(NSString *)aType ;
- (BOOL)loadDataRepresentation:(NSData *)data ofType:(NSString *)aType ;
```

Il primo metodo deve restituire il nome del file nib che contiene gli oggetti che rappresentano il documento a video. Con questo metodo `NSDocumentController` è in grado di selezionare il file nib appropriato per ogni tipo di documento. Il secondo metodo è chiamato al termine del caricamento del file nib; penso che qui inserirò tutte le istruzioni che negli esempi precedenti inserivo nel metodo `awakeFromNib`. Il terzo ed il quarto metodo servono per salvare e leggere i dati del documento da un file su disco. Il terzo metodo deve restituire, come oggetto della classe `NSData`, una rappresentazione del documento adatta ad essere salvata su disco. Il quarto metodo, infine, esegue l'operazione inversa: dai dati salvati su disco e forniti sotto forma di `NSData`, deve ricostruire i dati del documento e mostrarli a video. Eccoli uno alla volta. Il primo è molto facile:

```
- (NSString *)windowNibName
{
```

```
    return @"CatalogWin";
}
```

Avendo chiamato il file nib `CatalogWin.nib`, devo restituire la stringa senza estensione. Il secondo metodo ricorda molto `awakeFromNib` degli esempi dei capitoli precedenti, ma con qualche complicazione in più. Infatti, non ho qui la semplificazione dell'associazione automatica degli outlet al caricamento del file nib. Mi devo quindi arrangiare con le chiamate apposite.

```
- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{

    TOS9TCForm     *myDF1 = [[[ TOS9TCForm alloc ] init ] autorelease ];
     FileSizeForm      *myDF2 = [[[ FileSizeForm alloc ] init ] autorelease
];

    [super windowControllerDidLoadNib:aController];

      [[[ outlineView tableColumnWithIdentifier: @"typeCode" ] dataCell ]
setFormatter: myDF1 ];
     [[[ outlineView tableColumnWithIdentifier: @"creatorCode"] dataCell ]
setFormatter: myDF1 ];
     [[[ outlineView tableColumnWithIdentifier: @"fileSize"]     dataCell ]
setFormatter: myDF2 ];

    // devo costruire l'oggetto sorgente dei dati
    [ self setDataSource: [[LSDataSource alloc] init ] ];
    // e' la sorgente di dati di outlineView
    [ outlineView setDataSource: dataSource];

    [ self loadCatalogWithData: dataFromFile] ;
}
```

Lascio per il momento fuori dalla considerazione l'ultima riga del metodo, e passo al terzo metodo.

È una semplificazione del metodo `save2File` dell'esempio precedente, ho tolto tutto il codice relativo al panello Save, che sarà svolto diligentemente da `NSDocumentController`, e modifico quanto rimane. In pratica, l'istruzione per archiviare gli oggetti. Questa volta, invece che su file, lo faccio su un oggetto `NSData`, utilizzando l'apposito metodo:

```
- (NSData *)dataRepresentationOfType:(NSString *)aType
{
          return   (   [NSArchiver   archivedDataWithRootObject:   [dataSource
startPoint] ] );
}
```

Ed adesso, il metodo contrario: `NSDocumentController` fornisce i servizi per la selezione di un file in lettura, e passa un oggetto `NSData` al metodo seguente, che deve preoccuparsi di mostrare questi dati. Alla fine, mi viene fuori una cosa semplicissima:

```
- (BOOL)loadDataRepresentation:(NSData *)data ofType:(NSString *)aType
{
    if ( outlineView )
    {
```

```
        [ self loadCatalogWithData: data ] ;
    }
    else
    {
        dataFromFile = [ data retain ];
    }
    return ( YES );
}
```

Il metodo serve ad eseguire due operazioni differenti. Il primo caso, quando lo outlet `outlineView` esiste già, si verifica ad esempio quando l'utente sceglie di ritornare ad una versione di documento precedentemente salvata. Nel caso invece di una nuova finestra per il documento, è attivo il secondo caso, dal momento che questo metodo è chiamato prima del caricamento del file nib relativo. Tutto quello che devo fare è mantenere vivo l'oggetto `data`, assegnandolo (dopo un `retain`) alla variabile d'istanza. Ma allora, quando è che i dati vengono inseriti all'interno della finestra, ovvero, in altre parole, quando è chiamata il metodo `loadCatalogWithData` che fa tutto il lavoro sporco? Ecco spiegata l'ultima riga del metodo `windowControllerDidLoadNib:`, chiamato appunto al termine del caricamento del nib.

Riassumendo: se si seleziona un file in seguito ad una chiamata `Apri...` da menu, `NSDocumentController` fornisce i servizi base, che produce un oggetto `NSData` e lo passa al metodo `loadDataRepresentation:`. Costui piglia i dati, outlineView non esiste ancora, e li inserisce su `dataFromFile`. Poi `NSDocumentController` carica il file nib e crea la finestra; al termine, esegue `windowControllerDidLoadNib:`, che esegue le sue inizializzazioni e finalmente chiama `loadCatalogWithData` come ultima istruzione.

Se invece l'utente invoca `New` o `Nuovo` da menu, `NSDocumentController` salta al caricamento del file nib e chiude con `windowControllerDidLoadNib:`. Ancora una volta, quindi, `loadCatalogWithData` deve trattare due casi: i dati ci sono (Apri) oppure non ci sono (Nuovo documento).

E quindi, ecco il codice:

```
- (void) loadCatalogWithData: (NSData *) data
{
    if ( data )
    {
          NSMutableArray * catData = [NSUnarchiver unarchiveObjectWithData:
data ];
        // assegno il tutto alla sorgente di dati
        [ dataSource setStartPoint: catData ] ;
        // dico che sono cambiate le cose
        [ outlineView reloadData ];
    }
    else
    {
        [ dataSource setStartPoint: nil ] ;
        // dico che sono cambiate le cose
        [ outlineView reloadData ];
    }
}
```

Nel primo caso, estraggo i dati del documento dall'oggetto `NSData`, utilizzando il

metodo inverso dell'archiviazione. Questi dati rappresentano i punti iniziali della outlineView, che quindi assegno e informo delle modifiche. Nel secondo caso, inizializzo i dati e nil.

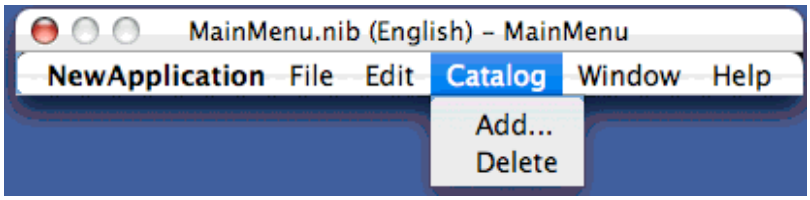
Compilo e provo. Funziona. Che meraviglia.

Adesso però devo vedere che fare dei menu.

Capitolo 018 - Menu e Palette

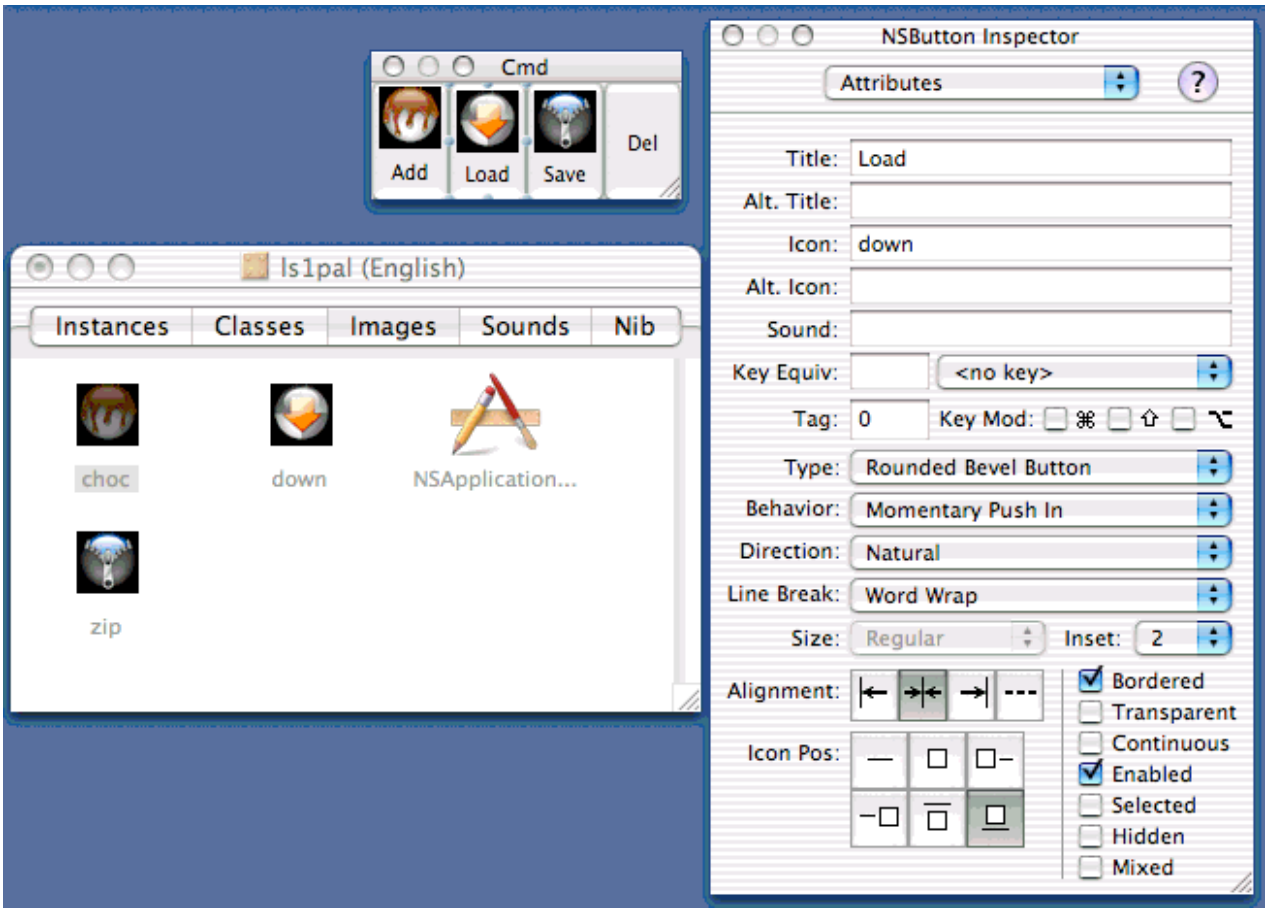
In questo capitolo aggiungo due voci di menu e una palette di comandi all'applicazione finora costruita. Il menu l'ho già introdotto il capitolo precedente, con due voci all'interno di un menu specifico; intendo con questo sostituire i pulsanti che, ancora un capitolo precedente, utilizzavo per aggiungere elementi al catalogo di file. I pulsanti, usciti di scena con i menu, rientrano invece all'interno di una palette che voglio provare ad inserire in questo ambiente multi-documento.

Primo inserimento: 4 Febbraio 2002



Sorgenti: Tentativi e molti errori.

INTERFACCIA



Come al solito, la prima cosa da fare è di entrare in IB e di costruire l'interfaccia utente. Se le due voci di menu `Add...` e `Delete` erano già state definite in precedenza,

manca totalmente la palette. Come è giusto, questa si trova in un file nib separato, che chiamo `ls1pal.nib` (ovvero, la prima palette di livio sandel...). è una cosa molto semplice, prevedo quattro bottoni, uno per aggiungere elementi al catalogo, uno per eliminarli, uno per aprire cataloghi (equivalente alla voce di menu `Open...`) ed uno per salvare il contenuto dei cataloghi (equivale a `Save`).

Visto che ci sono, scopro come sia molto facile aggiungere immagini e suoni ai pulsanti.

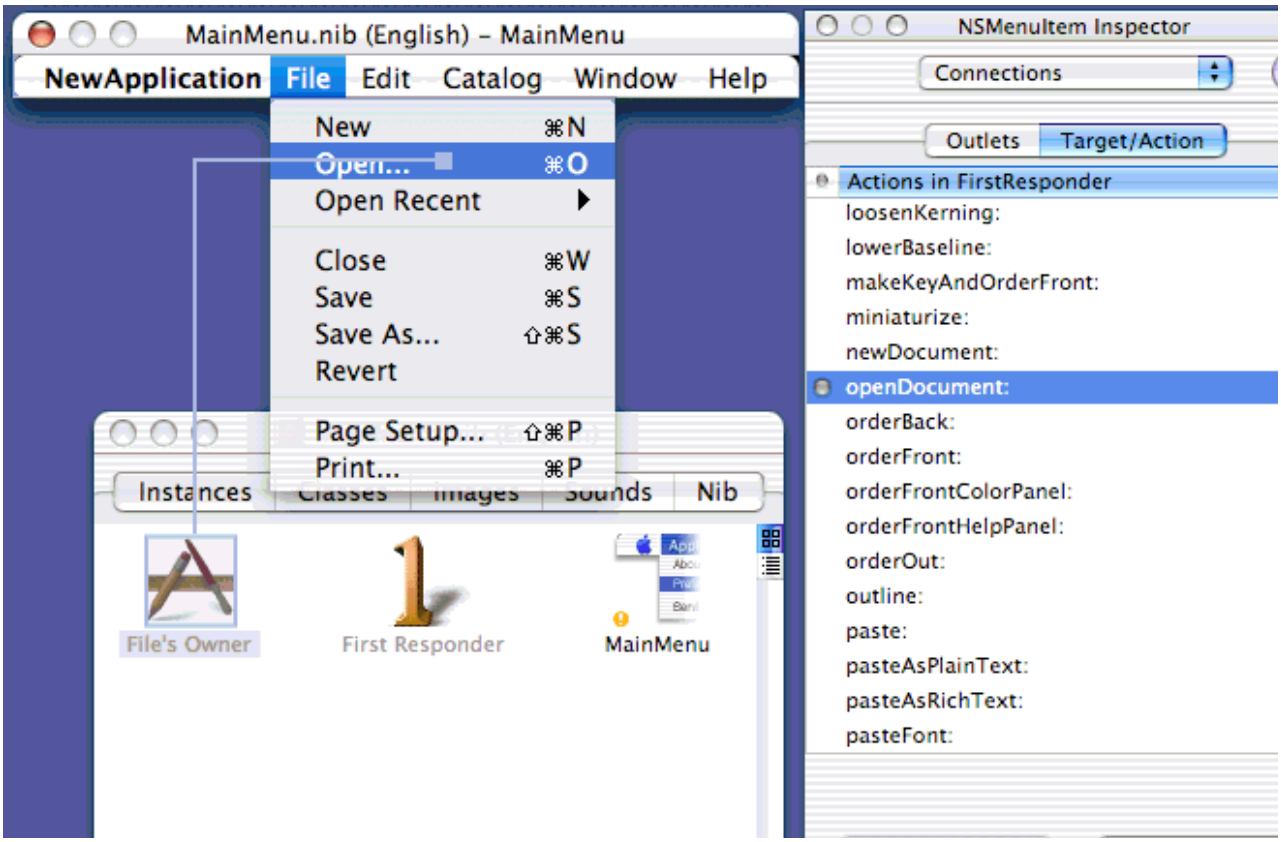
Per prima cosa, occorre importare le immagini che si intendono usare all'interno del progetto XCode. Io ho recuperato dei disegni non so più da dove in formato pict.

Questi disegni sono poi disponibili anche in IB una volta che si seleziona il pannello `Images`. Per aggiungere una immagine ad un pulsante si tratta semplicemente di fare drag'n'ndrop dalla finestra al pulsante. Scopro poi altri giochini interessanti che si possono fare con i pulsanti. Se ad esempio utilizzate una seconda immagine, assegnandola al campo `alt.icon`, si possono scambiare tra loro le immagini all'attivazione del pulsante, semplicemente selezionando la voce opportuna nel menu `Behaviour` della palette di info relativa al pulsante (funziona anche per pulsanti di solo testo, scambia tra loro il `Title` e `alt.Title`).

Per aggiungere un suono, stesso meccanismo. Tuttavia, trovo piuttosto noioso il suono sul pulsante (alla lunga, lavorandoci un po', mi infastidisce...).

Ovviamente, posso provare subito al momento l'effetto che fa la palette dei pulsanti da dentro IB, selezionando il menu `Test Interface`.

COLLEGAMENTI



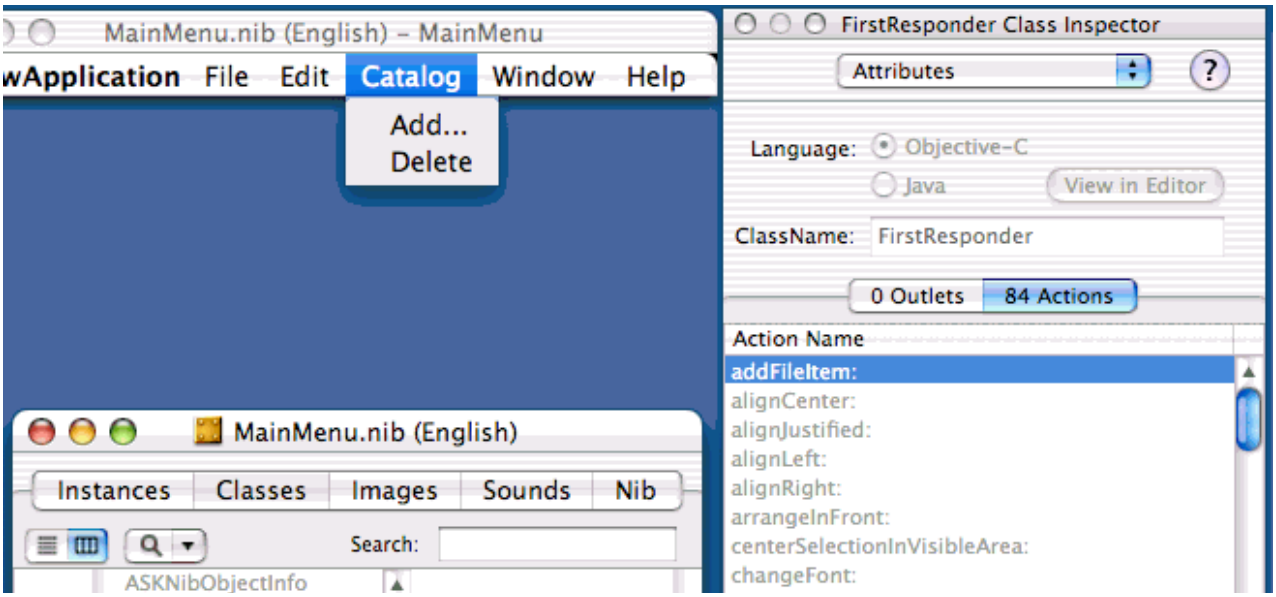
Adesso comincio la parte interessante, ovvero i collegamenti. Una cosa che era passata inosservata nel capitolo precedente, erano i collegamenti predefiniti del menu principale. Apro il file `MainMenu.nib` dentro IB, ed esamino le singole voci di menu. La maggior parte di queste è già associata ad una qualche operazione. Ad esempio, la voce"`New` ha già realizzato, nel paradigma target/action, il collegamento tra la selezione del menu e l'azione `NewDocument`. Sì, ma a chi è diretto il messaggio? Al `First Responder`, l'oggetto misterioso di questo paragrafo.

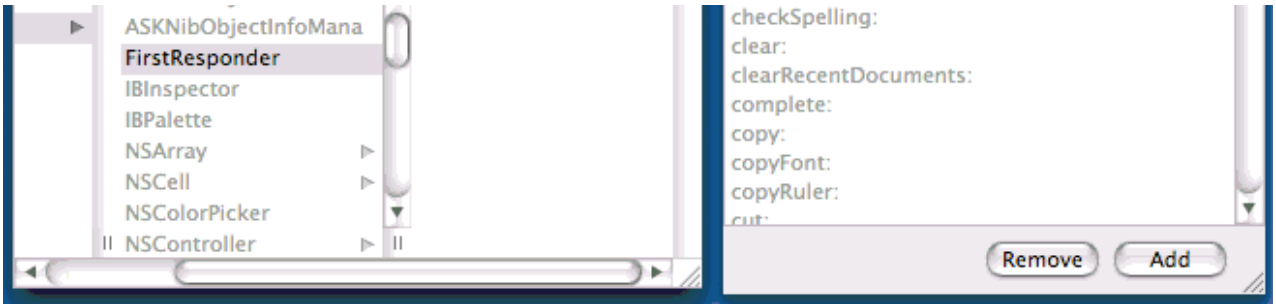
La piglio da lontano: una applicazione, una volta che ha effettuato tutte le inizializzazioni necessarie, entra nell'infinito loop degli eventi. L'oggetto `NSApplication` è il gestore principale dell'applicazione: esegue un passo del loop, estrae, se presente, un evento, e lo distribuisce a chi di dovere. La distribuzione dell'evento dipende anche dal tipo di evento stesso. Ad esempio, un clic del mouse è diretto all'oggetto visuale immediatamente sotto il mouse. Le cose si complicano se l'evento è un carattere da tastiera. Chi è che riceve l'evento? Diciamo la finestra davanti a tutte le altre (cosa per altro non sempre vera...), ma, all'interno della finestra, quale dei vari possibili elementi?

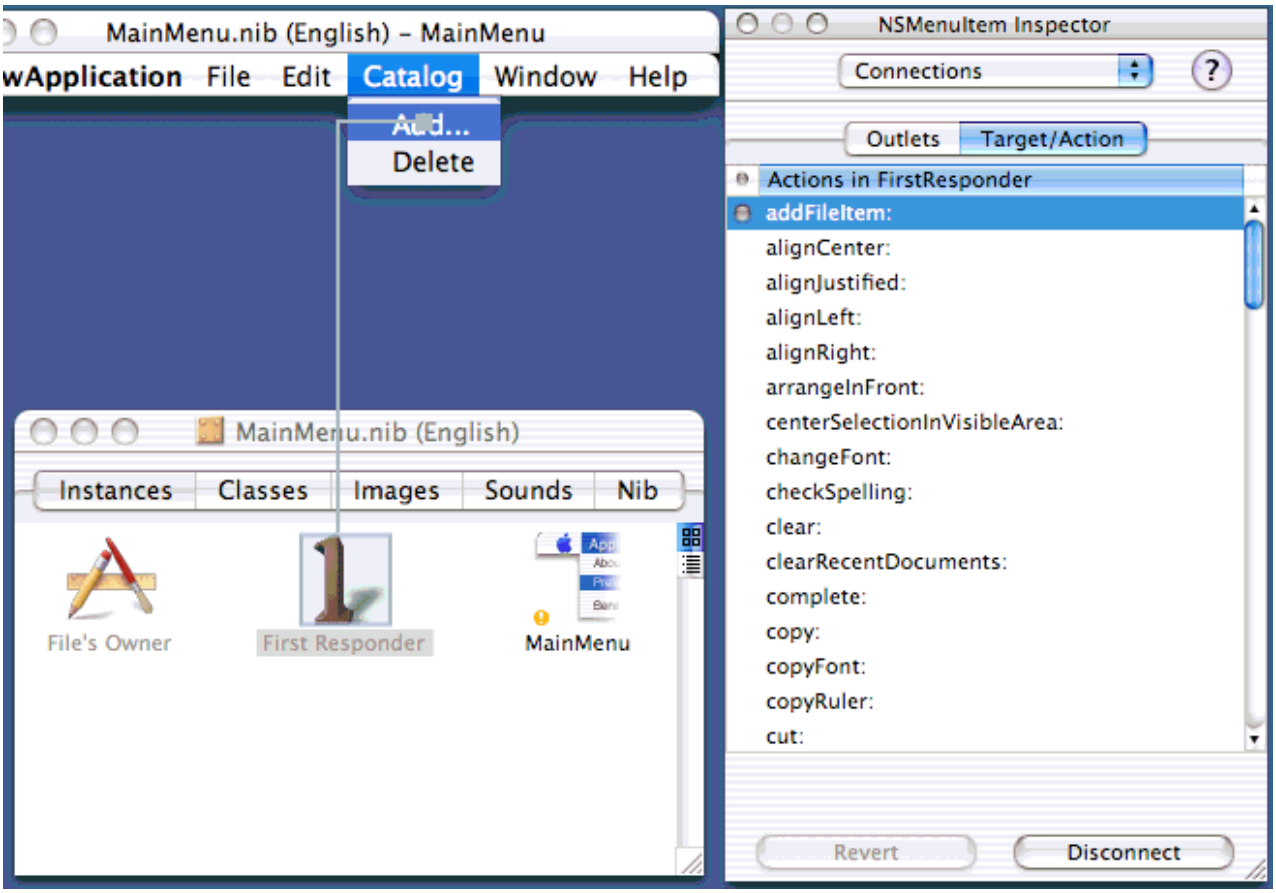
Ebbene, l'elemento all'interno di una finestra che riceve per primo tutti gli eventi non dedicati, è il `First Responder` (risponditore designato).

Se il risponditore designato è in grado di trattare l'evento, lo gestisce, consumandolo. Se non è in grado di farlo, passa l'evento al risponditore successivo nella catena dei risponditori. Tipicamente, lo passa al suo superiore gerarchico. Alla fine, i gestori degli eventi generici sono la finestra, il documento e l'applicazione. Ad esempio, nel caso dei menu, la maggior parte dei comandi del menu `File` va a riferirsi al documento (apri, chiudi, salva, stampa), e quindi alla classe controllore del documento, CatalogDoc (l'unica, in questo momento).

Mi viene quindi naturale associare ai due nuovi menu `Add...` e `Delete` una accoppiata target/action, inserendo l'azione come uno dei metodi della classe CatalogDoc.







Vado quindi nel file nib del menu principale, seleziono la classe `First Responder` nella finestra principale, ed aggiungo le azioni `addFileItem:` e `delItem:` nell'elenco. Poi passo ai nuovi menu e, con la classica azione control+drag, collego i menu all'istanza di `First Responder`.

Torno a questo punto in XCode, e a mano aggiungo ii metodi corrispondenti nei file della classe `CatalogDoc`.

```
- (void) addFileItem: (id)sender ;
- (void) delItem: (id)sender ;
```

Per la scrittura effettiva dei metodi, il primo non presenta preblemi, dal momento che ricorda molto da vicino un metodo già scritto negli esempi precedenti (anzi, praticamente non cambia):

```
- (void) addFileItem: (id)sender
{
```

```
    int risposta;
    NSOpenPanel *oPanel = [ NSOpenPanel openPanel ];

    [ oPanel setFrame: NSMakeRect(0, 0, 500, 200) display: NO ];
    [ oPanel setTitle:@"Aggiungi File" ];
    [ oPanel setPrompt:@"Seleziona un file qualsiasi" ];
    [ oPanel setCanChooseDirectories:YES ];
    [ oPanel setCanChooseFiles:YES ];
    [ oPanel setAllowsMultipleSelection:NO ];
    [ oPanel setResolvesAliases:NO ];

    risposta = [oPanel runModalForTypes: nil];

    if (risposta == NSOKButton) {
        NSArray *filesToOpen = [ oPanel filenames ];
        NSString *aFile = [ filesToOpen objectAtIndex: 0 ];
         FileStruct * fInfo = [[ FileStruct alloc ] initTreeFromPath: aFile
];
        [ dataSource addFileEntry: fInfo ];
        [ outlineView reloadData ];
    }
}
```

RICORSIONE

Arrivo adesso alla novità del metodo di eliminazione di un elemento dalla finestra del catalogo. Dopo aver inserito con leggerezza il menu nell'interfaccia, mi sono pentito, dal momento che mi sembrava un compito improbo. Poi, riflettendoci, diventa molto più semplice. Il problema è che ho un elemento all'interno di una gerarchia, e non ho alcun punto di riferimento... bisogna esplorare tutta le gerarchia alla sua ricerca. Comincio così:

```
- (void) delItem: (id)sender
{
    FileStruct * fInfo ;
    int    rowsel = [ outlineView selectedRow ];
    if ( rowsel == -1 ) return ;
    // se arrivo qui, c'e' una riga selezionata
    fInfo = [ outlineView itemAtRow: rowsel ] ;
    // adesso voglio eliminare questo elemento da dataSource
    // e' un vero pasticcio: devo cercare la directory di cui
    // e' figlio, cosa non facile...
    // ed allora, considero ricorsivamente gli start points e vado
    DeleteItemFromHierarchy( [dataSource startPoint], fInfo );
    [ outlineView reloadData ];
}
```

All'inizio, discrimino il fatto se effettivamente c'è una riga selezionata all'interno del documento. Se non c'è, il metodo `selectedRow:` della classe `NSOutlineView` restituisce (-1), e quindi il metodo non ha nulla da fare. Una volta scoperta che c'è una riga selezionata, ricavo l'elemento selezionato. Noto che l'elemento è l'oggetto vero e proprio, e non ha una sua rappresentazione. Non basta adesso eliminare l'oggetto, devo anche eliminare il collegamento a questo oggetto, in altre parole, devo cercare la directory in cui l'elemento (il file) è contenuto. Mi appoggio ad una funzione che scrivo appositamente, e che sarà chiamata ricorsivamente (e se qui non avete mai visto una funzione ricorsiva, fate un bel respiro, prendetevi un po' di tempo, e preparatevi ad un ragionamento non facile).

La struttura dati è costituita da una serie di `NSMutableArray`, in cui ogni elemento del vettore è potenzialmente un altro `NSMutableArray`. La mia ricerca comincia dal livello più alto, quindi dal vettore dei punti di partenza della sorgente dati.

La funzione `DeleteItemFromHierarchy` riceve come parametri il vettore e l'elemento da eliminare. Resituisce il valore 1 non appena trova ed elimina l'oggetto, 0 in caso contrario. A questo punto, esamino il codice:

```
int
DeleteItemFromHierarchy ( NSMutableArray * hier , FileStruct * fItem )
{
    int ni ;
    unsigned int    i;

    // se lo item e' nell'array, bene
    if ([ hier containsObject: fItem ])
    {
        [ hier removeObject: fItem ] ;
        return ( 1 );
    }
    // non c'e', lo cerco in tutti i suoi figli
    ni = [ hier count ] ;
    for ( i = 0 ; i < ni ; i ++ )
    {
        FileStruct * locRoot = [ hier objectAtIndex: i ];
        if (DeleteItemFromHierarchy ( [locRoot fileList] , fItem ) == 1 )
            return ( 1 );
    }
    // se arrivo qui, non ho ancora trovato il file
    return ( 0 );
}
```

Se sono fortunato, l'oggetto `fItem` si trova subito all'interno del vettore. Uso il metodo `containsObject:` proprio della classe `NSMutableArray` per verificare questo fatto. Se così è, elimino l'elemento utilizzando ancora una volta un metodo di `NSMutableArray`, ovvero `removeObject:`. L'effetto di questo metodo è molto di più pesante di quanto appare a prima vista. Infatti il metodo di suo invia un release all'oggetto. Così facendo, l'oggetto non ha più nessuno che si riferisca a lui, e quindi riceve anche un messaggio di `dealloc`. Però io avevo sovrascrittto il metodo `dealloc`, facendo in modo che non venisse eliminato solo l'oggetto, ma anche il vettore che, nel caso il file fosse una directory, contiene tutti i file contenuti nella directory. Per tanto, l'eliminazione di un oggetto provoca automaticamente l'eliminazione di tutta la sotto-gerarchia di file che da questo oggetto si dipana.

Vado avanti. Sono sfortunato, e l'elemento cercato non è direttamente all'interno del vettore di partenza. Mi tocca quindi esaminare tutte le sotto-gerarchie di ciascun elemento del vettore. Ecco quindi un ciclo for che itera su tutti gli elementi (che ho opportunamente contato). Estraggo dall'elemento i-esimo del vettore considerato il vettore con tutti i suoi file, ed invoco nuovamente la funzione `DeleteItemFromHierarchy` scendendo di un livello nella gerarchia.

Il motivo per cui la funzione restituisce 0 in caso di fallimento e 1 in caso di riuscita della cancellazione è per fermare la ricerca al momento del reperimento dell'oggetto. È infatti inutile cercare l'oggetto nel resto della gerarchia ancora da esplorare nel momento in cui l'ho trovato ed eliminato.

PALETTE

Giunti qui con tutto funzionante, non mi resta che collegare i pulsanti della palette. Con l'idea del risponditore designato, la cosa è semplicissima. Collego con la tecnica del control+drag il pulsante `Add` al metodo `addFileItem:`, `Del` a `delItem:`; poi, sfrutto i metodi esistenti `openDocument:` e `saveDocument:` per collegarei pulsanti `Load` e `Save`. Gli ultimi due metodi sono molto semplicemente quelli che IB collega di default alle voci di menu `Open...` e `Save`.

C'è da gestire l'apparizione e la chiusura della palette stessa. Infatti, la palette non è creata automaticamente al lancio dell'applicazione, in quanto si trova in file nib differente da quello principale (seguo il principio *ogni finestra il suo nib*). Devo quindi aggiungere il controllore della finestra, sotto forma della classe `palette`.

Ora, all'interno dell'applicazione esiste sempre una ed una sola palette; utilizzo allora una tecnica differente dal solito. Dico che esiste un metodo per la classe (attenzione: per la classe, non per ogni istanza) `palette` che permette di recuperare l'oggetto palette attivo

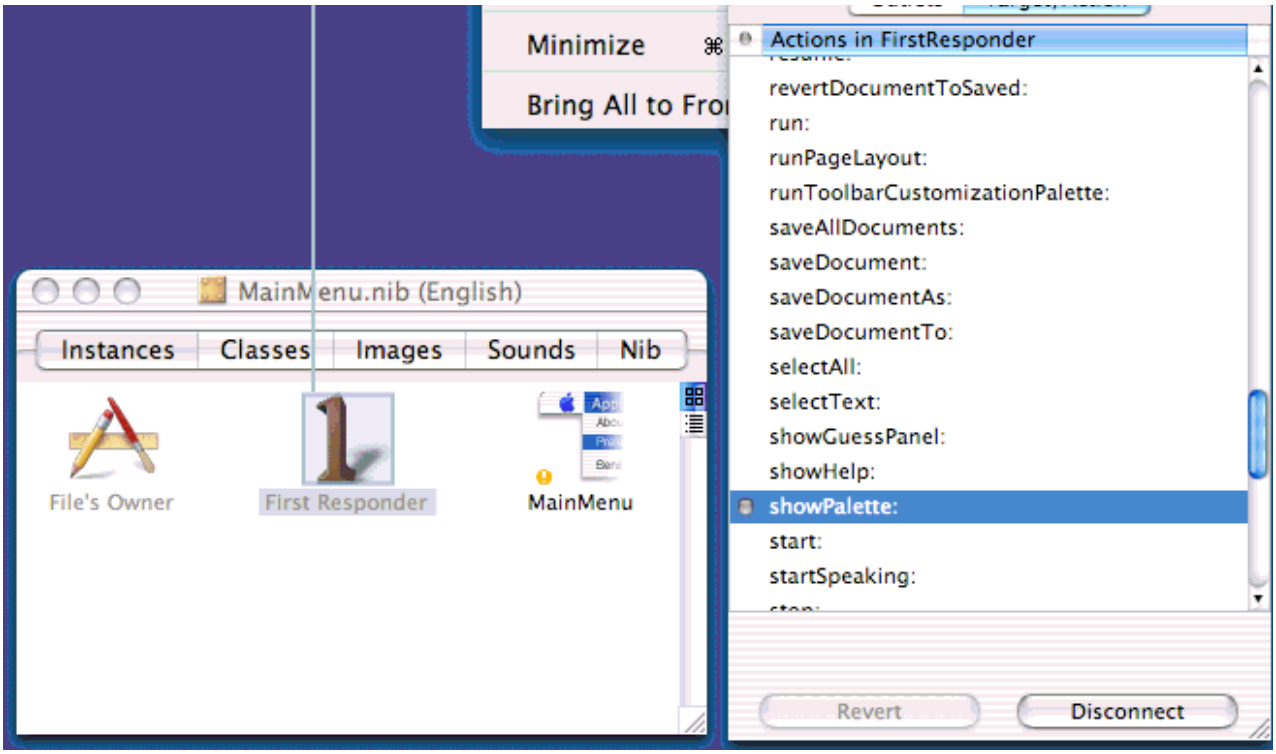
```
+ (id) sharedPalette
{
    static palette * _sharedPalette = nil ;

    if ( ! _sharedPalette )
    {
        _sharedPalette = [ [palette alloc] init ];
    }

    return ( _sharedPalette );
}
```

In primo luogo, c'è da notare il metodo, che comincia con '+' invece che con il classico '-'. Con questo, si indica che il metodo è della classe piuttosto che di ogni istanza. Poi c'è una variabile definita `static`. Questa variabile è unica per la classe, ed è inizializzata a `nil` (cioé, nulla). La prima volta che si invoca il metodo, `_sharedPalette` vale `nil`, e il metodo alloca ed inizializza la palette. Tutte le altre volte, restituisce l'oggetto `palette` costruito la prima volta. In questo modo, sono sicuro di costruire una sola volta la palette.

Il metodo `init` è molto semplice:

```
- (id) init ;
{
    self = [ self initWithWindowNibName: @"ls1pal"];
    return ( self );
}
```

Uso il metodo `initWithWindowNibName:` per caricare la palette dal file nib che la contiene.



Non mi rimane adesso che aggiungere un meccanismo per visualizzare la palette. Torno in IB ed aggiungo una voce di menu `Show Palette` nel menu `Window`. A questa voce ci collego il metodo `showPalette:` che mi sono premunito di definire all'interno di `First Responder`. Poi, all'interno della classe `CatalogDoc`, aggiungo in XCode il metodo corrispondente:

```
- (void) showPalette: (id)sender
{
    [[palette sharedPalette] showWindow: sender] ;
}
```

E qui c'è il problema sul quale mi sono arenato. Il meccanismo funziona la prima volta che seleziono il menu: la finestra appare nel suo splendore. Se però la chiudo, e provo a riaprirla, non succede alcunché. E non capisco. Magari, bisogna leggere un po' di documentazione.

Infatti (qualche ora più tardi). Ho dimenticato una cosa stupidissima. Il collegamento tra il `File's Owner` del file nib che contiene la palette e la finestra della palette stessa. Detto per esteso: quando ho costruito la finestra della palette, data la semplicità della classe controllante (`palette` è appunto una sotto-classe di `NSWindowController`), mi sono dimenticato della cosa più semplice, ovvero, quale sia la finestra che la classe controlla. In pratica, si tratta di importare in IB il file `palette.h` in modo da rendere noto il tipo di `File's Owner`, e poi di collegare tramite lo outlet predefinito `window` di `File's Owner` la finestra che ho definito.

Facile, banale e vitale.

Capitolo 019 - Sesso Droga e Drag'n'Drop

Purtroppo, o per fortuna, in questo capitolo non si parla di sesso e di droga, ma solo di drag'n'drop. Anzi, a dirla tutta, solo della parte 'drop'. Scopo infatti di questo capitolo è di realizzare le funzioni di drop all'interno di una finestra del Catalogo. In questo modo è possibile aggiungere file e directory intere trascinandole da una finestra del Finder all'interno di una finestra di Catalogo.

Sorgenti: A ruota libera a partire dalla documentazione Apple.

Primo inserimento: 18 Febbraio 2002

PROTOCOLLI

Ho già parlato di protocolli, come di un meccanismo che aggiunge funzionalità ad una classe senza dover definire una sottoclasse. In pratica un protocollo è una collezione di metodi; una classe si dice aderire al protocollo se definisce tutti (o parte) dei metodi che fanno parte del protocollo.

Nel caso, interessa il protocollo `NSDraggingDestination`; raggruppa i metodi che il destinatario di una operazione di drag'n'drop (lasciate che chiami l'operazione *draggare*; è orribile, ma non posso ripetere ogni volta la locuzione precedente). Come è noto, quando seleziono qualcosa e la draggo in giro, il sistema operativo fornisce una sorta di immagine dell'oggetto draggato.

Contestualmente, sempre il sistema operativo si occupa di mandare dei messaggi agli oggetti sopra cui l'utente sta draggando l'oggetto, fino a perfezionare l'operazione di drag'n'drop quando l'utente rilascia il mouse e con lui l'oggetto draggato (lasciate che chiami questa operazione *droppare*).

NSDRAGGINGDESTINATION

Il protocollo consiste di sei metodi; diciamo che l'utente ha pescato un oggetto e lo sta draggando in giro. Ad un certo punto raggiunge l'oggetto della mia applicazione sul quale è possibile droppare l'oggetto draggato.

Non appena l'immagine draggata entra nello spazio del destinatario, il sistema operativo invia il messaggio `draggingEntered:` al destinatario. Finché l'oggetto rimane in zona, al destinatario è inviato il messaggio `draggingUpdated:`. Se l'utente cambia idea e si sposta fuori dell'area del destinatario, il sistema invia il messaggio `draggingExited:`. Se rientra, si ricomincia con `draggingEntered:`.

Se invece l'utente droppa l'oggetto sopra il destinatario, possono accadere due cose: se il destinatario non è in grado di trattare l'oggetto droppatogli addosso, questo ritorna immediatamente al suo posto e tutto finisce lì. Se invece il destinatario accetta la droppata, il sistema operativo invia tre messaggi: `prepareForDragOperation:`, in

cui il destinatario può predisporsi al ricevimento dati, `performDragOperation:`, in cui il destinatario effettua finalmente l'operazione; infine, se tutto è andato bene, al destinatario è inviato il messaggio `concludeDragOperation:`.

La cosa sembra inutilmente complicata, ma in realtà ogni metodo ha la sua ragion d'essere, come scoprirò più tardi. Infatti, adesso cerco di capire due cose: cosa c'entrano gli Appunti (la PasteBoard) e come fare a dire che un destinatario è in grado di ricevere oggetti droppati.

NSPasteboard

A quanto pare, il meccanismo per draggare oggetti ha a che fare con gli Appunti, ovvero con quel luogo misterioso dove va a finire tutto ciò che copio o taglio, in quell'intervallo di tempo che intercorre tra l'operazione di copiatura e quella di incollaggio. Quanto copio qualcosa (o, nella fattispecie, quando comincio a draggare qualcosa), le informazioni dell'oggetto copiato o draggato vanno a finire in un archivio temporaneo d'Appunti, un oggetto della classe `NSPasteboard`.

Quando il destinatario cerca di capire con che oggetto draggato ha a che fare, ricava le informazioni recuperandole appunto da una `NSPasteboard`.

In effetti, ogni metodo del protocollo è parametrizzato nello stesso modo:

```
- (tipoRisposta) nomeDelMetodo: (id <NSDraggingInfo>) sender ;
```

In altre parole (e più chiaramente), quando il sistema operativo invia un messaggio `NSDraggingDestination` (ad esempio, `draggingUpdated:`), aggiunge come parametro un oggetto dal quale è possibile recuperare le informazioni tramite una pasteboard. Ancora poco chiaro? Ecco un esempio di codice (copio brutalmente dalla documentazione Apple):

```
- (BOOL)draggingUpdated:(id <NSDraggingInfo>)sender {
    NSPasteboard *dragPasteboard;

    // get the dragging pasteboard from the NSDraggingInfo
    dragPasteboard = [sender draggingPasteboard];
     // now use the pasteboard to get whatever information the destination
wants
    types = [dragPasteboard types];
    // etc.
}
```

Cosa succede? La prima istruzione eseguita è copiare le informazioni dell'oggetto draggato localmente, su di un oggetto `NSPasteboard`, invocando il metodo `draggingPasteboard:`. A questo punto, si possono sfruttare i metodi propri della pasteboard per ricavare le informazioni. Non stupisca il metodo `types:`. Come sono diversi le tipologie degli oggetti che posso copiare negli Appunti (testo, immagini, eccetera), così sono diverse le tipologie degli oggetti che draggo. Qualche applicazione può accettare come oggetti droppati testi ed immagini (si pensi ad un elaboratore di testi), l'applicazione Catalogo accetta come oggetti droppati dei nomi di file (e directory).

COSA DROPPARE

Eccomi arrivato all'altra questione. Come indicare (e a chi) quali oggetti si possono droppare felicemente sulle finestre dell'applicazione. In primo luogo, appare chiaro che per droppare qualcosa, il destinatario deve essere rappresentato da una porzione di spazio sullo schermo. Questo significa che solo finestre (oggetti `NSWindow`) o viste (oggetti `NSView`) possono aderire al protocollo `NSDraggingDestination`. Considerato che la maggior parte degli oggetti visibili a video (forse tutti) sono in qualche modo sottoclassi di `NSView`, non c'è problema. Piuttosto, la difficoltà sta altrove; perché un oggetto possa accettare oggetti droppati, deve aderire al protocollo. Piglio ad esempio l'oggetto `NSOutlineView` che costituisce la maggior parte (tutto lo spazio utile, in effetti) della finestra Catalogo. L'oggetto è una sottoclasse di `NSView`, quindi può andar bene. Devo adesso trovare un meccanismo per attaccarci i metodi del protocollo. Il fatto è che non ho a disposizione i file .h e .m da modificare... L'unica possibilità che ho senza fare acrobazie è di sfruttare il meccanismo della delega. Anche di questo ho già parlato: in pratica una classe dice che una serie di messaggi non sono da lei trattati direttamente, ma solo delegati ad un'altra classe, indicata appunto come classe delegata.

Bene, per economizzare sulle classi e sugli oggetti, decido che la finestra del documento Catalogo è lei in prima persona destinataria delle operazioni di drop. Poiché ho già un delegato per tale finestra (che, guarda caso, è proprio la classe documento `CatalogDoc`), aggiungo la definizione dei sei metodi all'interno del file `CatalogDoc.m`.

Inoltre, dichiaro che tale finestra accetta come oggetti droppati solo dei nomi di file (quindi, path completi di file o directory). Ecco quindi che devo aggiungere la seguente istruzione all'interno del metodo `windowControllerDidLoadNib:`, ovvero quando l'ambiente operativo ha terminato di caricare l'interfaccia finestra e si appresta a visualizzarla a tutti.

```
- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{
    // ... le solite cose gia' viste...
    // registro la finestra che accetti drag
    [ [ aController window] registerForDraggedTypes:
        [NSArray arrayWithObject: NSFilenamesPboardType] ];

}
```

Esamino l'ultima complicata istruzione un passo alla volta. In primo luogo, devo recuperare la finestra vera e propria. Siamo in una classe `NSDocument`, quindi devo passare attraverso una classe `NSWindowController` prima di arrivare alla finestra: ecco quindi che piglio `aController`, che è proprio il controllore della finestra appena creata, e invio il messaggio `window`: per recuperare la finestra. Dico quindi che questa finestra accetta (si registra presso il sistema operativo) che siano droppati su di essa degli oggetti di un certo tipo (o di diversi tipi, se preferisco): devo usare il metodo `registerForDraggedTypes:`. L'argomento di questo metodo è appunto l'elenco dei tipi di oggetto che accetto; io ho scelto di accettare solo oggetti del tipo `NSFilenamesPboardType`, ovvero dei nomi di file. Tuttavia, ho dovuto ugualmente costruire un vettore (con un unico elemento) per passarlo come argomento.

Altri tipi di oggetti sono ad esempio del 'semplice' testo in formato RTF (`NSRTFPboardType`), oppure una immagine TIFF (`NSTIFFPboardType`), cose del genere (un elenco si trova nella documentazione di `NSPasteboard`).

I Metodi

Comincio allora a realizzare i sei metodi. Comincio con

```
- (unsigned int)draggingEntered:(id <NSDraggingInfo>)sender
```

che arriva alla finestra quando l'utente dragga qualcosa su di essa.

Il metodo deve restituire un valore che indica cosa se ne farà il destinatario dell'oggetto se questo venisse droppato. Ci sono diverse possibilità, tra cui la copia, il collegamento, eccetera. Io decido che ne faccio una sorta di collegamento, quindi restituisco la costante predefinita `NSDragOperationLink`. Noto che questo valore può influenzare il modo con cui l'oggetto draggato è rappresentato a video. Il sistema operativo infatti, in base al valore restituito, può modificare l'immagine che rappresenta l'oggetto spostata dal cursore.

Nel metodo qui sotto riportato controllo anche che l'oggetto draggato preveda di poter fornire una informazione sotto forma di nome di file.

```
- (unsigned int)draggingEntered:(id<NSDraggingInfo>)sender
{
    NSPasteboard *pboard;

    pboard = [sender draggingPasteboard];

    if ([[pboard types] indexOfObject:NSFilenamesPboardType] != NSNotFound)
{
        return NSDragOperationLink;
    }
    return NSDragOperationNone;
}
```

Il metodo successivo è facile in quanto segue la stessa filosofia del precedente:

```
- (unsigned int)draggingUpdated:(id<NSDraggingInfo>)sender
{
    return NSDragOperationLink;
}
```

Qui è molto facile, continuo semplicemente a rispondere che farò un collegamento dell'oggetto draggato. In realtà, io rispondo `NSDragOperationLink`, non perché ci abbia ragionato a lungo sopra, ma solo perché mi sembra la risposta più ragionevole (e, soprattutto, funziona...).

Anche il terzo metodo è molto facile:

```
- (void)draggingExited:(id<NSDraggingInfo>)sender
{
    return ;
```

```
}
```

Non avendo fatto nulla di speciale, non devo fare cose speciali quando l'utente se va cambiando idea sul drag'n'drop (magari invece stava semplicemente passando sopra la mia finestra per andare altrove...). In casi di drag'n'drop più evoluti, potrebbe essere necessario fare qualcosa anche qui.

Sto arrivando al nocciolo della questione:

```
- (BOOL)prepareForDragOperation:(id<NSDraggingInfo>)sender
{
    return ( YES );
}
```

Il metodo sembra abbastanza facile; in realtà, il compito del metodo è di prepararsi ad effettuare l'operazione di ricezione dell'oggetto droppato (qui l'utente ha già rilasciato il mouse), e di rispondere YES oppure NO se pensa di poter eseguire con successo o meno l'operazione stessa. Il sistema operativo, in base alla risposta di questo metodo, decide se riportare l'oggetto a posto (nel punto di partenza) e chiudere qui tutta la faccenda (il metodo ha risposto NO), oppure tutta la storia arriva al suo fine naturale, il drop vero e proprio.

Questa operazione è propria del metodo successivo, che ha finalmente qualche riga di codice:

```
- (BOOL)performDragOperation:(id<NSDraggingInfo>)sender
{
    NSPasteboard * dragPasteboard;
    FileStruct     * fInfo ;
    NSArray         * pList ;
    NSString     * fName ;
    int            i, numelem ;

    // get the dragging pasteboard from the NSDraggingInfo
    dragPasteboard = [sender draggingPasteboard];
    pList = [ dragPasteboard propertyListForType: NSFilenamesPboardType ] ;
    numelem = [ pList count ] ;
    for ( i = 0 ; i < numelem ; i++ )
    {
        fName = [ pList objectAtIndex: i ] ;
        fInfo = [[ FileStruct alloc ] initTreeFromPath: fName ];
        [ dataSource addFileEntry: fInfo ];
    }
    return ( YES) ;
}
```

Comincio dalla fine, cioè dal valore di ritorno, che è YES se l'operazione si è conclusa felicemente, oppure NO se ci sono stati problemi (in questo secondo caso credo che il sistema riporti velocemente l'oggetto da dove era partito). Nel codice, ho utilizzato molte variabili per chiarezza espositiva, andando contro la mia natura di raggruppare tutto in poche istruzione. Allora: per prima cosa, recupero le informazioni dell'oggetto droppato col metodo `draggingPasteboard` a chi ha appena droppato l'oggetto. Poi ricavo i dati dalla pasteboard col metodo `propertyListForType`: Indico esplicitamente che mi devono essere restituiti dati relativi al nome di un file. Non chiedetemi perché uso questo metodo piuttosto che un altro; vi basti sapere che ho esaminato col debugger cosa

diavolo c'era dentro la `dragPasteboard`, e dopo avere interpretato un bel po' di numeri in esadecimale mi sono reso conto che era una property list che contenere un array... A questo punto dentro `pList` ho un array di nomi di file; non faccio altro che esaminarli uno ad uno ed aggiungerli alla lista degli elementi trattati dall'oggetto `dataSource`.

Concludo con l'ultimo metodo. L'operazione si è conclusa felicemente ed il sistema operativo mi dà un'occasione per portare via la spazzatura, spolverare e mettere un po' d'ordine.

```
- (void)concludeDragOperation:(id<NSDraggingInfo>)sender
{
    [ outlineView reloadData ];
    return ;
}
```

Ne approfitto allora per dire alla `outlineView` di ricaricare i dati, che se ne sono aggiunti di nuovi. In realtà, potevo farlo a conclusione del metodo precedente, e lasciare vuoto quest'ultimo metodo, ma mi dispiaceva avere un altro metodo vuoto...

Capitolo 020 - Ricominciamo!

Ricomincio a lavorare con Cocoa riprendendo in mano le vecchie cose, operando qualche modifica, ed accorgendomi che ci sono alcune cose nuove.

Sorgenti: Cosmetica e rilettura documentazione sparsa.

Primo inserimento: 2 settembre 2002

RILEGGENDO I FILE

Per riprendere confidenza con la faccenda dopo tutto questo tempo, decido di rivedere passo passo tutto il codice, vedendo se c'è qualcosa da aggiustare o che non capisco.

Riprendo allora la classe `LSFileInfo` e vedo se ci sono alcune cose da mettere a punto.

Vado nella documentazione e trovo quanto segue sugli attributi di un file:

Value	Type
NSFileSize (in bytes)	NSNumber
NSFileModificationDate	NSDate
NSFileOwnerAccountName	NSString
NSFileGroupOwnerAccountName	NSString
NSFileReferenceCount (number of hard links)	NSNumber
NSFileIdentifier	NSNumber
NSFileDeviceIdentifier	NSNumber
NSFilePosixPermissions	NSNumber
NSFileType	NSString
NSFileExtensionHidden	NSNumber containing a boolean
NSFileHFSCreatorCode	NSNumber containing an unsigned long
NSFileHFSTypeCode	NSNumber containing an unsigned long

I primi quattro attributi sono facili da capire: la dimensione del file stesso, quando è stato modificato l'ultima volta, gli attributi relativi ai diritti di proprietà dell'utente e del gruppo cui appartiene l'utente. Anche gli ultimi due attributi sono chiari: si tratta dei vecchi codici utilizzati in Mac OS 9 (e precedenti) per identificare il tipo ed il creatore del file. Queste informazioni erano utili al sistema operativo per mostrare la corretta icona nel Finder e per lanciare l'applicazione più appropriata per il trattamento del file. Queste stesse informazioni non sono più utili in Mac OS X in quanto sono ricavate per altra via (dall'estensione del file, per esempio). Quindi, il fatto che un file presenti questi due campi non vuoti è sintomo che deriva da una installazione precedente del sistema operativo (ma non necessariamente: le applicazioni carbonizzate mantengono tipo e creatore).

`NSFileType` e `NSPosixPermission` derivano dalla natura Unix di Mac OS X. In

particolare `NSFileType` permette di discriminare la natura del file. Ora, come dovrebbe essere noto, nel mondo Unix tutto è un file: sotto la nozione di file entrano tranquillamente le **directory** (un file che colleziona altri file), i **socket** (una porta di comunicazione con il mondo esterno), i **terminali** video (ancora, una porta di comunicazione con il mondo), e sono tutte trattate logicamente allo stesso modo. Un file si apre o si crea, si scrive o si legge, si chiude quando si ha finito. Ecco le possibilità:

String	Meaning
NSFileTypeUnknown	Unknown file type
NSFileTypeCharacterSpecial	Character special file
NSFileTypeDirectory	Directory
NSFileTypeBlockSpecial	Block special file
NSFileTypeRegular	Regular file
NSFileTypeSymbolicLink	Symbolic link
NSFileTypeSocket	Socket

In realtà, non sono riuscito a trovare un utilizzo sensato di questo campo. In effetti, a meno che non si voglia catalogare il disco su cui si trova il sistema operativo, si troveranno solamente file normali e directory, in quanto tutti i file di sistema sono abilmente mascherati dal Finder, che non li visualizza.

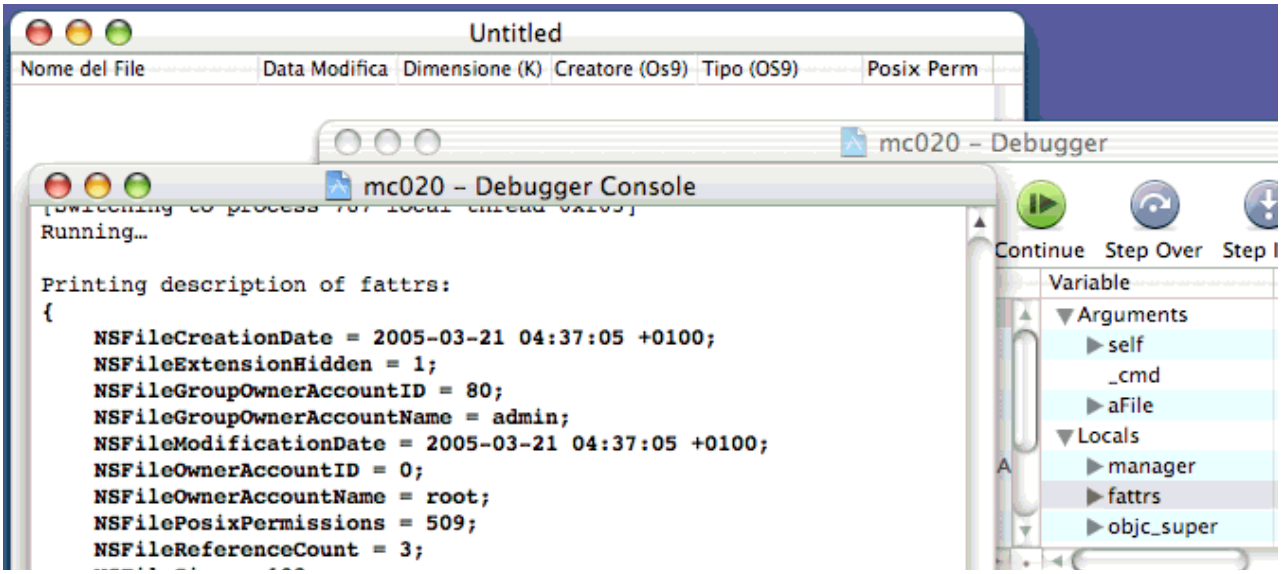
(Vedi comunque qui per altre considerazioni in merito).

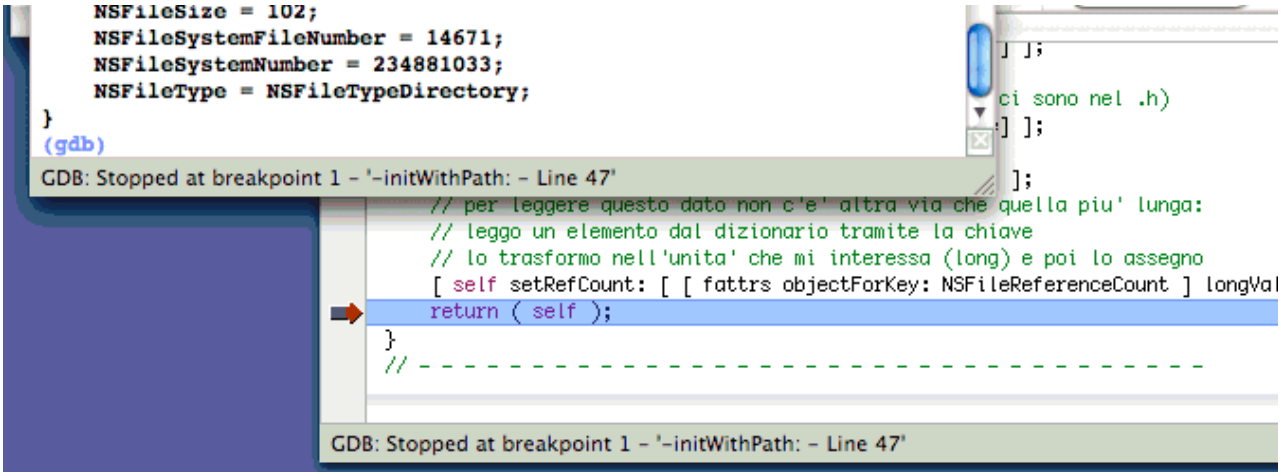
Mi rimangono tre campi. `NSFileReferenceCount` dice di quanti collegamenti quel file è riferimento. Si tratta di collegamenti 'hard' di stampo Unix, che sono simili, ma non la stessa cosa, degli alias cui si è abituati. Normalmente mi aspetto un valore di 1 (il file stesso), a mano che si tratti di file speciali di sistema.

(Sempre qui si discute dei link in Mac Os X).

Rimangono poi altri due campi, `NSFileIdentifier` e `NSFileDeviceIdentifier`, di cui nulla so (e che subito scopriremo non esistere...).

Imparato questo, mi limito ad aggiungere i campi che ancora non avevo considerato alla mia struttura di `LSFileInfo`, per renderla più completa.





Ovviamente, messo alla prova il codice con il debugger, vado incontro a delle sorprese. I due campi sopra sconosciuti sono veramente sconosciuti. Un semplice controllo col debugger mostra che il dizionario degli attributi restituito dall'istruzione

```
NSDictionary *fattrs = [manager fileAttributesAtPath: aFile
traverseLink:NO];
```

riporta quanto segue (per i curiosi, è l'applicazione '`Calcolatrice`'):

```
Printing description of fattrs:
{
    NSFileCreationDate = 2005-03-21 04:37:05 +0100;
    NSFileExtensionHidden = 1;
    NSFileGroupOwnerAccountID = 80;
    NSFileGroupOwnerAccountName = admin;
    NSFileModificationDate = 2005-03-21 04:37:05 +0100;
    NSFileOwnerAccountID = 0;
    NSFileOwnerAccountName = root;
    NSFilePosixPermissions = 509;
    NSFileReferenceCount = 3;
    NSFileSize = 102;
    NSFileSystemFileNumber = 14671;
    NSFileSystemNumber = 234881033;
    NSFileType = NSFileTypeDirectory;
}
```

Nessuna presenza dei due campi incriminati, sostituiti da due altrettanto sconosciuti `NSFileSystemFileNumber` e `NSFileSystemNumber`.

Riscrivo il mio codice per tenere conto di questi due campi, anche se al momento non so proprio che farci (ho il tragico sospetto che abbiano a che fare con il file system e la posizione del file all'interno del file system stesso: esaminando i numeri prodotti da altri file, `NSFileSystemNumber` non cambia, mentre `NSFileSystemFileNumber` varia).

(Cosa assolutamente non vera: su uno stesso volume ho trovato file con NSFileSystemNumber con valori differenti...).

FATTI E FORMATI NUOVI

Una cosa interessante è invece il fatto che sono comparsi (o non li avevo visti prima)

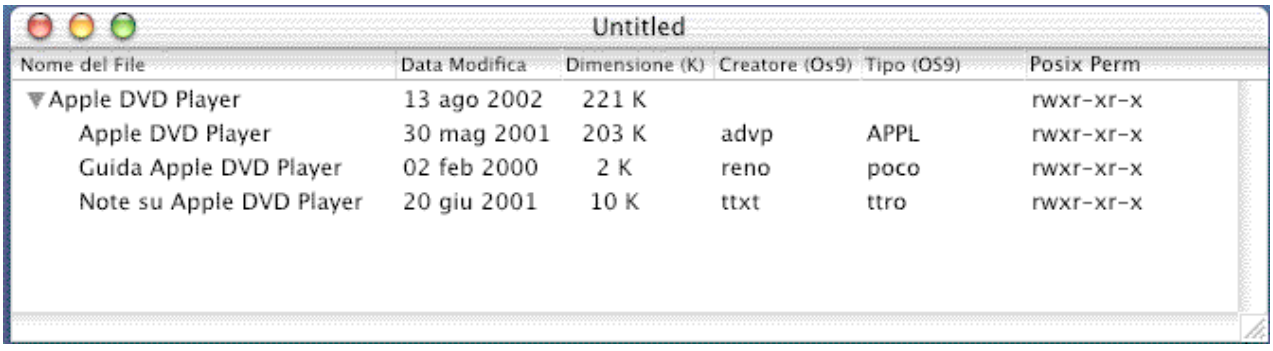
dei metodi di comodo per leggere elementi dal dizionario che contiene gli attributi del file. Alcuni di questi metodi non sono nemmeno documentati, ma presenti nel file di interfaccia `NSFileManager.h`. Con tutto ciò, e tenendo conto di tutta una serie di interventi cosmetici, ho in pratica ristrutturato i file `LSFileInfo.h` e `LSFileInfo.m`, cambiando in particolare le variabili d'istanza e i relativi metodi accessor. Ricordo che il tipo `OSType` altri non è che un `long`.

Da ricordare (altrimenti si perdono i dati salvando ed aprendo i file) i due metodi di codifica/decodifica. Questi ultimi due metodi dicono anche una cosa interessante: i file salvati in precedenza non sono più compatibili con questa versione, in quanto adesso la struttura dati salvata è piuttosto diversa. Volendo fare i precisi, si poteva prevedere all'inizio del file un codice di riconoscimento della versione di file, oppure si poteva cambiare il suffisso del file. Data la pochezza dell'applicazione, non me ne cruccio, ma un programma commerciale che si rispetti dovrebbe prevedere la lettura di ogni versione precedente di file salvato (vi vengono in mente programmi che così non fanno? Ci sono cattivi programmatori...).

La presenza (o meglio, la rinnovata considerazione) del campo `NSFilePosixPermissions` mi ha indotto a costruire una classe Formatter per questa rappresentazione. Per capirci, quando fate `ls -la` sul terminale, i permessi di lettura/scrittura sono rappresentati dalla stringa di 'r','w' e 'x' all'inizio di ogni riga (in realtà, gli ultimi nove caratteri di una stringa di dieci: il primo carattere indica che si tratta di un file normale con '-' piuttosto che di una directory con 'd' o altro):

```
host191-123:~/Temporaneo djzero$ ls -l
total 8
drwxr-xr-x   10 djzero djzero 340 Aug  9 19:01 Tevac widget
-rw-r--r--    1 djzero djzero 1809 Aug 8 21:42 alienrss.xml
```

Qui si dice che il file '`Tevac widget`' può essere letto ('r'), scritto ('w') ed eseguito ('x') dal possessore `djzero` (dando luogo ai primi tre caratteri 'rwx'), solo letto ed eseguito dal gruppo (e quindi 'r-x') e da tutti gli altri (chiudendo con 'r-x'). Queste stesse informazioni, in bella copia, si trovano nella finestra di Info che si può attivare dal Finder, nella parte relativa ai privilegi.



Detto questo, la classe formatter piglia il `long` che codifica questi privilegi e la trasforma proprio nella stringa di nove caratteri sopra considerata, tendo conto che ogni carattere deriva in pratica dal fatto che un bit sia ad uno piuttosto che a zero:

```
- (NSString *)
stringForObjectValue:    (id)anObject
```

```
{
    char        permstring[10] ;
    unsigned long     perm ;

    // controllo che l'oggetto sia un numero...
    if (![anObject isKindOfClass:[NSNumber class]]) {
        return nil;
    }
    // ricavo il long che mi da i permessi
    perm = [ anObject longValue ] ;
    // e poi e' tutto un sottile gioco di bit
    permstring[0] = ( perm & 0x0100 ) ? 'r' : '-' ;
    permstring[1] = ( perm & 0x0080 ) ? 'w' : '-' ;
    permstring[2] = ( perm & 0x0040 ) ? 'x' : '-' ;
    permstring[3] = ( perm & 0x0020 ) ? 'r' : '-' ;
    permstring[4] = ( perm & 0x0010 ) ? 'w' : '-' ;
    permstring[5] = ( perm & 0x0008 ) ? 'x' : '-' ;
    permstring[6] = ( perm & 0x0004 ) ? 'r' : '-' ;
    permstring[7] = ( perm & 0x0002 ) ? 'w' : '-' ;
    permstring[8] = ( perm & 0x0001 ) ? 'x' : '-' ;
    // chiudo la stringa C
    permstring[9] = 0 ;
    // converto la stringa C in NSString e la restituisco
    return ( [ NSString stringWithCString: permstring ] );
}
```

(Ci sono modifiche a quanto qui detto in questo capitolo).

DIMENSIONI STRANE

Passo adesso a gettare qualche luce (ma non risolvere) il problema delle dimensioni di un file. Tornando all'esempio precedente del player DVD, la dimensione riportata è di 208190. Niente di più sbagliato, almeno a sentire il Finder: costui, aprendo la finestra delle Info, dice che l'applicazione è di 480177 byte, più del doppio. Che fine hanno fatto i byte mancanti? Aprendo una finestra del Terminale, la dimensione riportata continua ad essere 200K (l'esempio è nel listato già visto sopra). Del resto, tornando al System 9,

la dimensione definitiva dell'applicazione è di 480K. Il fatto è che il Terminale e il metodo Cocoa riportano la sola dimensione della *data fork* (come si può evincere usando una qualsiasi utility che riporti queste informazioni, un nome a caso, File Buddy). Per i file nativi Mac Os X, che non risentono della 'vecchia' ripartizione tra data fork e resource fork, le dimensioni sono sempre corrette e congruenti tra tutti im etodi di ispezione.

Questa storia era vera quando scrivevo la prima volta questa pagina. La cosa era dovuta al fatto che molte applicazioni erano ancora scritte utilizzando Carbon, e costruite per girare sotto Classic e Mac OS X; attualmente, con Tiger, le applicazioni sono scritte sostanzialmente in Cocoa o equivalente, e il problema dello spazio mancante, dovuto essenzialmente alla resource fork, è scomparso; tuttavia, lascio qui considerazioni e figure come utile discussione.

Per risolvere il problema della dimensione, temo non ci sarà altra possibilità che ricorrere a Carbon, ovvero alle funzioni che si interfacciano direttamente con il sistema operativo.... (ma per ora lascio perdere).

La cosa avverrà in capitolo successivo.

Questa investigazione mi ha fatto venire in mente una simpatica modifica che posso fare al programma, affinché la dimensioni di una directory non siano proprio quelle fisiche (che in effetti interessano a pochi), ma che riporti la dimensione occupata da tutti i file in essa compresi (e in generale in tutta l'alberatura che sta nella directory stessa). Questo giochetto, a prima vista piuttosto noioso, si risolve in realtà elegantemente grazie alla procedura ricorsiva utilizzata per aggiungere file all'alberatura.

Già che ci sono, introduco una modifica nel meccanismo di filtraggio. A suo tempo, avevo escluso dall'esplorazione le directory il cui nome terminava con `.app`, ovvero, evitavo di esplorare il contenuto delle applicazioni. Questo fatto però comporta che la dimensione dell'applicazione è quella del file directory, che in genere non è quello che ci interessa. Sarebbe meglio conoscere per intero lo spazio occupato dall'applicazione e dai suoi file accessori. Quindi, mi ritrovo a dover considerare tutti i file, ma di procedere alla visualizzazione di solamente una parte (quei file il cui nome non comincia per '.'), e dell'espansione di una parte (si espandono le directory ma non le applicazioni).

A questo punto, è bene parlare dei `bundle`: in Mac OS X il concetto di bundle è assimilabile a quello di directory specializzata a contenere codice e altre risorse specifiche per l'esecuzione di quel codice. Molte cose di utilizzo comune sono dei bundle: ad esempio una applicazione è un bundle; una libreria è un bundle; un plug-in (che so, di Photoshop) è un bundle. Un bundle accorpa in una unico posto tutte le risorse necessarie al funzionamento del codice contenuto: in questo modo si evita di disperdere file in giro per il disco (ad esempio, i file di help), e le applicazioni si copiano e si installano senza eccessivi problemi.

Il mio problema è come riconoscere un bundle. Infatti, vorrei poter visualizzare nell'elenco il nome del bundle, ed evitare di espanderlo (ché il contenuto di una applicazione, con risorse e quant'altro raramente mi interessa), pur continuando a darne la dimensione, in byte, corretta (derivante dalla somma di tutto ciò che vi è dentro). Ebbene, la documentazione (`System Overview`, lo avete da qualche parte

all'interno della gerarchia `/Developer`) afferma che il Finder riconosce i bundle (e quindi mostra l'icona dell'applicazione piuttosto che l'icona di una cartella generica) da un bit nel suo database, o dall'estensione. Poiché a me il bit non lo fornisce nessuno (ed ignoro come si possa prelevare), tutto quello che posso fare è riconoscere l'estensione del nome del file: sono bundle tutte le cartelle il cui nome termina per `.app`, `.framework`, `.bundle`.

NUOVO INIZIALIZZATORE

Mi accingo a riscrivere (parte) del metodo `initTreeFromPath:` della classe `FileStruct` per tenere conto delle dimensioni delle directory.

```
- (id) initTreeFromPath: (NSString*) fullPath
{
    // mi metto via un file manager
    NSFileManager     *fileManager = [NSFileManager defaultManager];
    BOOL        isAdir, fileOK ;
    int        i ;

    // per prima cosa, inizializzo super
    [ super initWithPath: fullPath ] ;
    // e adesso, se il file puntato e' una directory, espando
    fileOK = [ fileManager fileExistsAtPath: fullPath isDirectory: & isAdir
];
    // espando sempre, purche' il file esista e sia una directory
    if ( fileOK && isAdir)
    {
        // variabile per tenere traccia della dimensione parziale
        long    dirSize = 0;
        // vettore destinato a contenere i file
        NSMutableArray     *tmpfileList ;
        // l'elenco dei file
             NSArray  *dirContent  =  [  fileManager  directoryContentsAtPath:
fullPath ];
        // conto quanti file ci sono all'interno
        int numFile = [ dirContent count ];
        // costruisco un vettore di dimensione adatta
        tmpfileList = [[ NSMutableArray alloc ] initWithCapacity: numFile ]
;
        // adesso, per ogni elemento, costruisco l'albero e poi lo aggiungo
        for ( i = 0; i < numFile; i++)
        {
            FileStruct    * tmpFile ;
            NSString     * myfile = [ dirContent objectAtIndex: i ] ;

            // costruisco l'albero
            tmpFile = [[FileStruct alloc] initTreeFromPath:
                [ fullPath stringByAppendingPathComponent: myfile]] ;
            // aggiungo la dimensione (cumulata) alla directory corrente
            dirSize += [ tmpFile fileSize ];
            // aggiungo il file alla lista
            [tmpfileList addObject: tmpFile ];
        }
        // assegna al file la nuova dimensione
        [ self setFileSize: dirSize ];
        // adesso copio il vettore con la giusta dimensione
              fileList  =  [[  NSMutableArray  alloc  ]  initWithCapacity:  [
tmpfileList count] ];
        [ fileList setArray: tmpfileList ];
```

```
    }
    else
    {
        // va beh, e' un file normale
        fileList = nil ;
    }
    return ( self );
}
```

Il codice mi pare sufficientemente commentato da non aver bisogno di ulteriore commento. Vogliate solo apprezzare l'eleganza della ricorsione, che calcola correttamente le dimensioni cumulate delle directory comunque annidate siano l'una dentro l'altra (gioie dell'esplorazione in profondità degli alberi).

Compilo il tutto, correggo i molti errori (che voi qui non vedete, visto che li ho già tutti corretti), ed eseguo. In effetti la cosa funziona.

Beh, non proprio. Quando procedo alla cancellazione di qualche file, la dimensione della directory che lo contiene non è aggiornata... Dovrei sottrarre alla dimensione della directory le dimensioni del file cancellato. In realtà non so se sia una buona cosa: sto tenendo un catalogo, e la dimensione della directory non è cambiata solo perché io ho cancellato dal catalogo un file... Decido per semplicità (o pigrizia) di lasciare le cose come stanno.

Con questi, ed altri ritocchi cosmetici, ho finito anche con `FileStruct` (che ne è uscita un po' più semplice).

NUOVI FILTRI

A questo punto, il grosso lavoro di filtro deve essere svolto dalla classe `LSDataSource`. Ripeto un'altra volta i due problemi:

1. evitare di visualizzare alcune categorie di file; li chiamo **dotFiles** per ricordare che sono in pratica i file il cui nome comincia con il carattere punto.

2. evitare di espandere i bundle; sono i file che hanno delle particolari estensioni.

Con questo in mente, i metodi della classe hanno subito una pesante revisione, che ha portato all'introduzione di quattro interessanti funzioni:

```
BOOL            skipDotFile ( NSString * fileName ) ;
BOOL            checkIfBundleDirectory( NSString * fileName );
short         countNormalFiles( FileStruct * directory );
FileStruct *    getNormalFile ( FileStruct * directory, short index ) ;
```

La prima funzione è il filtro sui dotFiles; in pratica, esamina il nome del file e decide se il file partecipa alla festa oppure è lasciato alla porta:

```
BOOL
skipDotFile ( NSString * fileName )
{
    const char    * fn ;        // nome del file in C
    // trasformo il nome del file in una stringa C
```

```
    fn = [ fileName cString] ;
    // perche' poi cosi' escludo quelli che cominciano con '.'
    if (*fn == '.' )
        return ( TRUE ) ;
    // escludo anche i file che si chiamano "Icon\r"
    // non chiedete il perche' delle '\r'
    if ( [ fileName isEqual: @"Icon\r"] )
        return ( TRUE ) ;
    // ... altre condizioni di esclusione
    // - - - - - - - - - - - - - - - - -
    return ( FALSE );
}
```

Nel fare le prove, mi sono imbattuto nel problema dei file Icona. Come è noto, nel MacOs9 si possono attribuire icone fantasiose alle cartelle; queste icone sono mantenute in un file (nascosto) all'interno della cartella stessa. Per evitare di mostrare questi file (di scarso interesse), li considero dotFiles e li elimino dalla visualizzazione. La cosa buffa (che non capisco) è il nome del file, che io pensavo fosse "Icon", ed invece possiede un carattere di ritorno carrello alla fine del nome (misteri dei sistemi operativi).

La seconda funzione verifica se una directory è da considerare un bundle o meno. Molto semplicemente, si guarda il nome:

```
BOOL
checkIfBundleDirectory( NSString * fileName )
{
    // per ricavare l'estensione, c'e' un metodo apposito
    if ( [ fileName hasSuffix: @".app"] )
        return ( YES );
    if ( [ fileName hasSuffix: @".bundle"] )
        return ( YES );
    if ( [ fileName hasSuffix: @".framework"] )
        return ( YES );
    return ( NO );
}
```

Le altre due funzioni trattano il problema posto da `NSOutlineView`. Queste classi, per visualizzare al proprio interno la lista gerarchica di elementi, necessitano di sapere per ogni elemento quanti sottoelementi possiede (quindi, di ogni directory, quanti sono i file contenuti nella directory) e di poter accedere all'elemento i-esimo contenuto. La cosa è complicata dal fatto che nella conta sono considerati anche i dotFiles da non visualizzare. Per tanto, la terza funzione conta i file presenti in una directory saltando i dotFiles:

```
short
countNormalFiles( FileStruct * directory )
{
    short    i, count ;

    count = 0 ;    // all'inizio, nessuno
    for ( i = 0 ; i < [directory numOfFiles] ; i++ )
    {
        // se devo saltarlo, vado avanti senza incrementare il contatore
        if ( skipDotFile( [[directory getFileAtIndex:i] fileName ] ) )
            continue ;
        // se arrivo qui, e' un file normale, incremento il contatore
        count += 1 ;
    }
```

```
    return ( count );
}
```

La quarta funzione restituisce l'i-esimo file nella directory, continuando a saltare i dotFiles:

```
FileStruct *
getNormalFile ( FileStruct * directory, short index )
{
    short    i, count ;

    count = 0 ;    // contatore corrente
    for ( i = 0 ; i < [directory numOfFiles] ; i++ )
    {
        FileStruct * currFile ;
        // recupero il file
        currFile = [directory getFileAtIndex:i] ;
        // se devo saltarlo, passo avanti
        if ( skipDotFile( [ currFile fileName ] ) )
            continue ;
        // sono arrivato all'indice richiesto ?
        if ( count == index )
            // si, restituisco l'elemento corrente
            return ( currFile );
        // se arrivo qui, non ho ancora raggiunto l'elemento
        count += 1 ;
    }
    // ovviamente, qui non dovrei mai arrivare...
    return ( nil) ;
}
```

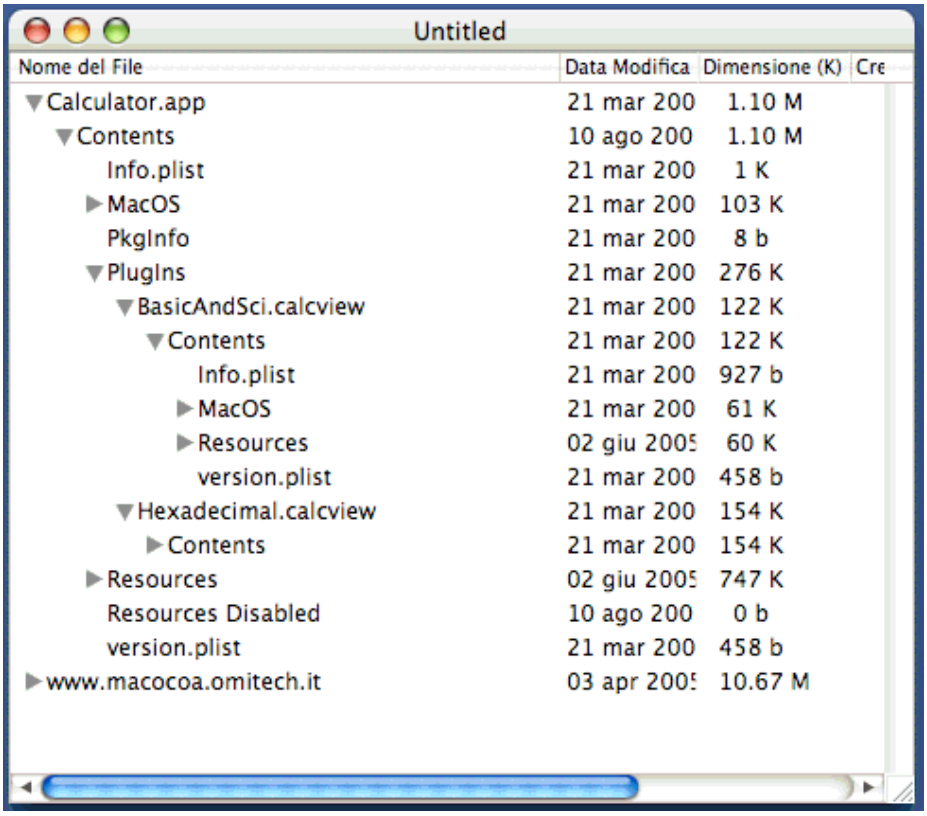
A questo punto, i due metodi necessari a `NSOutlineview` diventano

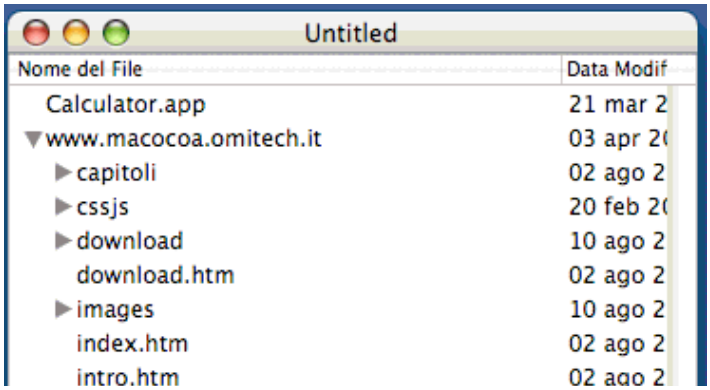
```
- (int)
outlineView:            (NSOutlineView *)outlineView
    numberOfChildrenOfItem:    (id)item
{

    // se l'item e' nil, stiamo parlando della radice
    if (item == nil)
            // dico quindi che ci sono tanti elementi quanti presenti nel
vettore
        return( [ [self startPoint] count ]);
    // se arrivo qui, mi si chiede quali figli ha un file
    // tratto subito i file normali, che non hanno figli
    if ( [item numOfFiles] == 0 )
        return ( 0 ) ;
    // se arrivo qui, ho una directory
    // devo eliminare dal computo i dotFiles
    // ed alora, mi tocca esaminare tutti i file e contare quelli
    // che mi interessano
    return ( countNormalFiles( item) ) ;
}

- (id)
outlineView:        (NSOutlineView *) outlineView
    child:        (int) index
    ofItem:        (id) item
{
    if (item == nil)
        return( [ [self startPoint] objectAtIndex: index ]);
```

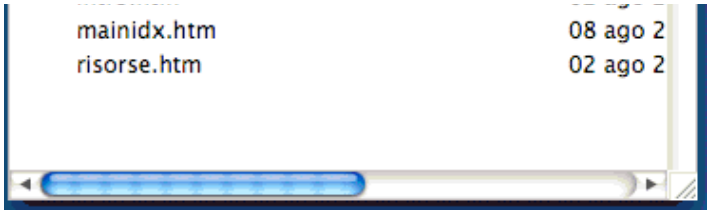
```
    // devo saltare i dotfiles
    return ( getNormalFile ( item, index ) );
}
```

Infine, per vedere se un elemento di `NSOutlineview` è espandibile o meno, si utilizza:

```
- (BOOL)
outlineView:            (NSOutlineView *)outlineView
    isItemExpandable:    (id)item
{
    if (item == nil)
        return ( YES ) ;
    // se non ha figli, e' un file, non si espande
    if ( [item numOfFiles] == 0 )
        return ( NO ) ;
    // se arrivo qui, l'item ha figli
    // espando allora i non bundle
    return (! checkIfBundleDirectory([ item fileName ]) ) ;
}
```







Nelle due immagini, l'aspetto prima e dopo la cura.

Non sono ancora soddisfatto. Non mi piace inserire direttamente nel codice una serie di scelte che in realtà sono proprie dell'utente, ovvero, se visualizzare o meno i bundle e se visualizzare o meno i dotFiles. Per questo motivo, voglio realizzare una finestra con cui l'utente possa assegnare delle preferenze di visualizzazione.

La politica di funzionamento sarà quindi che in sede di lettura dati (cioè, quando si aggiungono file al catalogo) sono comunque conteggiati tutti i file; in sede di visualizzazione, ci saranno dei filtri per visualizzare o meno determinati file o per espandere o meno determinate directory. Ma di questo ne parlo il prossimo capitolo, cui vi rimando anche per tutti i file ed il progetto.

Capitolo 021 - Preferirei di no

Scopo del capitolo è di costruire un meccanismo (finestra e messaggistica) per l'impostazione di un sistema di preferenze.

Sorgenti: Documentazione ed esempi di Apple.

Primo inserimento: 2 settembre 2002

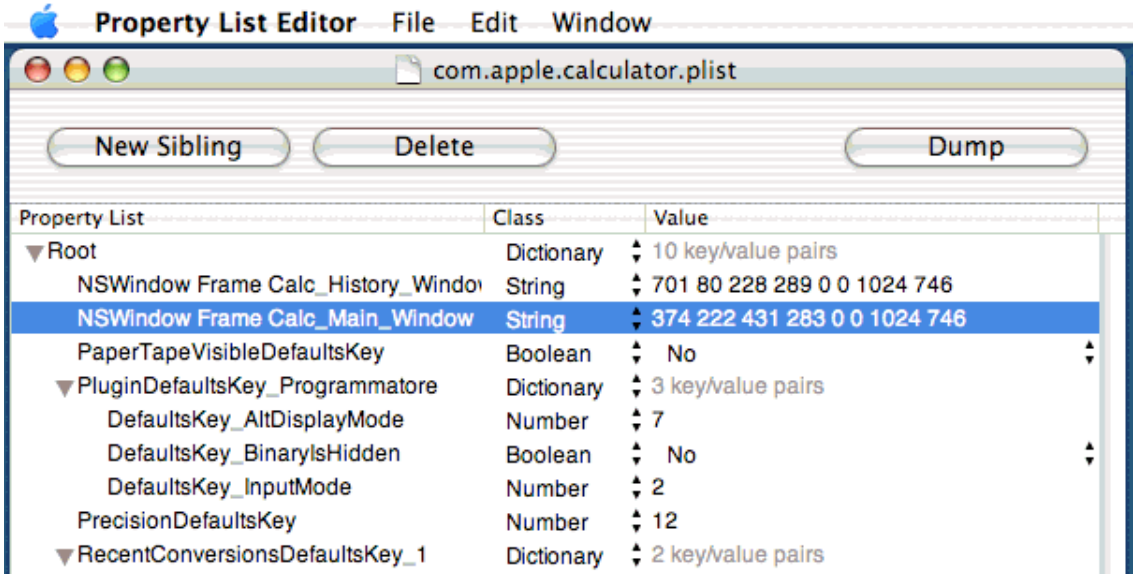
Il codice di questo capitolo sarà stravolto successivamente.

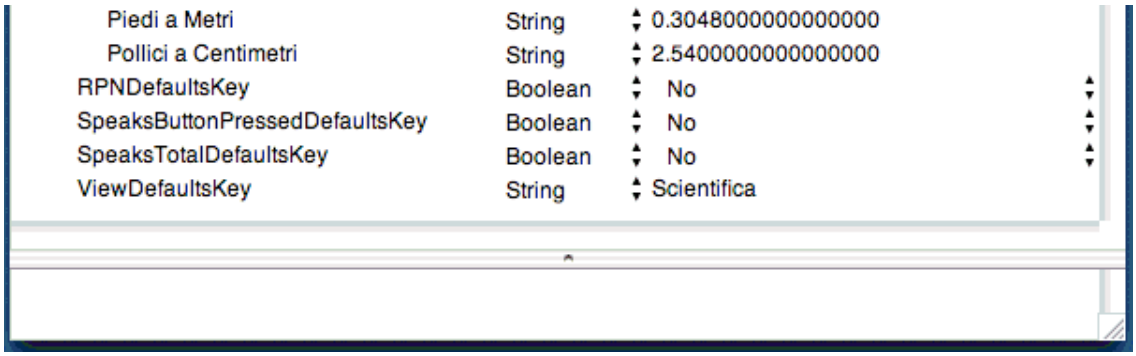
Le Preferenze

Il problema lasciato in sospeso il capitolo precedente era di poter determinare il comportamento della `NSOutlineView` quando si tratta di espandere i bundle e di visualizzare i dotFiles. Piuttosto che determinare tale comportamento direttamente all'interno del programma, preferisco fare in modo che sia l'utente stesso a decidere, utilizzando una finestra di Preferenze.

In primo luogo, scopro che le Preferenze, nel MacOSX, sono memorizzate in un file di banale testo. I documenti che raccolgono le preferenze, diversi per ogni utente, sono raccolti nella directory `~/Library/Preferences` (ricordo che la tilde rappresenta di directory principale dell'utente, nel mio caso `/Users/djzero00`). Tali documenti si presentano sotto forma di documento XML (di cui ignoro al momento natura, sintassi e semantica). Tuttavia, pragmaticamente, scopro che esiste un programma, il `Property List Editor`, presente in qualità di tool di sviluppo (nella directory `/Developer/Applications`) che fa proprio al caso mio.

A partire da un certo punto in poi (che mi sono perso, ma potrebbe essere Tiger), i file delel preferenze non sono più sotto forma testuale, ma sono diventati file binari. L'applicazione Property List Editor continua a leggerli tranquillamente, ma non è più possibile aprirli con un normale editor di testo (o meglio, si possono aprire, ma non ci si capisce alcunché).





Aprendo a caso un file di preferenze (Calcoaltrice, ad esempio), noto che il testo prima pieno di tag, codici ed altri numeri bizzarri, risulta adesso molto più strutturato (ed ancora abbastanza incomprensibile, ma essendo la prima volta che vedo un file di preferenze la cosa non mi rattrista). Una cosa da notare è il nome del file delle preferenze, che ricorda un po' un indirizzo internet a rovescio (nel caso `com.apple.calculator.plist`).

Per gestire il meccanismo delle preferenze, Cocoa mette a disposizione una classe, `NSUserDefaults`, che svolge la maggior parte del lavoro. Andando a vedere la documentazione relativa, esiste una lunga spiegazione di quali sono i valori di default, come sono cercati nei vari database, eccetera.

Alla prova dei fatti, le cose sono in realtà piuttosto semplici. Basta una istruzione:

```
NSUserDefaults      *defaults = [NSUserDefaults standardUserDefaults];
```

Con questa, costruisco un oggetto della classe `NSUserDefaults` che gestisce il meccanismo. Da quest'oggetto, attraverso metodi appositi, sono in grado di leggere, scrivere e gestire in generale le varie preferenze presenti. Ora, una preferenza può essere un oggetto di una di queste classi: `NSData`, `NSString`, `NSNumber`, `NSDate`, `NSArray`, or `NSDictionary`; ogni oggetto è identificato da una chiave (una specie di dizionario, insomma). Di più, ci sono metodi di convenienza per leggere e scrivere numeri interi, floating point, booleani, vettori, per cancellare un elemento, eccetera. Tutto molto semplice; la cosa più complicata è gestire l'intera faccenda all'interno del programma.

Infatti, all'interno del programma esistono diversi insiemi concettualmente differenti di Preferenze (ho definito degli oggetti della classe `NSDictionary`). Il primo insieme di preferenze, che chierò `defFile`, è quello stabilito dal file delle preferenze; c'è poi l'insieme `defCode`, ovvero i valori stabiliti direttamente dal codice (quelli attivi in assenza del file di preferenze, o la prima volta che è eseguito il programma da un nuovo utente...). I valori defCode sono ovviamente sovrascritti da defFile, se all'interno delle Preferenze sono appunto presenti nuovi valori.

Poi abbiamo due insiemi più volatili: l'insieme `defCurr` contiene i valori correnti delle Preferenze; l'insieme `defDisp` i valori delle Preferenze in corso di manipolazione dall'utente.

All'inizio dei tempi, all'insieme defCurr sono assegnati i valori di defCode; poi utilizzo il meccanismo fornito da `NSUserDefaults` per ricavare i valori di defFile e

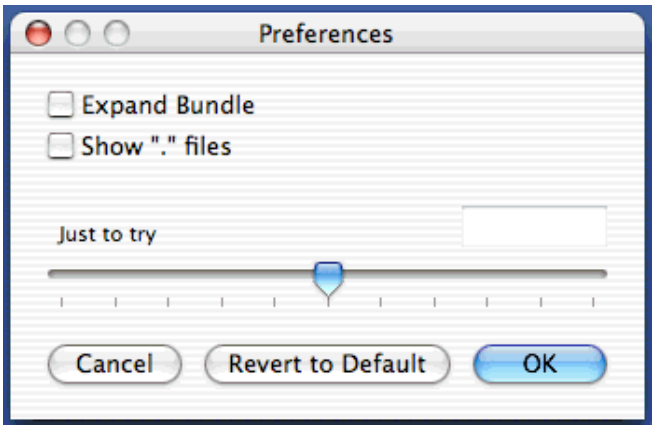
sovrascrivere i corrispondenti campi di defCurr. Ad un certo punto, l'utente aprirà la finestra per la gestione delle preferenze. In quel momento assegno a defDisp i valori di defCurr, e con defDisp mostro la finestra all'utente. Quando costui gioca con la finestra, modifica i valori di defDisp, fino a che non decide cosa fare. La scelta è ridotta a tre possibilità:

1. l'utente decide che i valori impostati vanno bene e da l'ok alla finestra; in tal caso i valori di defDisp sovrascrivono defCurr e contemporaneamente sono aggiornati i valori di defFile.

2. l'utente decide di aver pasticciato abbastanza e vuole tornare ai valori di default precedenti: su defDisp sono nuovamente copiati i valori di defCurr (e la finestra delle preferenze aggiornata di conseguenza).

3. l'utente ha pasticciato abbastanza e decide di lasciar perdere, chiudendo la finestra senza apportare modifiche. I valori di defDisp sono buttati via, e quelli di defCurr non hanno subito mutamenti.

LA FINESTRA E LA CLASSE



Detto questo, passo a descrivere la finestra delle preferenze: ci sono due pulsanti, uno per selezionare l'espansione dei bundle e l'altro per la visualizzazione del dotFiles. Per mia istruzione, ho aggiunto uno slider senza ragione pratica, ma giusto per aggiungere qualcosa ad una finestra altrimenti piuttosto spoglia ed un valore non booleano al sistema di preferenze.

Tutto ciò giustifica questo primo pezzo di codice, in cui sono presenti le variabili d'istanza ed alcuni metodi interessanti della classe `Preferences`:

```
@interface Preferences : NSObject
{
    IBOutlet NSButton    *expandBundleButton;
    IBOutlet NSButton    *dummy1Button;
    IBOutlet NSSlider    *dummy2Slider;

    // Current, confirmed values for the preferences: def-curr
    NSDictionary         *defCurrValues;
    // Values read from preferences at startup: def-code + def-file
    NSDictionary         *defFileValues;
    // Values displayed in the UI: def-disp
```

```
    NSMutableDictionary     *defDispValues;
}


- (void) userSelectUpdate:(id)sender;
- (void) userSelectRestore:(id)sender;
- (void) userSelectCancel:(id)sender;
+ (Preferences *)sharedInstance;
- (void)showPanel:(id)sender;
```

I tre outlet servono per accedere ai vari elementi dell'interfaccia, per impostare e leggere il valore; i primi tre metodi sono le tre Action corrispondenti ai tre pulsanti 'ok', 'Restore defaults' e 'cancel'.

Per comodità ho poi definito poi tre chiavi di accesso alle preferenze

```
#define     keyExpandBundle     @"ExpandBundle"
#define     keyShowDotFiles     @"ShowDotFiles"
#define     keyDummy02          @"Dummy02"
```

Tuttavia, la prima cosa da fare, è aprire la finestra. In Interface Builder ho quindi aperto il file principale MainMenu.nib e mi sono posto il problema di come collegare la voce Preferences del menu dell'applicazione. Al contrario della palette, che può essere aperta solo in presenza di una finestra di catalogo (il che spiega il collegamento al First Responder della voce), qui ho fatto una cosa diversa (non ho inventato nulla, ho guardato l'esempio TextEdit presente nella cartella /Developer/Examples): ho aggiunto un'istanza della classe Preferences direttamente in IB. In questo modo ho collegato la voce di menu direttamente all'istanza di Preferences invocando l'action ShowPanel. In sede di inizializzazione, infatti, faccio in modo che sia costruita ma non visualizzata la finestra, che poi appare e scompare come comandato dall'utente. La presenza di una istanza unica condivisa dall'applicazione, e l'uso di metodi (falsamente) di classe fanno in modo che la finestra sia unica.

(Il meccanismo di gestione della finestra sarà profondamente modificato in un capitolo successivo).

LETTURA E SCRITTURA DELLE PREFERENZE

Ho quindi il metodo di inizializzazione:

```
// metodo di inizializzazione (che legge i defaults)
- (id) init
{
    // mantengo unica la finestra: se l'ho gia' creata, faccio nulla
    if (sharedInstance)
    {
        [self dealloc];
    }
    else
    {
        // costruisco l'istanza delle preferenze
        [super init];
        // carico le preferenze dal file
                defCurrValues  =  [[[self  class]  preferencesFromDefaults]
copyWithZone:[self zone]];
```

```
        // sono sia def-curr che def-file
        defFileValues = [defCurrValues retain];
        // assegno gli stessi valori anche a def-disp
        [self discardDisplayedValues];
        sharedInstance = self;
        // inizializzo queste due variabili che mi servono poi
        lsPrefsYes = [[NSNumber alloc] initWithBool:YES];
        lsPrefsNo = [[NSNumber alloc] initWithBool:NO];
    }
    return sharedInstance;
}
```

La parte interessante è il caricamento dei defaults dal file apposito. Il metodo è di classe, giusto per mantenere unico il punto di accesso al file delle preferenze:

```
+ (NSDictionary *)
preferencesFromDefaults
{
    // pesco il file dei defaults e costruisco un dizionario
    NSUserDefaults     *defaults = [NSUserDefaults standardUserDefaults];
                NSMutableDictionary    *dict    =    [NSMutableDictionary
dictionaryWithCapacity:10];
    // inserisco il booleano che dice se espandere o meno i bundle
    readBoolDefault( dict, defaults, keyExpandBundle );
    // booleano per mostrare o meno i dotFiles
    readBoolDefault( dict, defaults, keyShowDotFiles );
    // un dummy intero
    readIntDefault( dict, defaults, keyDummy02 );

    return dict;
}
```

dove sono utilizzate le due funzioni accessorie seguenti:

```
void
readBoolDefault ( NSMutableDictionary * locDict, NSUserDefaults * locDef, id
key )
{
    id    obj ;
    // vedo se nei defualts c'e' un valore per la chiave
    obj = [ locDef objectForKey: key] ;
    // se c'e'
    if ( obj )
    {
          // recupero dal dizionario il booleano, poi lo converto in un
NSNumber
        // per reinfilarlo nel dizionario
                    [locDict  setObject:  [NSNumber  numberWithBool:[locDef
boolForKey:key]] forKey:key] ;
    }
    else
    {
           // recupero dal dizionario def-code il valore e lo metto del
dzionario
        [locDict setObject: [defaultValues() objectForKey:key] forKey:key] ;
    }
}
void
readIntDefault ( NSMutableDictionary * locDict, NSUserDefaults * locDef, id
key )
{
```

```
    id    obj ;
    // vedo se nei defaults c'e' un valore per la chiave
    obj = [ locDef objectForKey: key] ;
    // se c'e'
    if ( obj )
    {
            // recupero dal dizionario il booleano, poi lo converto in un
NSNumber
        // per reinfilarlo nel dizionario
                    [locDict  setObject:  [NSNumber  numberWithInt:[locDef
integerForKey:key]] forKey:key] ;
    }
    else
    {
            // recupero dal dizionario def-code il valore e lo metto del
dzionario
        [locDict setObject: [defaultValues() objectForKey:key] forKey:key] ;
    }
}
```

Da notare il codice per la lettura del default (vorrei che apprezzaste il fatto che stavolta il codice si dilunga, costruisce variabili intermedie, è commentato, insomma è più leggibile, anche per me): se all'interno di defFile c'è l'oggetto corrispondente ad una data chiave, viene utilizzato tale valore; altrimenti, si pesca da un dizionario inserito direttamente nel codice:

```
static NSDictionary *defaultValues() {
    static NSDictionary *dict = nil;
    if (!dict) {
    dict = [[NSDictionary alloc] initWithObjectsAndKeys:
        [NSNumber numberWithBool:NO], keyExpandBundle,
        [NSNumber numberWithBool:NO], keyShowDotFiles,
        [NSNumber numberWithInt:50], keyDummy02,
        nil];
    }
    return dict;
}
```

Totalmente simmetrici sono i metodi e le funzioni per la scrittura di questi default:

```
+ (void)
savePreferencesToDefaults:(NSDictionary *)dict
{
    // recupero il file dei defaults
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    // inserisco ordinatamente i vari elementi
      writeBoolDefault(  [[self  sharedInstance]  getPreferences],  defaults,
keyExpandBundle );
      writeBoolDefault(  [[self  sharedInstance]  getPreferences],  defaults,
keyShowDotFiles );
       writeIntDefault(  [[self  sharedInstance]  getPreferences],  defaults,
keyDummy02 );
}

void
writeBoolDefault( NSDictionary * locDict, NSUserDefaults * locDef, id key )
{
    // se il valore def-code e' lo stesso del valore def-file
        if  (  [  [defaultValues()  objectForKey:  key]  isEqual:  [locDict
objectForKey: key] ] )
```

```
    {
        // tolgo addirittura il valore dal file;
        // in questo modo lo tengo pulito e conciso
        [locDef removeObjectForKey: key];
    }
    else
    {
        // altrimenti, lo scrivo nel file
           [locDef setBool:[[locDict objectForKey: key] boolValue] forKey:
key];
    }
}

void
writeIntDefault( NSDictionary * locDict, NSUserDefaults * locDef, id key )
{
    // se il valore def-code e' lo stesso del valore def-file
         if ( [ [defaultValues() objectForKey: key] isEqual: [locDict
objectForKey: key] ] )
    {
        // tolgo addirittura il valore dal file;
        // in questo modo lo tengo pulito e conciso
        [locDef removeObjectForKey: key];
    }
    else
    {
        // altrimenti, lo scrivo nel file
          [locDef setInteger:[[locDict objectForKey: key] boolValue] forKey:
key];
    }
}
```

Qui ho aggiunto la complicazione inutile di eliminare una voce dal defFile nel caso in cui coincida con il valore stabilito direttamente dal codice. In questo modo il file delle preferenze si mantiene più piccolo e conciso, e solamente con i valori differenti da quelli stabiliti dal programmatore con coscienza.

LE AZIONI

Adesso sono comprensibili i tre metodi 'action' in corrispondenza della scelta dell'utente:

```
- (void) userSelectUpdate:(id)sender
{
    // devo recuperare i valori dalla finestra
    [defDispValues setObject:
        ([expandBundleButton state] ? lsPrefsYes : lsPrefsNo)
        forKey:keyExpandBundle];
    [defDispValues setObject:
        ([dummy1Button state] ? lsPrefsYes : lsPrefsNo)
        forKey:keyShowDotFiles];
    [defDispValues setObject:
        ([NSNumber numberWithInt:[ dummy2Slider intValue ]])
         forKey:keyDummy02];
    // recuperati i valori; li assegno ai correnti
    [ self commitDisplayedValues ] ;
    // salvo le preference su file
    [ Preferences saveDefaults ];
    // e per finire nascondo la finestra
    [[expandBundleButton window] setIsVisible: FALSE ];
```

```
}
```

Qui leggo lo stato dei tre controlli all'interno della finestra, uso `commitDisplayedValues` per aggiornare i valori, salvo i valori nel file e nascondo la finestra.

```
- (void)commitDisplayedValues
{
    if (defCurrValues != defDispValues)
    {
    [defCurrValues release];
    defCurrValues = [defDispValues copyWithZone:[self zone]];
    }
}
```

Più concisa la situazione in caso di restore:

```
// scelta restore sulla finestra: rileggo le prefs
- (void) userSelectRestore:(id)sender
{
    [ self discardDisplayedValues ] ;
}
```

dove tutto il lavoro è svolto dal metodo `discardDisplayedValues` :

```
- (void) discardDisplayedValues
{
    if (defCurrValues != defDispValues)
    {
        [defDispValues release];
        defDispValues = [defCurrValues mutableCopyWithZone:[self zone]];
        [self updatePrefWindow];
    }
}
```

Infine il metodo associato al pulsante `Cancel` semplicemente butta via i valori eventualmente aggiornati dall'utente (e poi chiude la finestra):

```
- (void) userSelectCancel:(id)sender
{
    if (defCurrValues != defDispValues)
    {
        [defDispValues release];
        defDispValues = [defCurrValues mutableCopyWithZone:[self zone]];
    }
    [[expandBundleButton window] setIsVisible: FALSE ];
}
```

I due metodi che interagiscono direttamente con la finestra sono i seguenti:

```
- (void)
updatePrefWindow
{
    // il solito trucco per vedere se la finestra c'e'
    if (!expandBundleButton) return ;
    // recupero il valore, ed imposto il bottone di conseguenza
                            [expandBundleButton      setState:[[defDispValues
```

```
objectForKey:keyExpandBundle] boolValue] ? 1 : 0];
      [dummy1Button setState:[ [defDispValues objectForKey:keyShowDotFiles]
boolValue] ? 1 : 0 ] ;
       [dummy2Slider setIntValue:[ [defDispValues objectForKey:keyDummy02]
intValue ] ];
}
```

Per aggiornare la finestra, piglio i valori di defDisp (quelli da visualizzare!) ed imposto lo stato dei controlli di conseguenza.

Invece, per mostrare la finestra, distinguo se si tratta della prima volta o meno:

```
- (void)
showPanel:(id)sender
{
    // infame trucco: se la finestra non e' stata creata, l'outlet
    // non e' stato assegnato...
    if (!expandBundleButton)
    {
        // carico la finestra usando il nome del nib-file
        if (![NSBundle loadNibNamed:@"prefsPane" owner:self])
        {
            // sono in errore... forse dovrei fare uqalcosa di meglio...
            NSLog(@"Failed to load Preferences.nib");
            NSBeep();
            return;
        }
        // non voglio che compaia nella lista delle finestre
        [[expandBundleButton window] setExcludedFromWindowsMenu:YES];
        // questo, lo ignoro
        [[expandBundleButton window] setMenu:nil];
        // inserisco i def-curr nella finestra
        [self updatePrefWindow];
        // la piazzo al centro dello schermo
        [[expandBundleButton window] center];
    }
    // porto la finestra davanti a tutti
    [[expandBundleButton window] makeKeyAndOrderFront:nil];
}
```

Mancano un po' di metodi aggiuntivi di servizio, che vi lascio esaminare con comodo.

L'USO DELLE PREFERENZE

A questo punto, il meccanismo delle preferenze è attivo e funzionante; occorre semplicemente utilizzarlo. Per comodità, c'è il seguente metodo (fintamente) di classe

```
+ (id)
objectForKey:(id)key
{
    return [[[self sharedInstance] getPreferences] objectForKey:key];
}
```

che utilizza questo metodo accessorio:

```
- (NSDictionary *)
getPreferences
```

```
{
    return defCurrValues;
}
```

In questo modo, per accedere ad un valore di preferenze, devo semplicemente scrivere qualcosa del tipo:

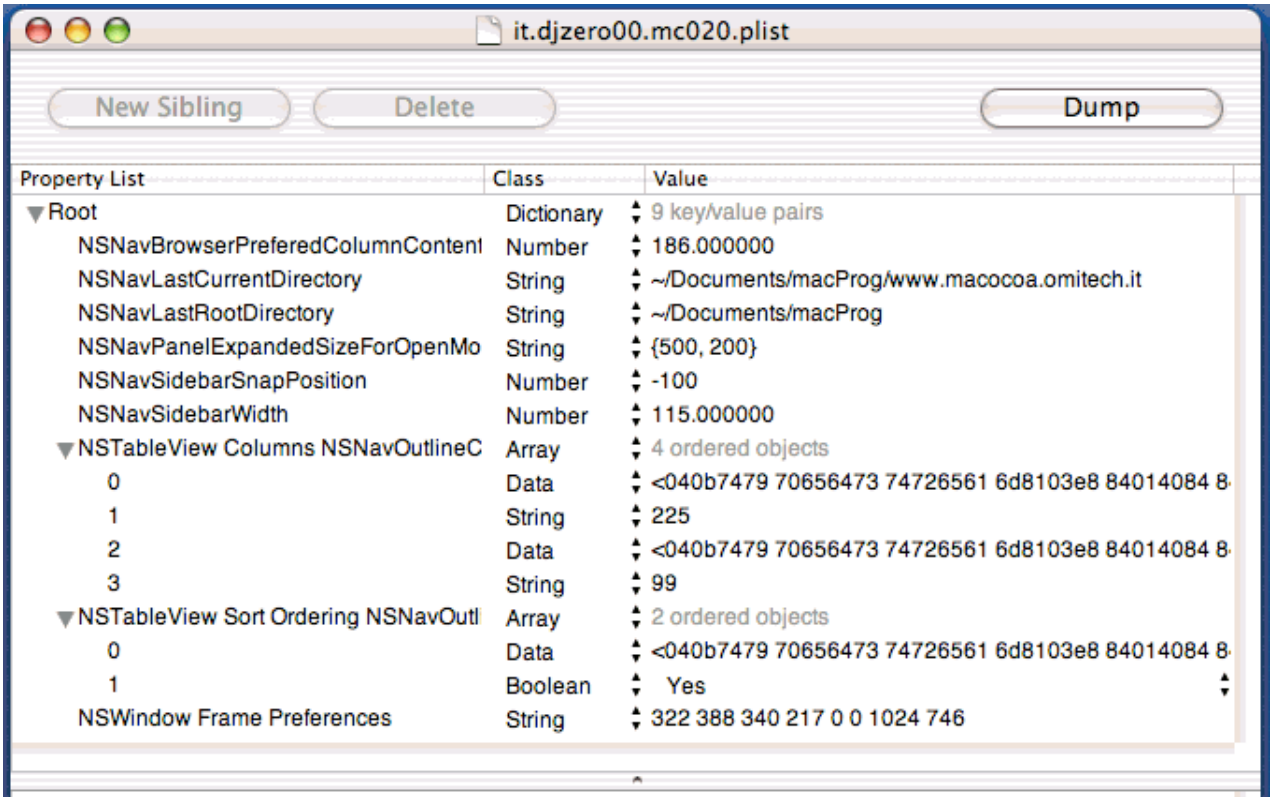
```
[Preferences objectForKey:keyDummy02] intValue]
```

per avere bello pronto il valore richiesto.

Devo solo riscrivere alcuni metodi della classe `LSDataSource`, con un codice che si spiega da sé:

```
- (int)
outlineView:            (NSOutlineView *)outlineView
    numberOfChildrenOfItem:    (id)item
{

    // se l'item e' nil, stiamo parlando della radice
    if (item == nil)
          // dico quindi che ci sono tanti elementi quanti presenti nel
vettore
        return( [ [self startPoint] count ]);
    // se arrivo qui, mi si chiede quali figli ha un file
    // tratto subito i file normali, che non hanno figli
    if ( [item numOfFiles] == 0 )
        return ( 0 ) ;
    // se arrivo qui, ho una directory
    // se devo mostrare anche i dotfiles, conto tutto
    if ( [[Preferences objectForKey:keyShowDotFiles] boolValue] )
        return ( [item numOfFiles] ) ;
    // se arrivo qui, devo eliminare dal computo i dotFiles
    // ed alora, mi tocca esaminare tutti i file e contare quelli
    // che mi interessano
    return ( countNormalFiles( item) ) ;
}


- (BOOL)
outlineView:            (NSOutlineView *)outlineView
    isItemExpandable:    (id)item
{
    if (item == nil)
        return ( YES ) ;
    // se non ha figli, e' un file, non si espande
    if ( [item numOfFiles] == 0 )
        return ( NO ) ;
    // se arrivo qui, l'item ha figli
    // se devo espandere anche i bundle, espando tutto
    if ( [[Preferences objectForKey:keyExpandBundle] boolValue] )
        return ( YES ) ;
    // se arrivo qui, non devo espandere i bundle
    // espando allora i non bundle
    return (! checkIfBundleDirectory([ item fileName ]) ) ;
}


- (id)
outlineView:        (NSOutlineView *) outlineView
```

```
    child:          (int) index
    ofItem:          (id) item
{
    if (item == nil)
        return( [ [self startPoint] objectAtIndex: index ]);
    // se visualizzo anche i dotFiles, non c'e' problema
    if ( [[Preferences objectForKey:keyShowDotFiles] boolValue] )
        return ([item getFileAtIndex:index]);
    // altrimenti, e' un bel pasticcio, devo saltare i dotfiles
    return ( getNormalFile ( item, index ) );
}
```



Rimane solo da identificare correttamente il file delle preferente. Per farlo, uso `Project Builder`, Tab principale `Target`, tab secondario `Application Setting`, campo `Identifier`, al quale ho attribuito il valore `it.djzero00.mc020`. Così facendo, lanciando l'applicazione, compare come per magia il file dallo stesso nome all'interno della directory delle preferenze utente.

I file sono tutti più o meno nuovi, nel senso che ci sono state ampie modifiche ai vecchi file ereditati dalle precedenti puntate. Ho preferito infatti rimodellare, ricommentare e ripulire i file con l'idea di rinfrescarmi la memoria. Non ci sono grandi e sostanziali modifiche; tuttavia, vi consiglio di ripartire anche voi da questi file, sia di codice che di interfaccia (nib).

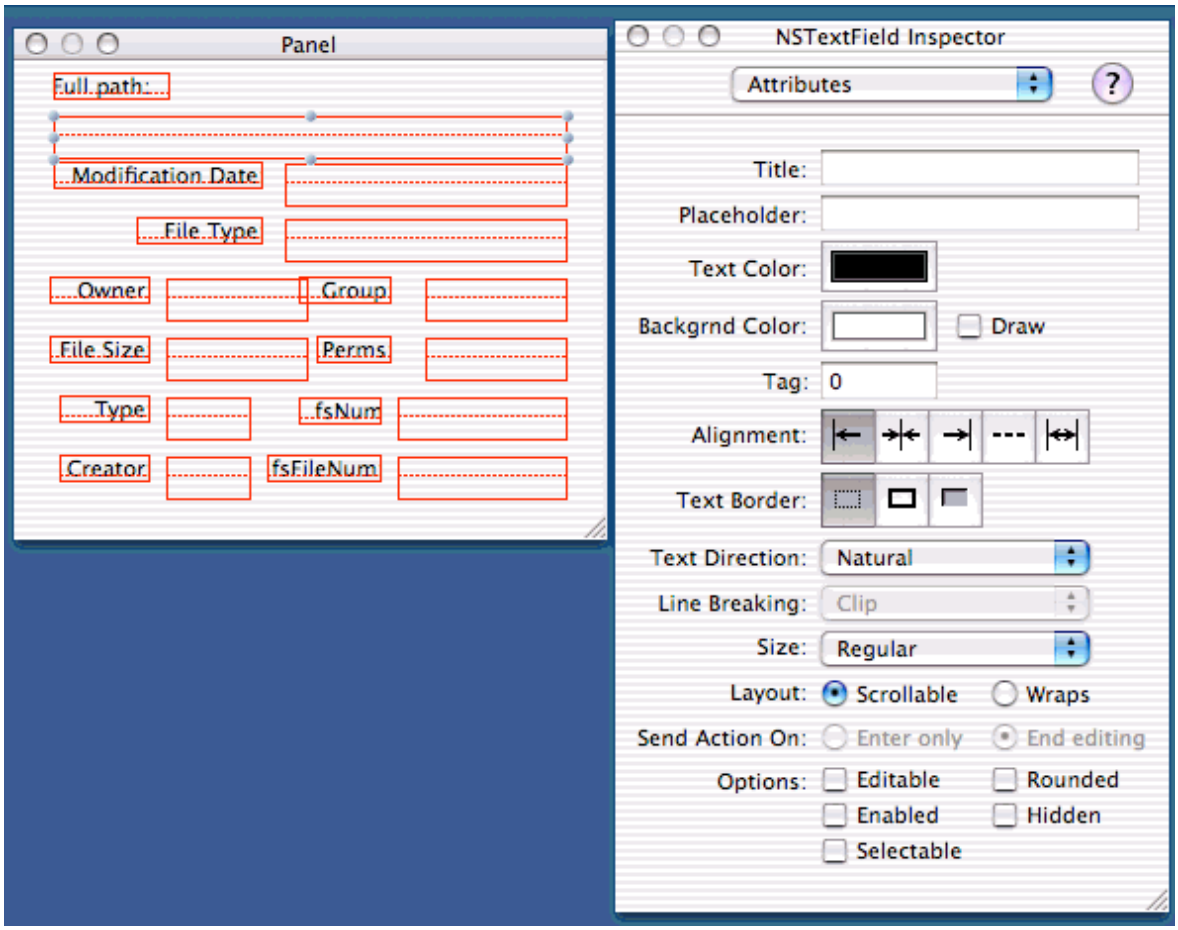
Capitolo 022 - Informazioni e notai

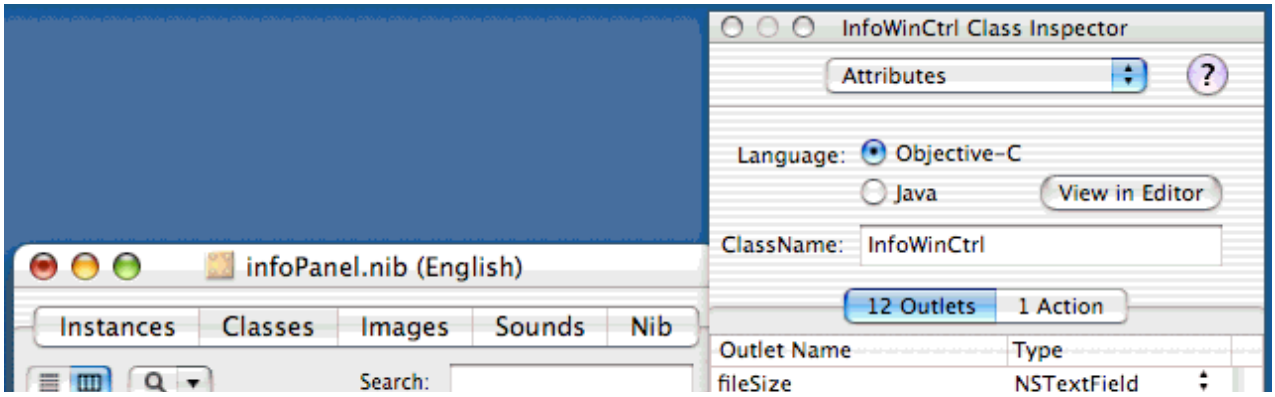
Scopo del capitolo è di realizzare una finestra di 'ispezione', ovvero una finestra all'interno della quale sono presentate tutte le informazioni relative ad un file.

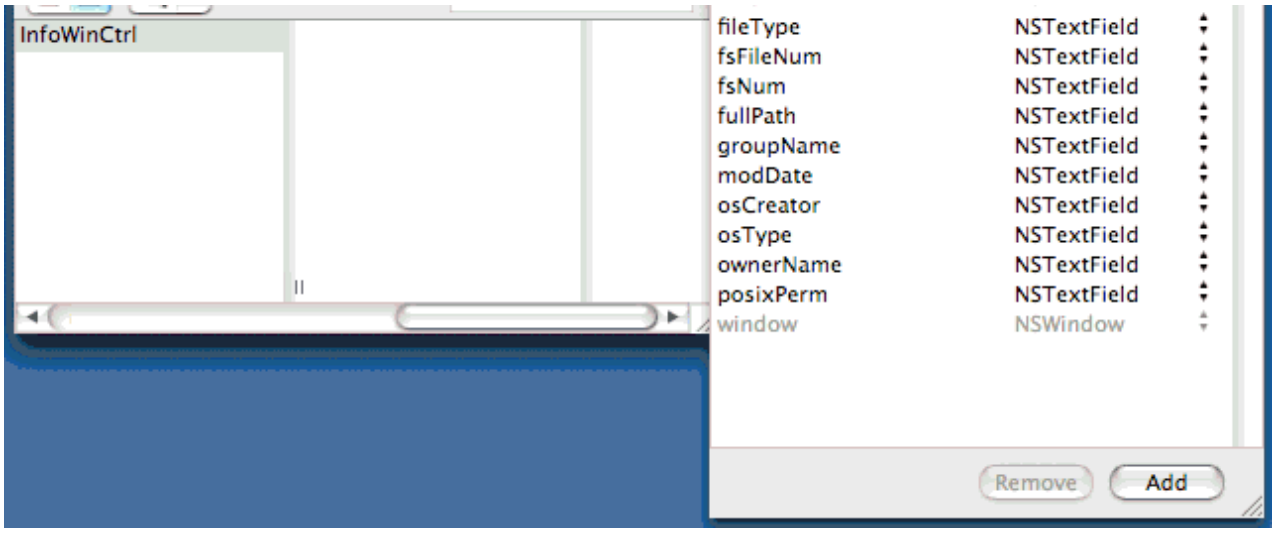
Sorgenti: Copio, semplifico e stravolgo un esempio su *Learning Cocoa*.

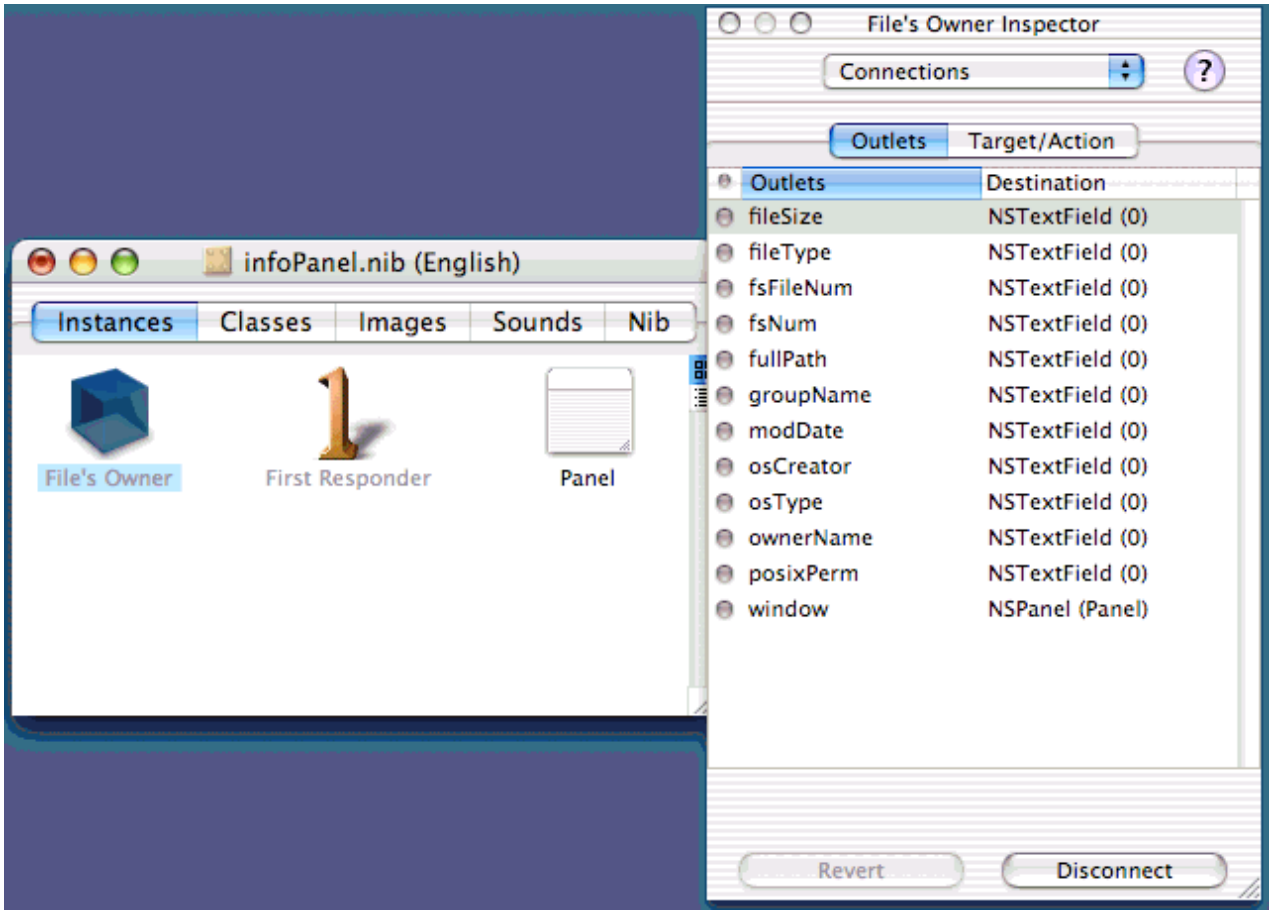
Primo inserimento: 2 settembre 2002

FINESTRA DI INFORMAZIONI





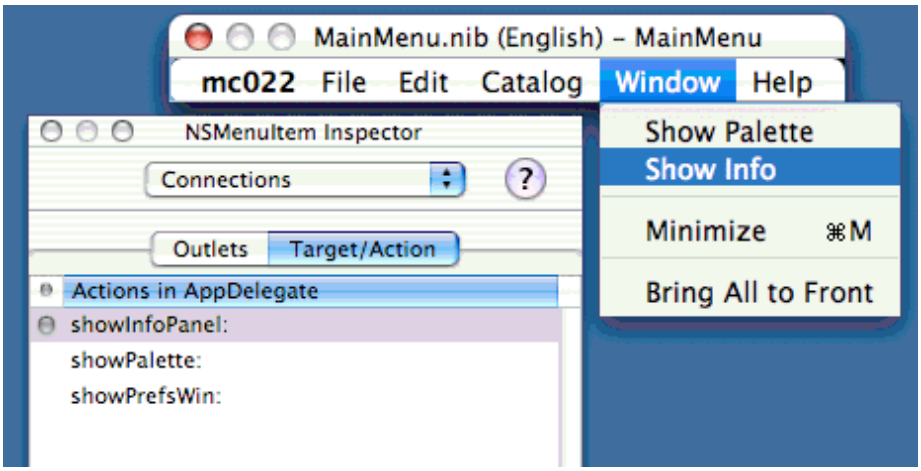




La finestra principale del catalogo è nata densa di informazioni, e si è via via spopolata. In effetti, terrei il nome del file, la data di modifica e la dimensione. La visualizzazione del resto è più o meno opzionale. Decido quindi di realizzare una finestra aggiuntiva in cui, dato un file, sono riportate per esteso le informazioni (il gruppo, il proprietario, il path completo, i permessi, eccetera). La finestra dovrebbe funzionare come la corrispondente finestra del Finder: selezionando un elemento all'interno del catalogo, automaticamente sulla finestra di info compaiono le informazioni pertinenti.

Ho collegato tramite una lunga sequenza di outlet il `File's Owner` ai vari campi della finestra, in modo da poterli riferire facilmente. Salvo tutto.

DELEGA ALL'APPLICAZIONE



Adesso devo collegare una qualche voce di menu con la finestra delle informazioni. Rimango in IB ed apro `MainMenu.lib`, dove appunto si trova il menù dell'applicazione. Aggiungo una voce di menu proprio sotto la simile voce che attiva la palette dei comandi; la chiamo `Show Info`. Ora si tratta di attribuire a questa voce un qualche metodo che apra la finestra.

Con questa, siamo alla terza finestra che deve essere unica all'interno dell'applicazione: c'è la palette dei comandi, poi la finestra delle preferenze, ed adesso questa finestra delle informazioni. Seguendo *Learning Cocoa*, utilizzo un altro (e tre) meccanismo per fare ciò; consiste nel definire una nuova classe, `AppDelegate`, tramite IB e direttamente all'interno del file `MainMenu.nib`. Alla classe aggiungo una action, `showInfoPanel`, che invita la finestra a farsi vedere. Faccio una istanza di questa classe, e collego la voce di menu seguendo il paradigma target/action. A questo punto torno in XCode e scrivo il metodo:

```
- (IBAction)showInfoPanel:(id)sender
{
    // mostro la finestra invocando l'istanza condivisa
    [ [ InfoWinCtrl sharedInfoWinCtrl ] showWindow: sender ] ;
}
```

Tengo a mente il metodo di classe `sharedInfoWinCtrl` e vado a scrivere tutti i vari metodi del controllore della finestra delle informazioni.

IL CONTROLLORE DELLA FINESTRA DELLE INFO

Fondamentalmente, deve scrivere questi tre metodi:

```
+ ( id )    sharedInfoWinCtrl ;
- ( id )    init ;
- (void)    windowDidLoad ;
```

Con il primo metodo (fintamente) di classe, mi procuro un metodo per accedere tranquillamente all'unica istanza, condivisa a tutta l'applicazione, della finestra delle info. Il secondo ed il terzo metodo partecipano alla costruzione della finestra.

```
+ ( id )
sharedInfoWinCtrl
{
    static    InfoWinCtrl    * locSharedWinCtrl = nil ;

    if ( ! locSharedWinCtrl )
    {
         locSharedWinCtrl = [[ InfoWinCtrl allocWithZone: [self zone]] init
] ;
    }
    return ( locSharedWinCtrl );
}
```

Qui definisco una variabile statica, inizialmente nulla; la prima volta che questo metodo viene invocato, è costruito l'oggetto; tutte le altre volte, l'oggetto già definito è subito pronto.

La prima volta poi che questo metodo è chiamato è necessariamente eseguito anche questo metodo di `init`, che giustamente si preoccupa di caricare la finestra dal nib corretto:

```
- (id )
init
{
    self = [ self initWithWindowNibName: @"infoPanel" ];
    if ( self )
    {
        [ self setWindowFrameAutosaveName: @"Info" ];
    }
    return ( self );
}
```

Infine, sempre la prima volta, ma quando la finestra ha finito di caricarsi, è eseguito il seguente metodo:

```
- (void)
windowDidLoad
{
    // costruisco i formattatori che mi servono
    // nota come riuso bellamente quelli per la outlineView
    TOS9TCForm     *myDF1 = [[[ TOS9TCForm alloc ] init ] autorelease ];
     FileSizeForm      *myDF2 = [[[ FileSizeForm alloc ] init ] autorelease
];
    FilePosixPerm     *myDF3 = [[[ FilePosixPerm alloc ] init ] autorelease
];
    [ super windowDidLoad ] ;
    // installo i formattatori
    [ osCreator setFormatter: myDF1 ];
    [ osType setFormatter: myDF1 ];
    [ fileSize setFormatter: myDF2 ];
    [ posixPerm setFormatter: myDF3 ];
    // altro codice che per ora non interessa
}
```

Qui ne approfitto per associare ad alcuni campi della finestra dei formatter (proprio quelli che a suo tempo avevo costruito per la `NSOutlineView`, e che adesso non servono più), ricordando per inciso che per la data ho inserito in IB il formatter standard per le date.

UNIFICAZIONE DELLE GESTIONI

Questo meccanismo mi piace talmente tanto che lo utilizzo anche per le finestre delle preferenze e dei comandi.

Se per la palette di comandi le cose si risolvono facilmente (rinomino anche qualche file), molto più complicata è la trasformazione della finestra delle preferenze: ne approfitto anche per fare un po' di riordino.

Dal nib principale elimino classe ed istanza di `Preferences`, che non servono più; ho infatti aggiunto altre due action alla classe `AppDelegate` in modo che permettano la visualizzazione delle finestre.

Trasformo la classe `Preferences` da semplice sottoclasse di `NSObject` a sottoclasse di `NSWindowController`; adesso la classe si chiama `PrefWinCtrl`. Ripulisco un po' i file e li adatto alla nuova situazione. Fondamentalmente si tratta di scrivere i due metodi `init` e `windowDidLoad` smembrando (ed eliminando) il vecchio metodo `showPanel`.

```
- (id )
init
{
    self = [ self initWithWindowNibName: @"prefsPane" ];
    if ( self )
    {
        [ self setWindowFrameAutosaveName: @"Preferences" ];
        // carico le preferenze dal file
                defCurrValues  =  [[[self  class]  preferencesFromDefaults]
copyWithZone:[self zone]];
        // sono sia def-curr che def-file
        defFileValues = [defCurrValues retain];
        // assegno gli stessi valori anche a def-disp
        [self discardDisplayedValues];
        // inizializzo queste due variabili che mi servono poi
        lsPrefsYes = [[NSNumber alloc] initWithBool:YES];
        lsPrefsNo = [[NSNumber alloc] initWithBool:NO];
    }
    return ( self );
}

- (void)
windowDidLoad
{
    [ super windowDidLoad ] ;
    // non voglio che compaia nella lista delle finestre
    [[self window] setExcludedFromWindowsMenu:YES];
    // questo lo ignoro
    [[self window] setMenu:nil];
    // inserisco i def-curr nella finestra
    [self updatePrefWindow];
    // la piazzo al centro dello schermo
    [[self window] center];
    // porto la finestra davanti a tutti
    [[self window] makeKeyAndOrderFront:nil];
}
```

Nella baraonda scompaiono alcuni metodi perché accorpati in altri, ed altri sono semplificati. Vi rimando al progetto completo per vedere tutte le modifiche.

Non sono tuttavia soddisfatto; ho mescolato allegramente una classe Controller con una classe Model, e sarebbe meglio tenerle separate... Prima o poi farò qualche altra modifica, me lo sento.

Accadrà nel prossimo capitolo.

NOTIFICHE

Arrivo adesso alla parte più interessante di tutto il capitolo, ovvero come riconoscere dall'interno della finestra delle Info che qualcosa è cambiato in una delle `NSOutlineView` che costituiscono il cuore del catalogo. Utilizzo l'interessante meccanismo delle notifiche.

Per capire come funziona, pensiamo alla redazione di un giornale. Il giornale è costituito da una serie di notizie, che vari giornalisti (eventualmente sparsi per il mondo) forniscono ad un centro di smistamento (la redazione). Esistono quindi una serie di agenti che producono notizie. D'altra parte, chiunque può abbonarsi al giornale; nel mondo moderno, è possibile pensare che qualcuno si abboni alle sole notizie di sport, addirittura alle sole notizie di cricket. Esistono quindi una serie di agenti che utilizzano queste notizie, e sono *informati* in tempo reale dell'accadimento delle notizie cui sono abbonati.

Tornando a noi, diciamo che all'interno di ogni applicazione il sistema operativo fornisce una redazione. A questa redazione i vari oggetti presenti all'interno della applicazione si rivolgono annunciando il verificarsi di determinati eventi (che so, un oggetto dedito a calcoli furibondi può annunciare il progredire delle operazioni, rendendo disponibile lo stato di avanzamento dei calcoli). Questa pubblicazione di eventi avviene automaticamente (come accade per molti oggetti dell'interfaccia) oppure esplicitamente, tramite messaggi appositi. D'altra parte, ogni oggetto può abbonarsi a certe categorie di eventi, ovvero ricevere messaggi in corrispondenza del verificarsi di determinati eventi. La redazione alacremente riceve eventi da molti oggetti, verifica se ci sono oggetti abbonati a questi eventi, se il caso li reinstrada, altrimenti, se nessuno è interessato, li butta via.

La cosa interessante del meccanismo è che chi produce gli eventi non deve sapere chi riceve, anzi, produce eventi in tutta tranquillità senza porsi il problema di capire se a qualcuno possano interessare. Viceversa, lo stesso evento può interessare a diversi agenti, e questi possono utilizzarlo in maniera concorrente e con comportamenti differenti.

GESTIRE LA FINESTRA INFO

Tornando al mio problema, scopro che gli oggetti della classe `NSOutlineView` pubblicano una notevole quantità di eventi; si trovano a notificare accadimenti quali lo spostamento ed il ridimensionamento di una colonna, l'espansione o la contrazione di un elemento, e, cosa che mi interessa, quando cambia la selezione corrente. Queste notifiche avvengono automaticamente, e normalmente cadono nel vuoto dal momento che nessuno si è abbonato. Detto questo, risulta molto semplice modificare l'aspetto della finestra di informazioni in corrispondenza di un cambio di selezione all'interno di una `NSOutlineView`.

In primo luogo, occorre sottoscrivere l'abbonamento. Ciò si ottiene con la seguente istruzione all'interno del metodo `winwowDidLoad`:

```
[[ NSNotificationCenter defaultCenter] addObserver: self
    selector: @selector( selectionChanged: )
    name: NSOutlineViewSelectionDidChangeNotification object: nil ] ;
```

Dapprima si rintraccia la redazione, ricavando un oggetto di classe `NSNotificationCenter`; a questo oggetto gli si invia un messaggio dal nome lungo come:

```
addObserver: selector:name:object:
```

in cui si dice di aggiungere un abbonamento a nome di qualcuno (l'argomento di `observer`), a tutti gli eventi di un certo tipo (l'argomento di `name`) coinvolgenti un determinato oggetto (l'argomento di `object`). Quando ciò accade, la redazione deve inviare un certo messaggio (l'argomento di `selector`), ovviamente a chi ha sottoscritto l'abbonamento. Nel mio caso, dico di inviare il messaggio `selectionChanged:` (che è un metodo che presto scriverò) quando cambia la selezione corrente all'interno di una `NSOutlineView`; avendo specificato `nil` come oggetto mittente di interesse, verranno inviate tutte le notifiche di quel tipo, indipendentemente da chi ha generato la notifica stessa (il che va bene: qualsiasi `NSOutlineView` all'interno dell'applicazione sarà un catalogo).

Non rimane altro che scrivere il metodo che svolge tutto il lavoro; a parte le prime righe, è tutta bassa manovalanza:

```
- (void )
selectionChanged: ( NSNotification *) notification
{
    int    row ;
    FileStruct    * locItem ;
    // guardo qual e' la outlineView interessata
    NSOutlineView    * outView = [ notification object ] ;
    // vedo se ci sono selezioni in corso
    row = [ outView selectedRow] ;
    // se non ce ne sono, faccio nulla
    if ( row == -1 )
        return ;
    // recupero l'elemento selezionato
    locItem = [ outView itemAtRow: row ] ;
        // adesso, piu' o meno ordinatamente, estraggo i vari campi
dell'elemento
     // e li attribuisco agli elementi dell'interfaccia della finestra di
info
    // stringa per stringa
    [ fileType setStringValue: [ locItem fileType]];
    [ groupName setStringValue: [ locItem ownGroupName]];
    [ ownerName setStringValue: [ locItem ownerName]];
    [ fullPath setStringValue: [ locItem fileFullPath]];
    // piglio l'intero e lo traformo in stringa
      [ fsFileNum setStringValue: [ NSString stringWithFormat: @"%d", [
locItem fsFileNum]] ];
     [ fsNum setStringValue: [ NSString stringWithFormat: @"%d", [ locItem
fsNum]]] ;
      // questi hanno un formattatore appiccicato, gli passo direttamente
```

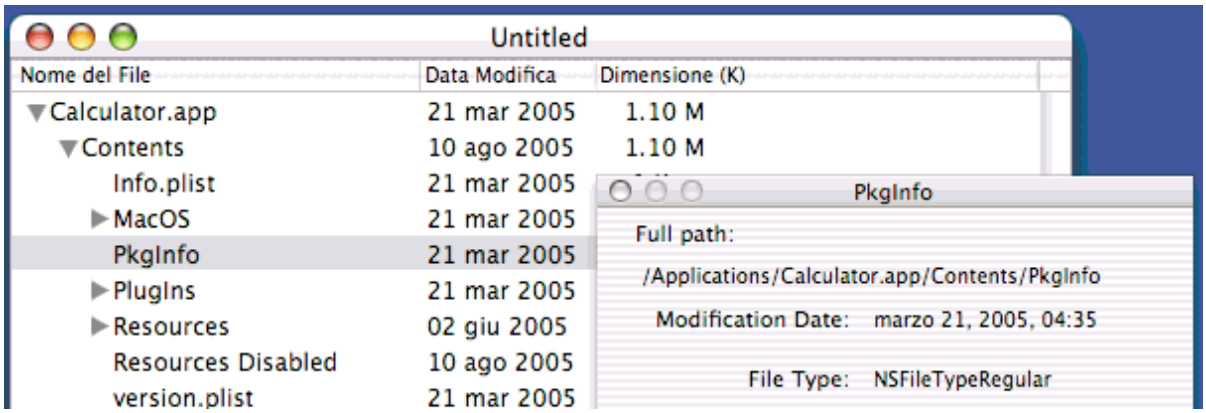
```
l'intero
    [ fileSize setIntValue:[ locItem fileSize] ];
    [ osCreator setIntValue: [ locItem creatorCode]] ;
    [ osType setIntValue: [ locItem typeCode] ];
    [ posixPerm setIntValue: [ locItem filePosixPerm] ];
     // qui, con il formattatore di data, gli passo direttamente l'oggetto
NSDate
    [ modDate setObjectValue: [ locItem modDate]];
    // infine, il nome del file e' il titolo della finestra
        [[ self window ] setTitleWithRepresentedFilename: [ locItem
fileFullPath ]] ;
}
```

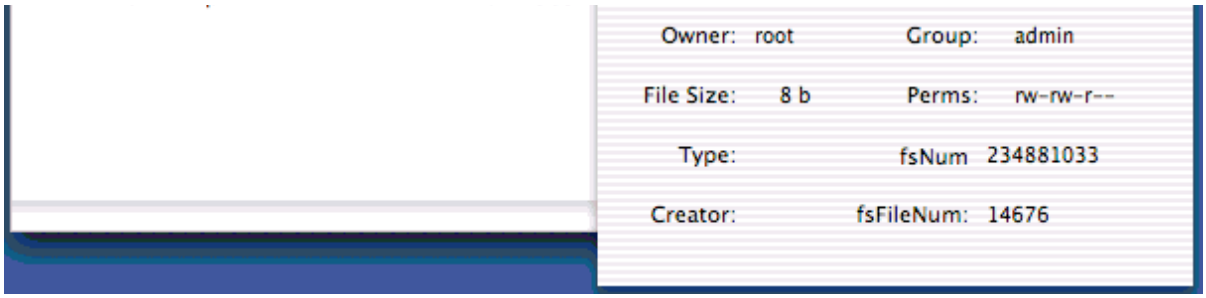
L'argomento del metodo è un oggetto di tipo `NSNotification`, che contiene alcune informazioni sulla notifica che ha scatenato tutta la faccenda; ciò che mi interessa è sapere qual è la `NSOutlineView` che ha generato l'evento; ciò si ottiene velocemente estraendo la variabile `object`. Dopo, è un gioco da ragazzi: dalla `NSOutlineView` ricavo la linea selezionata (se ce ne è una), dalla riga ricavo direttamente l'oggetto di classe `FileStruct`, e da qui i vari elementi per riempire la finestra delle Info. C'è solo da fare un po' di attenzione quando ho dei campi con un formatter attaccato; e poi il colpo di mano finale, con cui si cambia il titolo della finestra di informazioni assegnandogli il nome del file. Il metodo utilizzato, al posto del più ovvio `setTitle`, aggiunge un tocco di classe aggiuntivo: nel titolo della finestra è aggiunta anche l'icona del file.

Due minuti dopo: troppo bello per essere vero. Se il file non è in linea, non solo l'icona non viene riportata, ma anche il nome del file non è cambiato. Occorre procedere ad una modifica. Tuttavia, anche se cambio l'ultima istruzione con

```
if ( [ [NSFileManager defaultManager] fileExistsAtPath: [ locItem
fileFullPath ] ] )
        [[ self window ] setTitleWithRepresentedFilename: [ locItem
fileFullPath ]] ;
else    [[ self window ] setTitle: [ locItem fileName ]] ;
```

dove mi chiedo se il file sotto esame esiste, e solo se esiste uso il metodo che mostra l'icona, le cose non vanno bene. Una volta che il file non esiste più, l'ultima icona utilizzata rimane lì, dal momento che `setTitle` non pulisce completamente il titolo della finestra (provate ad aggiungere un CD o un hard disk esterno ad un catalogo, giocate un po' con la finestra delle Info con il volume in linea, poi espellete il CD e continuate a giocare con la finestra delle Info; capirete meglio ciò che sto faticosamente cercando di spiegare).





Alla fine, mi accontento di utilizzare `setTitle`.

Ci sono molte cose che mi disturbano ancora. Ad esempio, cambiando finestra di catalogo, la finestra delle Info non si aggiorna (dopotutto, la selezione non è cambiata). Deselezionando un elemento, la finestra delle info non cambia, e lascia esposte le informazioni relative all'ultimo file selezionato. Se apro la finestra delle info con un elemento già selezionato, la finestra delle info non presente le informazioni relative (dopo tutto, l'evento si è già verificato...). Ma di questo mi preoccuperò in altri tempi.

Accadrà nel capitolo 26.

Capitolo 023 - Preferenze rivisitate

In questo capitolo raggruppo due argomenti con parecchi effetti collaterali: la ristrutturazione del meccanismo delle Preferenze (ve l'avevo detto...) con la sua estensione, e la manipolazione della `NSOutlineView` del catalogo con la variazione del numero di colonne.

Sorgenti: Documentazione Apple e un po' di sudore.

Primo inserimento: 2 settembre 2002

Ancora Preferenze

Avevo detto che non ero soddisfatto del meccanismo di gestione delle preferenze; in particolare non mi piaceva il fatto che unica classe svolgesse funzioni di Controller della finestra e come Model del dizionario delle preferenze. Quindi, il primo argomento è la divisione della classe `PrefWinCtrl` in due. Ne approfitto per migliorare due aspetti: ripulire ancor di più il meccanismo ed incrementare le possibilità di scelta dell'utente.

La mia versione definitiva (e direi anche riusabile) del meccanismo di preferenze si appoggia ad una classe che nomino `UserPrefs` così strutturata:

```
@interface UserPrefs : NSObject {
    // valori correnti delle preferenze
    NSMutableDictionary         *defCurrValues;
}
+ ( id ) getPrefValue: (id) key ;
+ ( NSMutableDictionary*) getPrefs ;
+ (void) loadPrefsFromDefault ;
+ (void) savePrefsToDefault: ( NSMutableDictionary*) newPref ;
```

La classe ha come unica variabile d'istanza un dizionario con l'elenco dei campi di Preferenze. Sarà inoltre cura della classe fare in modo che esista una sola istanza della stessa, comune a tutta l'applicazione; questo è il motivo per cui l'accesso avviene solo attraverso i quattro metodi di classe indicati.

Il primo metodo è quello utilizzato da qualunque oggetto che vuole conoscere il valore di una delle Preferenze. Deve solo inviare il seguente messaggio:

```
[ UserPrefs getPrefValue: <chiave>]
```

Sono stato nel dubbio se aggiungere metodi di comodo per recuperare direttamente i valori delle Preferenze, ma poi ho lasciato perdere...

Gli altri tre metodi sono adatti agli oggetti che sono coinvolti nella gestione dell'intero insieme delle Preferenze: nella fattispecie, il controllore della finestra delle Preferenze `PrefWinCtrl`.

La realizzazione dei metodi deriva dalla vecchia gestione: ho definito due variabili statiche, inizializzate a `nil`. La prima per tenere traccia dell'unica istanza della classe, e la seconda per mantenere il dizionario delle Preferenze, così come deciso dal programmatore

```
static UserPrefs        * locSharedPref = nil ;
static NSMutableDictionary     * defCodeValues = nil ;
```

Poi ci sono due metodi di facile realizzazione

```
+ ( id ) getPrefValue: (id) key ;
{
    return [[locSharedPref defCurrValues] objectForKey:key];
}
+ ( NSMutableDictionary*) getPrefs
    { return [ locSharedPref defCurrValues] ; }
```

Molto più interessante il metodo `init`, che svolge parecchio lavoro interessante:

```
- ( id )
init
{
    // mi preoccupo che non siano create piu' istanze delle preferenze
    if ( locSharedPref )
        return ( locSharedPref );
    // se arrivo qui devo costruire l'istanza condivisa
    self = [ super init ] ;
    // costruisco il dizionario delle preferenze
    defCodeValues = [ NSMutableDictionary dictionaryWithContentsOfFile:
          [[NSBundle mainBundle] pathForResource: @"defCodeValues" ofType:
@"dict"]];
    [ defCodeValues retain ];
    // l' istanza condivisa e' proprio questa
    locSharedPref = self ;
    // carico le preferenze dal file
    [ UserPrefs loadPrefsFromDefault ];
    return (self );    // restituisco l'istanza
}
```

In primo luogo, l'inizializzazione ha luogo solo una volta, confrontando il valore della variabile `locSharedPref`. Il dizionario delle Preferenze costruito dal codice questa volta preleva i valori da un file (che poi si troverà all'interno del bundle dell'applicazione), per una più facile manutenzione (ricordo che nella realizzazione precedente il dizionario era inizializzato direttamente all'interno del codice con chiavi e valori). Importante il messaggio di `retain` inviato alla variabile: ho impiegato un bel po' di tempo per rendermi conto che il metodo `dictionaryWithContentsOfFile:` costruisce un oggetto `autorelease`, che scompariva dopo un po'....

Di seguito, assegno l'istanza appena creata alla variabile condivisa `locSharedPref`, per poi invocare un metodo che carica le preferenze dal file dei Default.

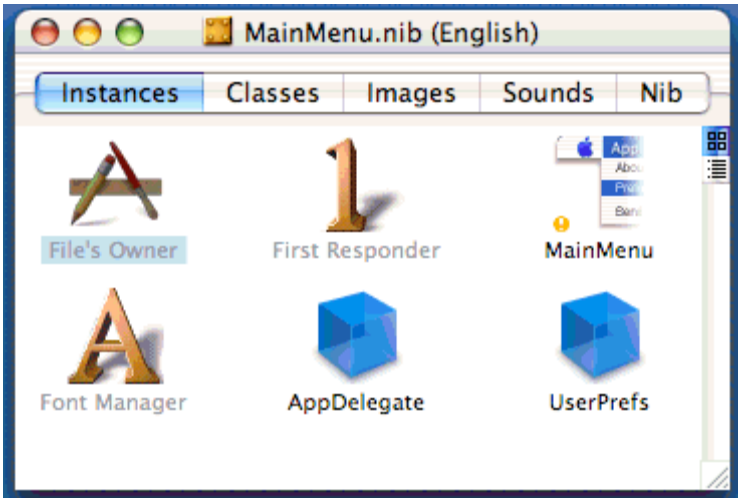
Per quanto riguarda gli altri metodi, non ci sono differenze concettuali di rilievo nei confronti della realizzazione precedente (a parte una piccola ma interessante istruzione che vedrò più tardi):

```
+ (void) loadPrefsFromDefault ;
+ (void) savePrefsToDefault: ( NSMutableDictionary*) newPref ;
```

Da notare che sono comunque metodi di classe per poterli invocare sempre e comunque attraverso messaggi del tipo

```
[ UserPrefs loadPrefsFromDefault ] ;
```

che semplificano notevolmente la gestione.



Rimane da precisare il momento in cui la classe è istanziata. Nella versione precedente questo avveniva nel momento in cui la finestra era aperta per la prima volta; ciò non è una buona cosa (le Preferenze occorrono fin dal primo momento); in questa versione, faccio in modo che il lavoro venga svolto automaticamente. Basta costruire una istanza della classe all'interno di IB: si importa `UserPrefs.h` all'interno di `MainMenu.nib` in modo da avere tale classe nella lista. Da qui, si istanzia la classe, che compare dunque all'interno della finestra. In questo modo, al caricamento del menu principale, è costruita anche l'istanza della classe `UserPrefs`.

NUOVO CONTROLLORE DELLE PREFERENZE

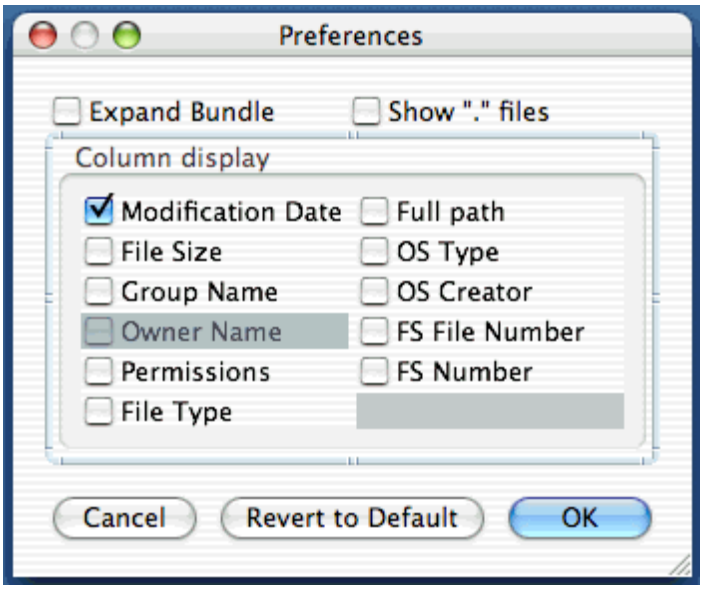
La classe `PrefWinCntrl` subisce una serie di modifiche importanti. In primo luogo, subisce un drastico ridimensionamento nelle variabili e nei metodi:

```
@interface PrefWinCtrl : NSWindowController
{
    IBOutlet NSButton    *expandBundleButton;
    IBOutlet NSButton    *showDotFilesButton;
    IBOutlet NSMatrix     *colDspMatrix;

    // contiene le pref in corso di manipolazione
    NSMutableDictionary     *defDispValues;
}

+ ( id )    sharedPrefsCtrl ;
- ( id )    init ;
- (void)    windowDidLoad ;
```

```
- (void) userSelectUpdate:(id)sender;
- (void) userSelectRestore:(id)sender;
- (void) userSelectCancel:(id)sender;
- (void) updatePrefWindow;

- ( NSMutableDictionary*) defDispValues ;
- (void) setDefDispValues: ( NSMutableDictionary*) newPref ;
```

Rimane una unica variabile d'istanza (a parte gli outlet), per tenere conto dei valori delle Preferenze in corso di modifica. Tuttavia, la parte che ha subito più modifiche è nascosta dall'innocente outlet `colDspMatrix`.

Ho infatti deciso di aggiungere una (lunga) serie di Preferenze per visualizzare o mano, all'interno della finestra del catalogo, le varie colonne con gli attributi del file. Finora infatti la `NSOutlineView` che costruisce l'intera finestra del catalogo prevedeva un numero fisso di colonne (numero variato nel corso dei singoli capitoli di questa storia). Adesso, la `NSOutlineView` avrà una unica colonna sempre presente (il nome del file), mentre le altre colonne appaiono e scompaiono secondo i desideri dell'utente.



Per prima cosa, dunque, ho modificato l'aspetto della finestra delle preferenze. Ho aggiunto una `NSBox` per pura bellezza, all'interno della quale si trova una `NSMatrix`. Questo oggetto è in realtà un contenitore di altri oggetti, che li raggruppa ma soprattutto li tiene ordinati secondo, appunto, una matrice. Nel caso, la `NSMatrix` è costituita da dodici pulsanti di spunta (`NSButtonCell`), divisi su due colonne. Undici di questi pulsanti corrispondono alle undici possibili colonne pertinenti ai vari attributi del file. Il dodicesimo pulsante è stato disabilitato e reso trasparente.

In IB è possibile collegare l'intera `NSMatrix` ad un outlet, oppure i singoli pulsanti. Per mia educazione, ho collegato la `NSMatrix`, per poi accedere, in lettura ed impostazione, ai singoli pulsanti attraverso questo outlet.

Tornando in XCode, il metodo che mantiene il contenuto della finestra congruente con lo stato delle Preferenze è per tanto il seguente:

```
- (void)
```

```
updatePrefWindow
{

#define        MATRIX_SET_STATE( key, row, col)        \
    [colDspMatrix setState: ([ [defDispValues objectForKey: key] boolValue]
? 1 : 0)    \
                      atRow: row column: col ]


    // recupero il valore, ed imposto il bottone di conseguenza
                            [expandBundleButton      setState:[[defDispValues
objectForKey:keyExpandBundle] boolValue] ? 1 : 0];
           [showDotFilesButton   setState:[   [defDispValues   objectForKey:
keyShowDotFiles] boolValue] ? 1 : 0 ] ;

    MATRIX_SET_STATE( keyColModDate,     0, 0 );
    MATRIX_SET_STATE( keyColFileSize,     1, 0 );
    MATRIX_SET_STATE( keyColGroupName,     2, 0 );
    MATRIX_SET_STATE( keyColOwnerName,     3, 0 );
    MATRIX_SET_STATE( keyColPosixPerm,     4, 0 );
    MATRIX_SET_STATE( keyColFileType,     5, 0 );
    MATRIX_SET_STATE( keyColFullPath,     0, 1 );
    MATRIX_SET_STATE( keyColOSType,     1, 1 );
    MATRIX_SET_STATE( keyColCreator,     2, 1 );
    MATRIX_SET_STATE( keyColFsFileNum,     3, 1 );
    MATRIX_SET_STATE( keyColFsNum,         4, 1 );
}
```

Sono stato (altrove) criticato per il mio uso smodato di macro (il costrutto `#define` all'interno del metodo). Secondo me un uso accorto delle macro permette di rendere il codice molto più leggibile ed ordinato. Ad ogni modo, a parte le due istruzioni iniziali che non dovrebbero presentare sorprese, nelle istruzioni successive (le macro) si svolge l'accesso ai singoli pulsanti della `NSMatrix`. Esiste allo scopo un metodo apposito, che permette di impostare lo stato delle celle dandone le *coordinate*, ovvero il numero di riga e di colonna. Occorre giusto notare che gli indici partono da Zero (seguendo la convenzione C) piuttosto che da Uno (la convenzione *naturale*).

Perfettamente simmetrico il metodo di aggiornamento delle Preferenze, attivato quando l'utente fa clic sul pulsante di Ok.

```
- (void)
userSelectUpdate:(id)sender
{

#define        MATRIX_GET_STATE( key, row, col )                        \
      [defDispValues  setObject:
                                  \
         ([[colDspMatrix cellAtRow: row column:col] state ] ? lsPrefsYes :
lsPrefsNo)     \
        forKey:key]


    // devo recuperare i valori dalla finestra
    [defDispValues setObject:
        ([expandBundleButton state] ? lsPrefsYes : lsPrefsNo)
        forKey:keyExpandBundle];
    [defDispValues setObject:
        ([showDotFilesButton state] ? lsPrefsYes : lsPrefsNo)
        forKey:keyShowDotFiles];
```

```
    MATRIX_GET_STATE( keyColModDate,     0, 0 );
    MATRIX_GET_STATE( keyColFileSize,     1, 0 );
    MATRIX_GET_STATE( keyColGroupName,     2, 0 );
    MATRIX_GET_STATE( keyColOwnerName,     3, 0 );
    MATRIX_GET_STATE( keyColPosixPerm,     4, 0 );
    MATRIX_GET_STATE( keyColFileType,     5, 0 );
    MATRIX_GET_STATE( keyColFullPath,     0, 1 );
    MATRIX_GET_STATE( keyColOSType,     1, 1 );
    MATRIX_GET_STATE( keyColCreator,     2, 1 );
    MATRIX_GET_STATE( keyColFsFileNum,     3, 1 );
    MATRIX_GET_STATE( keyColFsNum,     4, 1 );

    // salvo le preference su file
    [ UserPrefs savePrefsToDefault: defDispValues ];
    // e per finire nascondo la finestra
    [[self window] setIsVisible: FALSE ];
}
```

Qui la cosa è leggermente più pasticciata, dal momento che per leggere lo stato di una cella, dapprima occorre recuperare l'oggetto cella (col metodo `cellAtRow:column:`) e poi inviargli il messaggio `state`.

Alla fine, aggiorno le preferenze scrivendole sul file, e nascondo la finestra.

Gli altri metodi della classe `PrefWinCtrl` non sono particolarmente interessanti e non hanno subito modifiche di rilievo.

Ricordo che la visualizzazione della finestra avviene sempre attraverso l'uso della classe delegata `AppDelegate`.

MODIFICARE L'ASPETTO DI NSOUTLINEVIEW

Ora che all'interno delle Preferenze c'è scritto quali colonne devono essere visualizzate all'interno della finestra, occorre scrivere un po' di codice per adeguare la `NSOutlineView` a quanto richiesto. Allo scopo, ho scritto il seguente metodo:

```
- (void)
prefsUpdated: ( NSNotification *) notification
{
    // recupero le preferences e mostro o meno la colonna
     setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColFullPath]
boolValue], COLID_FULLPATH );
      setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColModDate]
boolValue], COLID_MODDATE );
     setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColFileSize]
boolValue], COLID_FILESIZE );
     setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColGroupName]
boolValue], COLID_GROUPNAME );
     setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColOwnerName]
boolValue], COLID_OWNERNAME );
     setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColPosixPerm]
boolValue], COLID_POSIXPERM );
     setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColFileType]
boolValue], COLID_FILETYPE );
      setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColCreator]
boolValue], COLID_OSCREATOR );
       setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColOSType]
```

```
boolValue], COLID_OSTYPE );
     setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColFsFileNum]
boolValue], COLID_FSFILENUM );
       setupColumn( outlineView, [[ UserPrefs getPrefValue: keyColFsNum]
boolValue], COLID_FSNUM );
    [ outlineView reloadData ];
    [ outlineView sizeLastColumnToFit ] ;
}
```

Questo metodo non serve a molto se non spiego prima la funzione seguente:

```
void
setupColumn( NSOutlineView *outView, bool addRem, NSString * colId )
{
    NSTableColumn    *tmpTC ;
    // vedo se esiste una colonna relativa
    tmpTC = [ outView tableColumnWithIdentifier: colId] ;
    // se non c'e', e devo rimuoverla...
    if ( (tmpTC == nil) && (addRem == FALSE) )
        return ;     // devo fare nulla
    // se c'e', e devo aggiungerla
    if ( tmpTC != nil && addRem )
        return ;    // devo fare nulla
    // se non c'e', e devo aggiungerla
    if ( tmpTC == nil && addRem )
    {
        // creo la NSTableColumn con l'identificatore
        tmpTC = [ [NSTableColumn alloc] initWithIdentifier: colId ];
        // imposto il titolo della colonna
         [[tmpTC headerCell] setStringValue: NSLocalizedString( colId, colId
) ];
        // attacco eventualmente un formattatore
        attachFormatter( tmpTC, colId );
        // aggiungo la colonna
        [outView addTableColumn: tmpTC ];
        return ;    // ho finito
    }
    // se arrivo qui, sicuramente c'e' la colonna e devo rimuoverla
    [outView removeTableColumn: tmpTC ];
}
```

Questa funzione intende aggiungere o eliminare una colonna di una `NSOutlineView`. L'operazione da effettuare è data dal valore della variabile booleana `addRem`: se questa è `TRUE`, devo aggiungere la colonna, se `FALSE` devo rimuoverla. La colonna da aggiungere dovrà avere l'identificatore specificato dall'argomento `colID`.

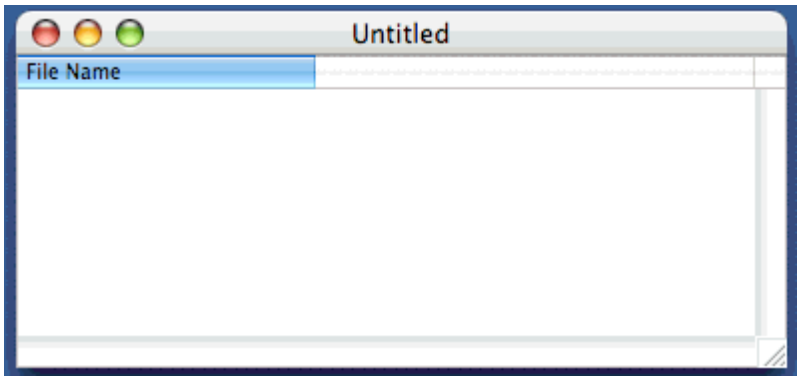
In primo luogo, vedo se tale colonna esiste o meno. Se infatti la colonna non esiste e devo rimuoverla, devo fare nulla. Ugualmente, se la colonna esiste e devo aggiungerla, devo fare nulla.

Le operazioni più interessanti si hanno quando la colonna non c'è e bisogna aggiungerla. Si costruisce allora un oggetto della classe `NSTableColumn` con opportuno identificatore; gli si assegna un titolo (ricavo il titolo utilizzando l'identificatore della colonna e prelevandolo da un file di stringhe localizzate; ho già fatto ciò in uno dei primi capitoli di Macocoa); se la colonna è adatta, gli assegno un formatter (ci arrivo tra breve); alla fine, aggiungo la colonna alla `NSOutlineView`.

L'ultimo caso rimasto, di eliminare una colonna presente, si sbriga con una sola

istruzione col metodo `removeTableColumn:`.

Sono adesso in grado di spiegare il metodo `prefsUpdated:`. Dimentichiamo il suo argomento. Il metodo è costruito essenzialmente da una ripetuta invocazione della funzione utilizzando come argomenti gli identificatori delle varie colonne e lo stato di visualizzazione delle stesse, così come prelevato dalle preferenze.



Il metodo è concluso da due istruzioni; la prima dice alla `NSOutlineView` di ridisegnarsi, dal momento che, molto probabilmente, è cambiato l'aspetto. La seconda è presente per ragioni estetiche: quando riduco il numero di colonne, può verificarsi che queste occupino meno spazio di quanto disponibile nella finestra, lasciando inestetici spazi non occupati da alcuno. Col metodo `sizeLastColumnToFit` si fa appunto in modo che l'ultima colonna si espanda fino a coprire tutto lo spazio disponibile.

Rimane la funzione per attribuire i formatter alle colonne:

```
void
attachFormatter( NSTableColumn *tc, NSString * colId )
{
    if ( [ colId isEqual: COLID_OSTYPE ] )
    {
         TOS9TCForm      *myDF1 = [[[ TOS9TCForm alloc ] init ] autorelease
];
        [[tc dataCell ] setFormatter: myDF1 ];
        return ;
    }
    if ( [ colId isEqual: COLID_OSCREATOR ] )
    {
         TOS9TCForm      *myDF1 = [[[ TOS9TCForm alloc ] init ] autorelease
];
        [[tc dataCell ] setFormatter: myDF1 ];
        return ;
    }
    if ( [ colId isEqual: COLID_FILESIZE ] )
    {
          FileSizeForm        *myDF2  =  [[[  FileSizeForm  alloc  ]  init  ]
autorelease ];
        [[tc dataCell ] setFormatter: myDF2 ];
        return ;
    }
    if ( [ colId isEqual: COLID_POSIXPERM ] )
    {
          FilePosixPerm       *myDF3  =  [[[  FilePosixPerm  alloc  ]  init  ]
autorelease ];
        [[tc dataCell ] setFormatter: myDF3 ];
        return ;
```

```
    }
    if ( [ colId isEqual: COLID_MODDATE ] )
    {
        NSDateFormatter *myDF4 = [[NSDateFormatter alloc]
            initWithDateFormat:@"%d %b %Y" allowNaturalLanguage:YES];
        [[tc dataCell ] setFormatter: myDF4 ];
        return ;
    }
}
```

L'unica cosa degna di nota è la sezione dedicata al formatter della colonna relativa alla data di modifica; fino ad adesso questo formatter è stato applicato direttamente in IB; qui bisogna inserirlo esplicitamente dal momento che la colonna è stata costruita ex-novo.

ANCORA NOTIFICHE

L'ultima cosa da fare è agganciare l'aggiornamento della `NSOutlineView` con l'aggiornamento delle preferenze. I più accorti tra voi avranno il sospetto che tale aggancio sarà realizzato attraverso il meccanismo delle notifiche (come l'argomento del metodo `prefsUpdated:` faceva prevedere).

L'idea è di fare in modo che quando le preferenze sono aggiornate, l'oggetto `UserPrefs` invia un messaggio di notifica al resto dell'applicazione. Ecco quindi come si presenta il metodo `savePrefsToDefault:`, compresa l'ultima istruzione che vi avevo preavvertimento.

```
+ (void)
savePrefsToDefault: ( NSMutableDictionary*) locDict
{
    // recupero il file dei defaults
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    // inserisco ordinatamente i vari elementi
    writeBoolDefault( locDict, defaults, keyExpandBundle );
    writeBoolDefault( locDict, defaults, keyShowDotFiles );

    writeBoolDefault( locDict, defaults, keyExpandBundle );
    writeBoolDefault( locDict, defaults, keyShowDotFiles );
    writeBoolDefault( locDict, defaults, keyColFullPath );
    writeBoolDefault( locDict, defaults, keyColModDate );
    writeBoolDefault( locDict, defaults, keyColFileSize );
    writeBoolDefault( locDict, defaults, keyColGroupName );
    writeBoolDefault( locDict, defaults, keyColOwnerName );
    writeBoolDefault( locDict, defaults, keyColPosixPerm );
    writeBoolDefault( locDict, defaults, keyColFileType );
    writeBoolDefault( locDict, defaults, keyColCreator );
    writeBoolDefault( locDict, defaults, keyColOSType );
    writeBoolDefault( locDict, defaults, keyColFsFileNum );
    writeBoolDefault( locDict, defaults, keyColFsNum );
    // dico a tutti che le pref sono state aggiornate
    [[NSNotificationCenter defaultCenter]
        postNotificationName: PREF_UPDATE_NOTIFICATION object:self];
}
```

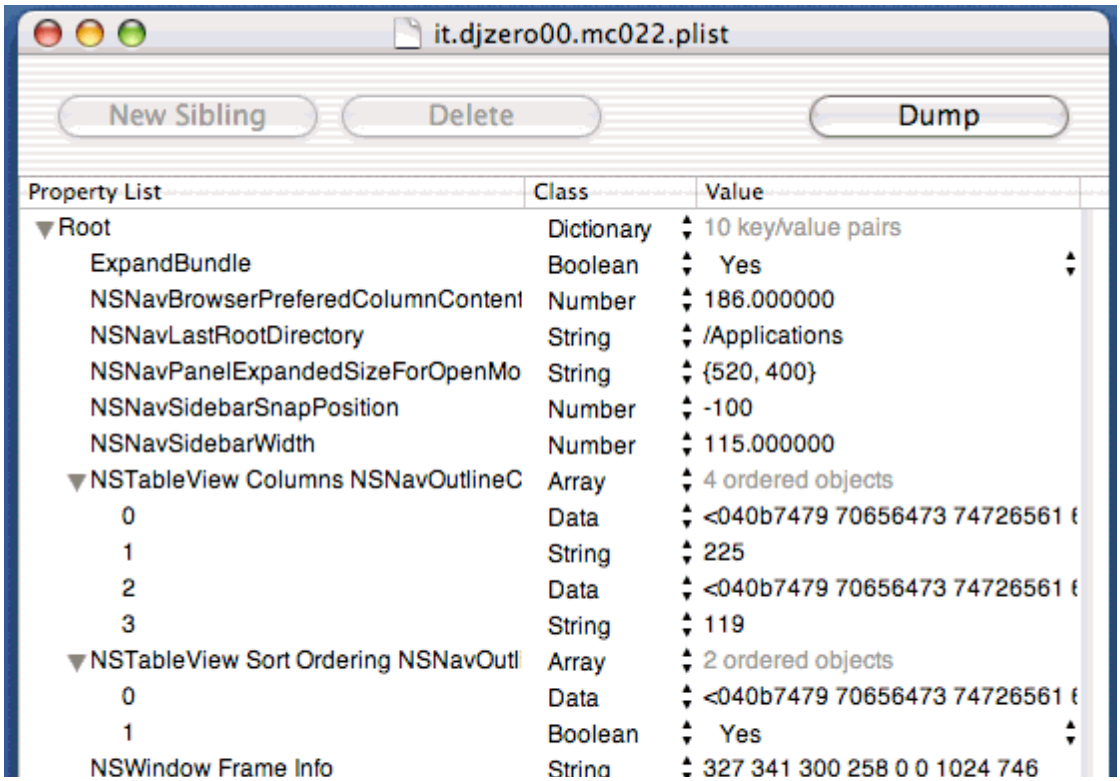
A parte la serie di scritture dei vari campi delle Preferenze, l'ultima istruzione invia appunto una notifica dal nome `PREF_UPDATE_NOTIFICATION` alla redazione; da notare

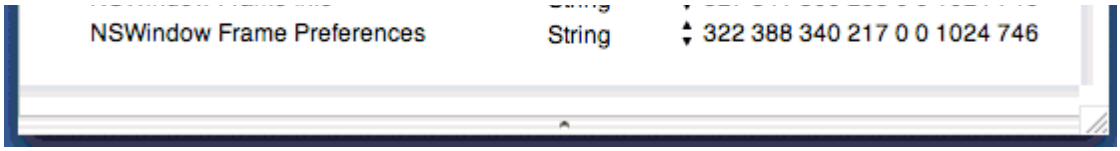
come non si preoccupi per nulla del fatto che ci sia qualcuno ad ascoltare.

Corrispondentemente, all'interno della classe `CatalogDoc` da qualche parte ci deve essere un abbonamento a queste notifiche: eseguo tutto ciò (e qualcosa di più) all'interno del metodo `windowControllerDidLoadNib:`:

```
- (void)
windowControllerDidLoadNib:    (NSWindowController *) aController
{
    ### vecchie istruzioni ###

    [ outlineView setAutosaveName: @"outlineViewOption" ];
    [ outlineView setAutosaveTableColumns: YES ];
    // mi abbono all'evento: pref aggiornate
    [[ NSNotificationCenter defaultCenter] addObserver: self
        selector: @selector( prefsUpdated: )
        name: PREF_UPDATE_NOTIFICATION object: nil ] ;
    // recupero le pref ed adeguo la finestra
    // in realta', faccio finta che siano state aggiornate le prefs
    [[NSNotificationCenter defaultCenter]
        postNotificationName: PREF_UPDATE_NOTIFICATION object:self];
}
```

La penultima istruzione è la sottoscrizione dell'abbonamento: quando un qualsiasi oggetto (argomento `nil`) esegue una notifica `PREF_UPDATE_NOTIFICATION`, invia all'istanza di `CatalogDoc` (l'argomento dell'Observer è `self`) il messaggio `prefsUpdated:`. Per essere poi sicuro che la finestra del catalogo si costruisca congruentemente alle preferenze espresse, si invia una notifica di preferenze aggiornate. Lo so bene che le preferenze non sono state aggiornate, ma ciò costringe la finestra appena costruita ad adeguarsi, attraverso il metodo `prefsUpdated:` (che per altro avrei potuto inviare direttamente: ma in tal caso avrei dovuto costruito un adeguato oggetto `NSNotification` da passare come argomento).





Mi rimangono da spiegare le altre due istruzioni presenti nel pezzo di codice mostrato. Queste non servono altro che a mantenere all'interno delle Preferenze la situazione delle colonne (le dimensioni orizzontali, visto che quali colonne sono presenti è già regolato da altre opzioni). a causa di queste due istruzione, all'interno del file delle Preferenze compaiono nuovi campi che appunto mantengono queste informazioni.

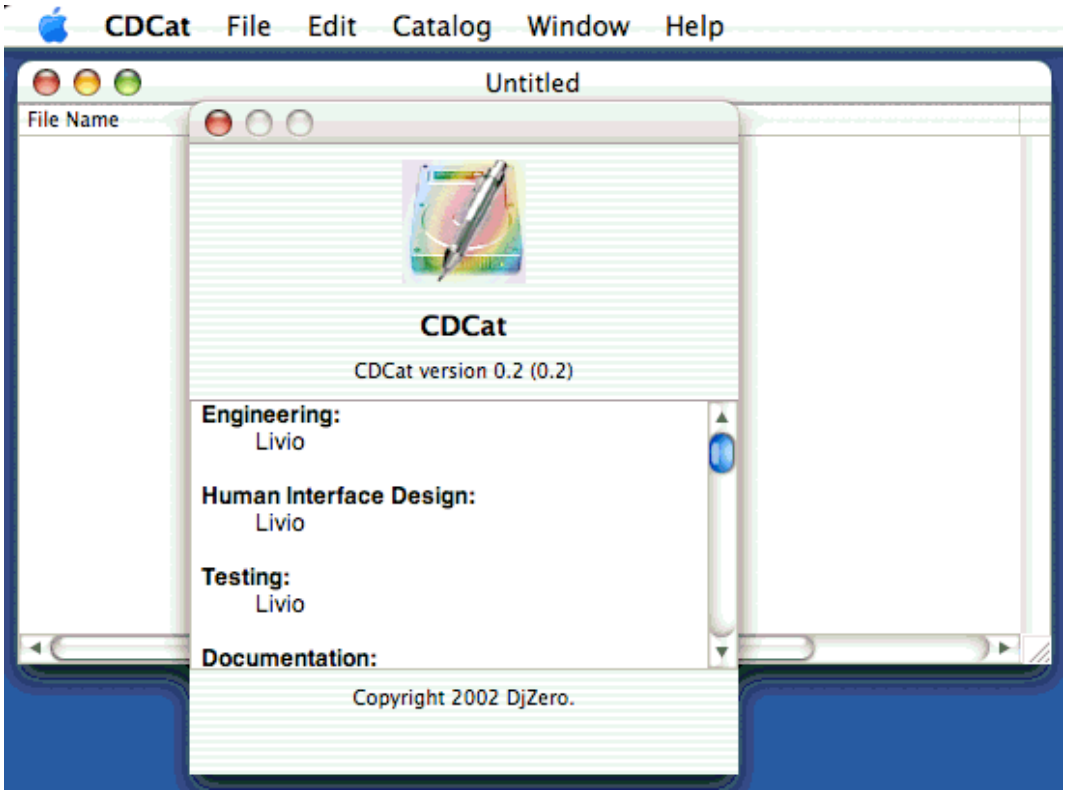
Capitolo 024 - A Proposito

Questo capitolo introduce molti nuovi concetti, forse frettolosamente, ma con lo scopo di arrivare ad una finestra About personalizzata ed animata.

Sorgenti: Faccio mio un tutorial su CocoaDevCentral.

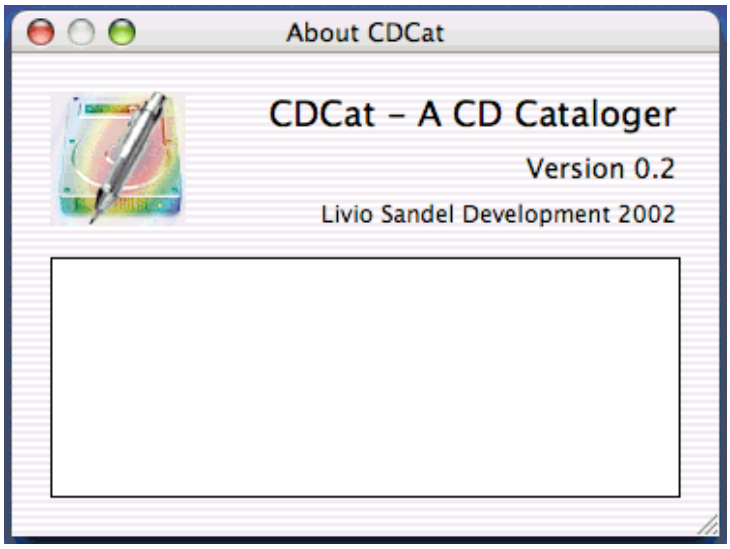
Primo inserimento: 2 settembre 2002

A PROPOSITO

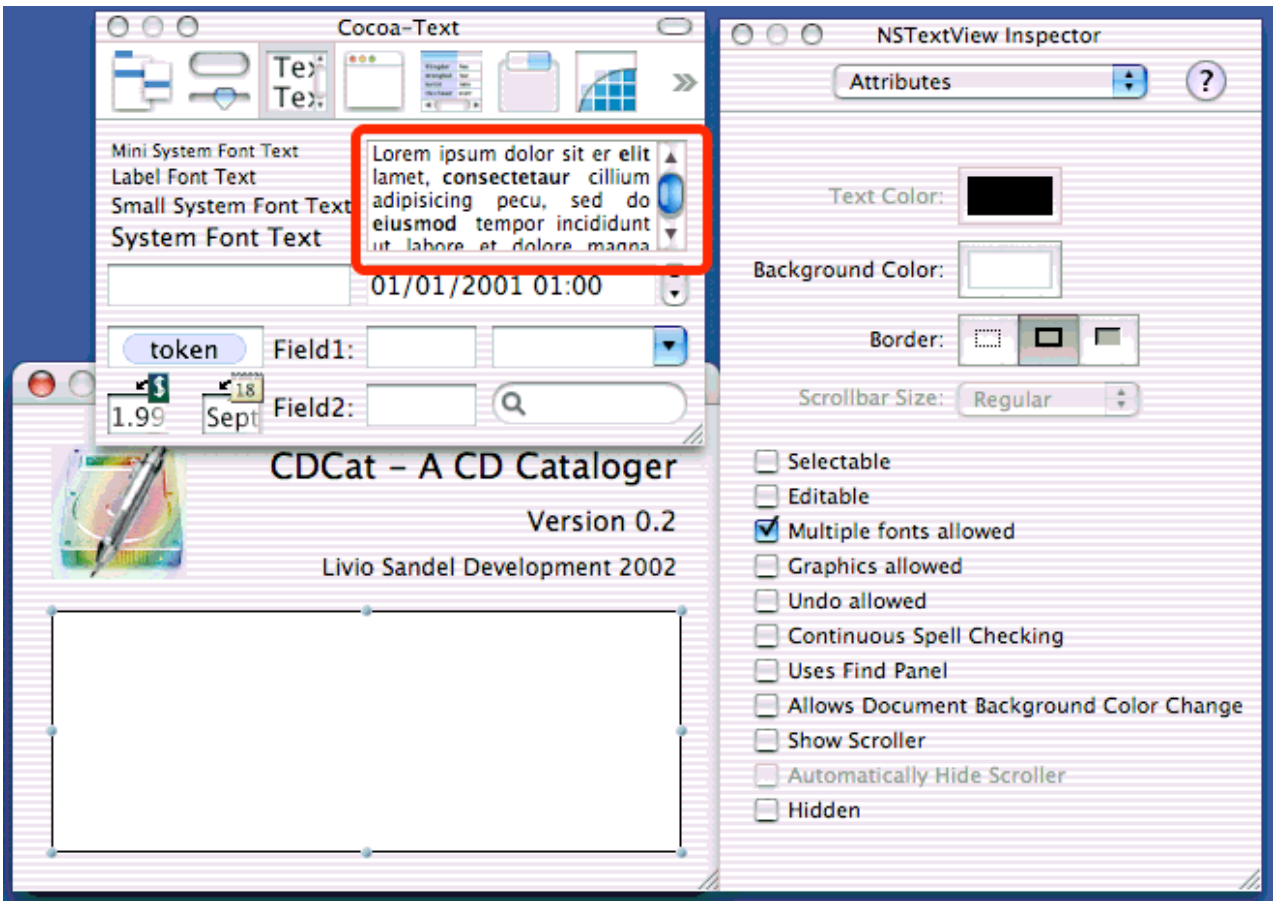


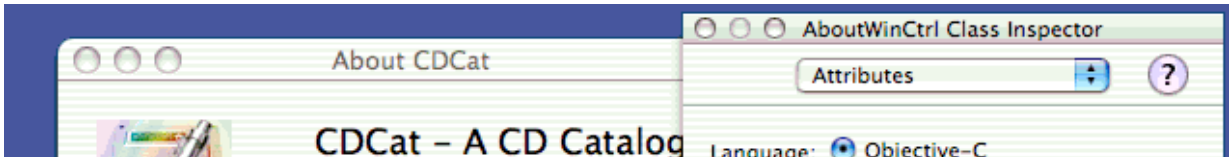
Quando si utilizza Cocoa, XCode e IB per la costruzione di una applicazione, si ottengono molte caratteristiche gratuite (copia, incolla, menu, eccetera). Tra le altre, si ottiene gratis anche la finestra `About`, in cui sono presentate le informazioni relative al programma. In maniera automatica Cocoa fornisce la finestra, che viene riempita dall'icona del programma (a proposito, ho cambiato l'icona del programma con quella porcheria che potete vedere in figura... se qualche anima pia mi fornisce gratuitamente una icona esteticamente più pregevole, è benvenuto) e dalle informazioni che sono recuperate dal file `Info.plist`. Inoltre, se è presente un file dal nome `Credits.rtf`, il suo contenuto è mostrato, completo di stili, all'interno di un campo di testo. Non sempre tale finestra è sufficiente a placare l'esibizionismo del programmatore. In questo capitolo, seguendo un tutorial che ho trovato sul sito Cocoa Dev Central, costruisco una finestra About personalizzata, in cui il testo del file `Credits.rtf` è mostrato scorrevole, a mo' di titoli di coda di un film. Per raggiungere questo scopo, saranno necessari molti nuovi concetti, che sfiorerò solamente per arrivare in tempi brevi all'obiettivo.

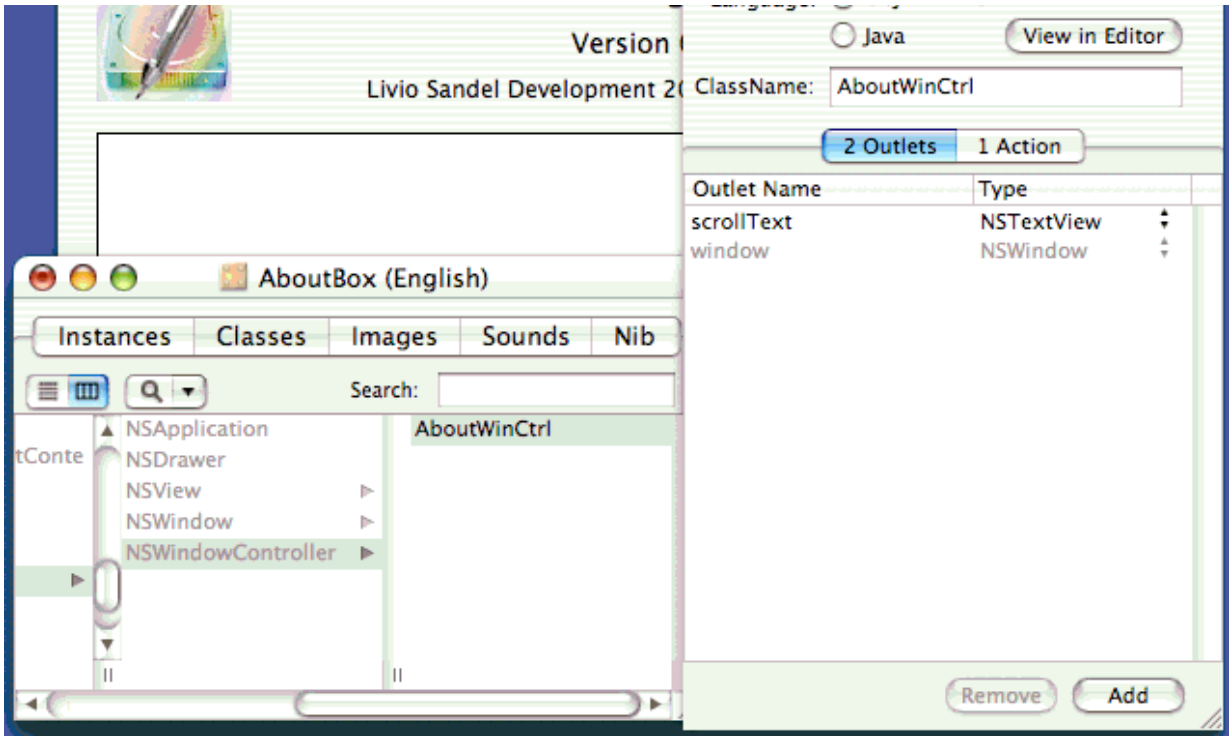
LA FINESTRA



Vado in IB e costruisco la nuova finestra di `About`. Genero un nuovo file nib, che nomino fantasiosamente `AboutBox.nib`. Oltre a riportare l'icona e qualche scritta di benvenuto, introduco un campo di testo. Si tratta di un oggetto della classe `NSTextView`.







Non si tratta del solito campo di testo del quale ho approfittato ampiamente finora, ma di un genere tutto diverso di bestia. È il cavallo di battaglia di ogni applicazione avente a che fare con la manipolazione di testo; per capirci, si può costruire un'applicazione simile a TextEdit con una finestra all'interno della quale c'è un solo oggetto `NSTextView`. Con questo oggetto si possono visualizzare testi in varie forme, fogge e dimensioni, immagini, sono attivi drag'n'ndrop, equivalenti di tastiera per la manipolazione, copia/incolla, controllo ortografico, eccetera. Tuttavia, tutto questo a me servirà a nulla. Tutto ciò di cui ho bisogno è il fatto che `NSTextView` è una sottoclasse di `NSView` e che posso, tramite i metodi ereditati, controllare da programma l'ammontare dello scrolling del testo. L'idea infatti è di infilare il file `Credits.rtf` all'interno del campo di testo scrollabile, nascondere la barra di scorrimento e controllare da programma lo scrolling del testo. Quest'ultima affermazione giustifica gli attributi del campo che ho impostato: accetto font multipli, ma non mostro la barra di scorrimento. Costruisco poi una sottoclasse di `NSWindowController`, che chiamo `AboutWinCtrl`; definisco un unico outlet, il campo di testo; costringo `File's Owner` ad essere una istanza di questa classe, collego l'outlet al campo.

Uovo di Pasqua

Già che sono in IB, aggiungo una action a `MainMenu.nib`, in totale similitudine con quanto fatto per le finestre di Info, la Palette dei Comandi e la finestra delle Preferenze. Quindi, al posto del collegamento standard della voce `About` del menu dell'applicazione, ci sostituisco questa azione. Tengo presente però il metodo invocato, perché mi serve per la realizzazione del metodo seguente:

```
- (IBAction)showAboutBox:(id)sender
{
    // recupero l'evento corrente, controllo lo stato della tastiera
    // e verifico se il tasto Alt è premuto
```

```
    if ([[NSApp currentEvent] modifierFlags] & NSAlternateKeyMask)
    {
          // la risposta e' si', procedo alla visualizzazioen della nuova
finestra
        // uso la solita tecnica
        [ [AboutWinCtrl sharedAboutWinCtrl] showWindow: sender] ;
    }
    else
    {
        // nessun tasto premuto, faccio vedere le solite cose
        [ NSApp    orderFrontStandardAboutPanel: self ];
    }
}
```

Questa volta, invece di eseguire le solite operazioni, realizzo un *Easter Egg*, letteralmente 'uovo di pasqua', ma sarebbe meglio parlare della sorpresa di questo uovo. Molto spesso i programmatori aggiungono caratteristiche nascoste ai loro programmi; spesso accade che la finestra di About, in particolari condizioni, sia diversa dal solito (famosa è rimasta la bandiera con l'iguana in Mac OS 7). Qui faccio lo stesso: se l'utente seleziona la voce del menu in condizioni normali, faccio vedere la finestra normale prodotta da Cocoa. Se invece tiene premuto il tasto Alt (Option per qualcuno), esibisco la nuova finestra.

MOSTRARE LA FINESTRA

A parte i soliti metodi di `init` e `sharedAboutWinCtrl`, le cose diventano interessanti con il metodo `windowDidLoad`, in cui si devono cominciare a predisporre un po' di cose:

```
- (void)
windowDidLoad
{
    NSString *creditsPath;
    NSAttributedString *creditsString;

    [ super windowDidLoad ] ;
```

Fino a qui non c'è nulla di particolare; lasciamo per il momento perdere le tre istruzioni seguenti:

```
    // inizio la posizione corrente verticale
    currentPosition = 0;
    restartAtTop = NO;
    // imposto l'istante di partenza, qualche secondo dopo l'inizio
            startTime    =    [NSDate    timeIntervalSinceReferenceDate]    +
ABOUTWIN_STARTSCROLLDELAY ;
```

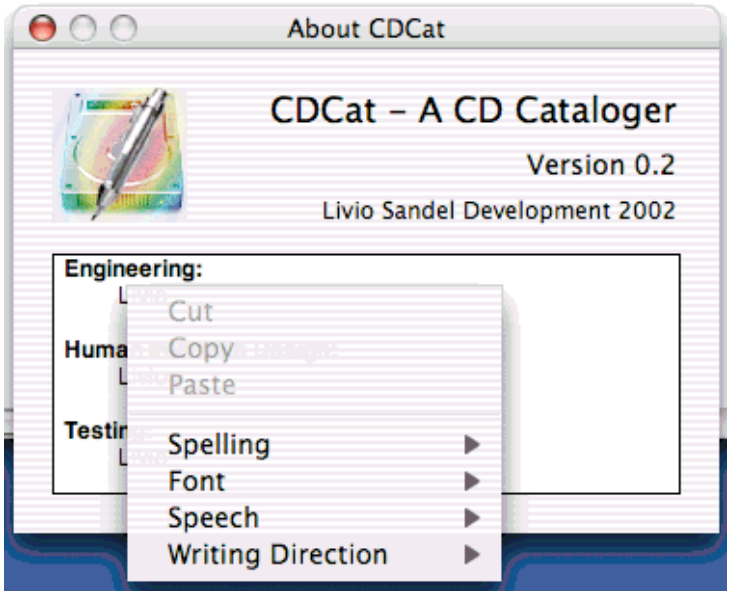
Qui di seguito carico in memoria il contenuto del file `Credits.rtf`; prima costruisco il path completo tenendo conto delle possibili localizzazioni (e quindi ricorro ai bundle), poi metto il contenuto del file all'interno di una `NSAttributedString`, piuttosto che in una normale `NSString`. Infatti una `NSAttributedString` è una stringa che possiede degli *attributi*, dove per attributo si intende un insieme di caratteristiche (font, dimensione, kerning, eccetera) che si applicano ad uno o più caratteri. In pratica, è una stringa di caratteri che mantiene la piena definizione degli stili presenti. Con l'oggetto

`NSAttributedString` ci riempio poi il campo di testo, mantenendo la formattazione RTF.

```
    // costuisco il path al file Credits.rtf localizzato
        creditsPath  =  [[NSBundle  mainBundle]  pathForResource:@"Credits"
ofType:@"rtf"];
    // carico il contenuto del file
      creditsString  =  [[NSAttributedString  alloc]  initWithPath:creditsPath
documentAttributes:nil];
    // riempio il campo di testo con il contenuto del file
    [scrollText replaceCharactersInRange:NSMakeRange( 0, 0 ) withRTF:
          [creditsString     RTFFromRange:  NSMakeRange(  0,  [creditsString
length] )
                documentAttributes:nil] ];
```

Dimentichiamo ancora una volta le due istruzioni seguenti...

```
    // calcolo l'altezza del capo di testo
    maxScrollHeight = [ creditsString size ].height ;
    // si comincia con l'altezza a Zero
    [scrollText scrollPoint:NSMakePoint( 0, 0 )];
```



Qui c'è un bel tocco di classe (che infame gioco di parole): normalmente (il comportamento di default previsto da Cocoa) un campo di testo si porta dietro un menu contestuale che permette di eseguire le operazioni di copia/incolla, di controllo ortografico, eccetera. Poiché questo non è il comportamento voluto in questo momento, dico di non associare menu alla vista. Poi il metodo chiude con le solite istruzioni relative alla finestra nel suo complesso.

```
    // evito di associare menu al campo (font, spelling, ecc)
    [scrollText setMenu:nil ];

    // non voglio che compaia nella lista delle finestre
    [[self window] setExcludedFromWindowsMenu:YES];
    // evito di associare menu alla finestra
    [[self window] setMenu:nil];
    // porto la finestra davanti a tutti
    [[self window] makeKeyAndOrderFront:nil];
```

```
}
```

È il momento adesso di parlare dello scrolling del campo e di come questo sarà eseguito. Dicevo che l'idea consiste nel mettere il testo all'interno di un campo scorrevole, nascondere la barra di scorrimento laterale e controllare da programma lo scorrimento del testo.

Per fare questo ci occorrono un po' di variabili: la posizione corrente di scrolling (`currentPosition`) e la massima posizione raggiungibile (`maxScrollHeight`); ci serve poi un flag per sapere se si deve ricominciare dall'inizio (`restartAtTop`), ed una indicazione di tempo per tenere fermo, all'inizio, lo scrolling, di modo che l'utente possa leggere con calma la parte iniziale del testo (che altrimenti scompare non appena inizia lo scrolling). Il metodo appena visto predispone dei valori adatti alle operazioni previste: si comincia dall'inizio, quindi `currentPosition` è nullo e `restartAtTop` è Vero, ed il tempo quando iniziare lo scrolling è quello attuale (al caricamento della finestra) più tre secondi (tre è appunto il valore di `ABOUTWIN_STARTSCROLLDELAY`). Per quanto riguarda l'altezza massima, c'è un metodo chiamato `size`, che restituisce le dimensioni del rettangolo di spazio occupato dalla visualizzazione della `NSAttributedString`; come questo metodo funzioni, lo ignoro (restituisce il rettangolo occupato all'interno della finestra, ma nessuno ha informato la `NSAttributedString` di questo fatto...). Tuttavia, pragmaticamente, accetto il risultato e lo uso, limitatamente alla parte di altezza, visto che la larghezza è quella della `NSTextView`.

> E infatti: `size` restituisce il rettangolo di spazio occupato 'naturalmente' dal testo, a prescindere dalla sua visualizzazione; qui, per pura fortuna, la dimensione orizzontale del testo è inferiore allo spazio reso disponibile dal campo di testo, per cui la dimensione verticale del testo concide con lo spazio occupato dal testo stesso una volta visualizzato; al momento non ho idea di come stabilire la reale dimensione verticale del testo una volta visualizzato.

BATTI IL TUO TEMPO

Adesso viene la parte interessante. Per eseguire lo scrolling da programma, devo avere la possibilità di eseguire l'aggiornamento della posizione di scrolling ad intervalli prefissati di tempo. In generale, qualsiasi animazione richiede l'esecuzione di un qualche ridisegno ad intervalli determinati di tempo. Esiste in Cocoa una classe pensata proprio allo scopo, ovvero `NSTimer`.

Sostanzialmente, un oggetto `NSTimer` attende un certo tempo, poi fa qualcosa. Tipicamente, invia un messaggio a qualcuno. Se istruito correttamente, continua a mandare il messaggio ad intervalli regolari. È proprio quello che mi serve. Da qualche parte all'interno della classe metterò una istruzione del tipo:

```
scrollTimer = [NSTimer scheduledTimerWithTimeInterval:
ABOUTWIN_TIMERINTERVAL
    target:self
    selector: @selector(scrollCredits:)
    userInfo: nil
    repeats: YES];
```

in cui costruisco un oggetto `NSTimer` con le caratteristiche di inviare il messaggio `scrollCredits:` (argomento di `selector`) a se stesso (argomento di `target`) e di

ripetere (argomento di `repeats`) l'operazione ogni `ABOUTWIN_TIMERINTERVAL` secondi (argomento di `scheduledTimerWithTimeInterval`). Quando devo fermare l'animazione, invio un messaggio del tipo:

```
[ scrollTimer invalidate];
```

che ferma le operazioni del timer.

Il posizionamento di queste due istruzioni introduce un nuovo concetto ancora, o meglio, una nuova notifica. Tra le notifiche inviate automaticamente da una finestra, c'è `NSWindowDidBecomeKeyNotification`, inviata quando la finestra si mette davanti a tutti (ed in particolare è prima destinataria di ogni attività sulla tastiera: questo spiega la parola 'key'). Ugualmente, quando perde questa caratteristica, c'è la notifica `NSWindowDidResignKeyNotification`. Utilizzando una classe delegata della finestra, per rispondere a queste due notifiche occorre scrivere i due metodi `windowDidBecomeKey:` e `windowDidResignKey:`. All'interno di questi due metodi inserisco le due istruzioni, la prima per innescare il timer e l'attività di animazione, la seconda per fermare il timer e l'animazione stessa.

Perché il meccanismo funzioni, bisogna ricordarsi di assegnare `File's Owner` come delegato della finestra di About all'interno del file `AboutBox.nib` (detto così è ovvio, ma provate voi a capire perché le cose non funzionano... ci ho messo un bel po'...).

ANIMAZIONE

Eccomi finalmente al cuore di tutta la faccenda, ovvero al metodo `scrollCredits:`, periodicamente invocato dal meccanismo di `NSTimer`.

```
- (void)
scrollCredits: (NSTimer *)timer
{
    // vediamo se e' passato il tempo di attesa iniziale
    if ([NSDate timeIntervalSinceReferenceDate] < startTime)
        return ;
```

Qui controllo se è passato sufficiente tempo dall'apertura della finestra, per tenere ferma l'animazione per un po' di tempo in modo che si possano leggere le prime righe.

```
    // se arrivo qui, bisogna cominciare a scrollare
    // vedo se devo ripartire dall'inizio
    if (restartAtTop)
    {
        // e' meglio aspettare un po'
                startTime = [NSDate timeIntervalSinceReferenceDate] +
ABOUTWIN_STARTSCROLLDELAY;
        // non devo piu' ricominciare dall'inizio
        restartAtTop = NO;
        // imposto la posizione dall'inizio
        [scrollText scrollPoint:NSMakePoint( 0, 0 )];
        return;
    }
```

Questo è il pezzo di codice in cui si entra quando si deve cominciare a scrollare la

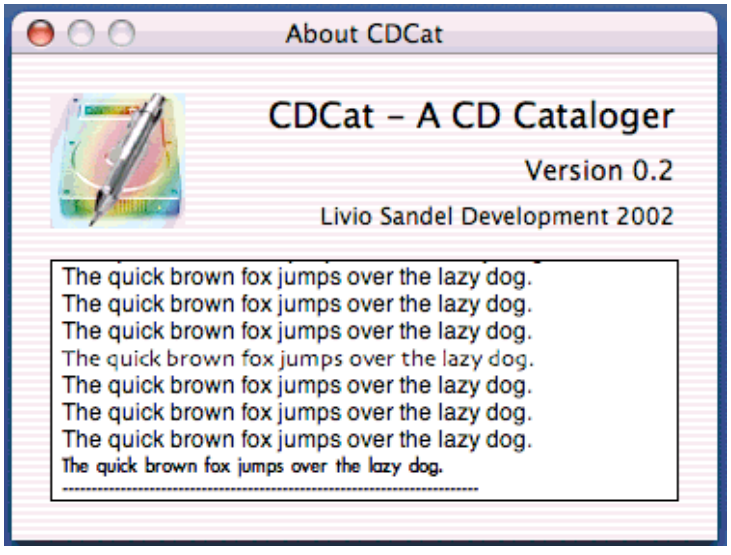
finestra (perché è appena scaduto il tempo di attesa, oppure perché si è arrivati alla fine del giro precedente). Imposto un paio di variabili, e poi dico che la posizione di scroll è Zero. Questa posizione va data come una coordinata orizzontale ed una coordinata verticale, per poter gestire scrolling nelle due direzioni. In questo esempio ci interessa la sola coordinata verticale (la seconda), che imposto a zero. La funzione (è una funzione, non un metodo) `NSMakePoint` costruisce un punto, che è appunto costituito da una coordinata orizzontale ed una verticale.

```
    // se arrivo qui, proseguo lo scrolling
    // vedo se per caso sono arrivato alal fine
    if (currentPosition >= maxScrollHeight)
    {
        // sono arrivato alal fine, reimposto il timer
                startTime  =  [NSDate  timeIntervalSinceReferenceDate]  +
ABOUTWIN_INTERSCROLLDELAY;
        // e riparto dall'inizio
        currentPosition = 0;
        restartAtTop = YES;
        return ;
    }
```

Qui si controlla se lo scrolling è terminato, verificando se la posizione corrente ha superato (o eguaglia) la massima. Le istruzioni elencate impostano una nuova attesa (per permettere la lettura delle ultime righe appena comparse) e le variabili per poter ricominciare dall'inizio.

```
    // se sono arrivato qui, non e' successo nulla di speciale
    // imposto allo scroll la posizione corrente
    [scrollText scrollPoint:NSMakePoint( 0, currentPosition )];
    // che poi incremento per la volta successiva
    currentPosition += ABOUTWIN_SCROLLINCREMENT;
}
```

Infine, questa è la parte che esegue lo scrolling vero e proprio: semplicemente, imposta il valore di scrolling corrente, e pi incrementare di un po' la posizione corrente. Variando il valore di `ABOUTWIN_SCROLLINCREMENT` (ed eventualmente dell'intervallo di attivazione del timer) si possono ottenere diverse velocità di scorrimento del testo.



Per vedere qualcosa di sensato, ho dovuto aggiungere un po' di testo inutile in coda al file `Credits.rft`, altrimenti il testo completo del file stava all'interno della `NSTextView` e non si aveva alcun effetto di scrolling.

Capitolo 025 - AIUTO!

Questo breve capitolo dice come aggiungere Help all'applicazione.

Sorgenti: Documentazione Apple e CocoaDevCentral.

Primo inserimento: 2 settembre 2002

AIUTO

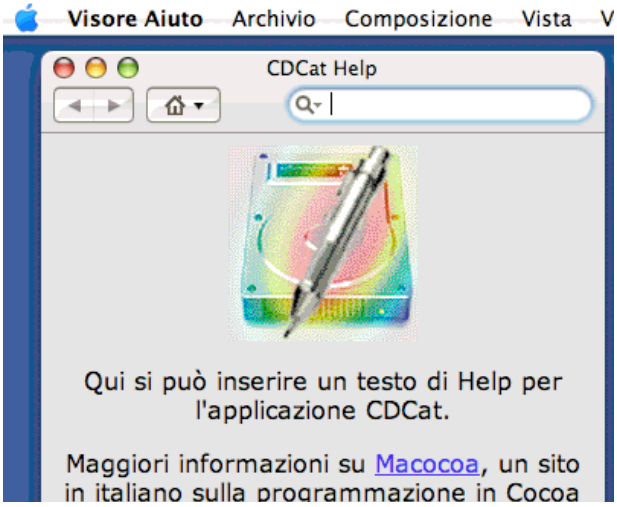
La questione è stata a lungo sofferta, finché non ho affrontato il problema di petto e mi sono messo a leggere la documentazione, dietro suggerimento di un tutorial su Cocoa Dev Central.

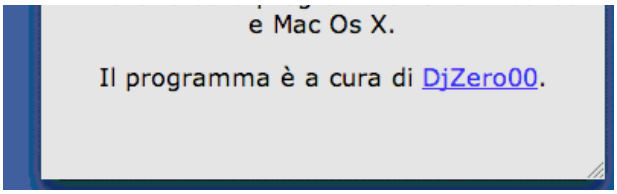
Come dovreste aver notato, Mac OS X ha un meccanismo di Help per le applicazioni che si basa su una normale struttura di file HTML. Attraverso la comune voce di menu `Aiuto` si attiva l'applicazione `Help Viewer`: altri non è che un comune browser HTML (semplificato, e senza troppi fronzoli su Javascript e compagnia), che legge tranquillamente file HTML conformi alla versione 3.2 del linguaggio (adeguata alla maggior parte degli scopi di Help).

Ora, l'architettura Cocoa fornisce di serie tutto l'occorrente per attivare questo Help. Bisogna solo avere un po' di accortezza nella preparazione dei file HTML e poi inserire due/tre voci nella `Info.plist`.

HTML

Per scrivere lo Help di una applicazione, conviene fare riferimento alla documentazione Apple, che voi troverete facilmente seguendo questo indirizzo (sulla documentazione locale): `/Developer/ Documentation/ Carbon/ HumanInterfaceToolbox/ AppleHelp/ ProvidingUserAssitAppleHelp/ index.html` e che invece io, trovandosi nella parte relativa a Carbon, ho inizialmente colpevolmente trascurato. Di particolare interesse la sezione `Designing a Help Book`, dalla quale stralcio velocemente alcuni punti.

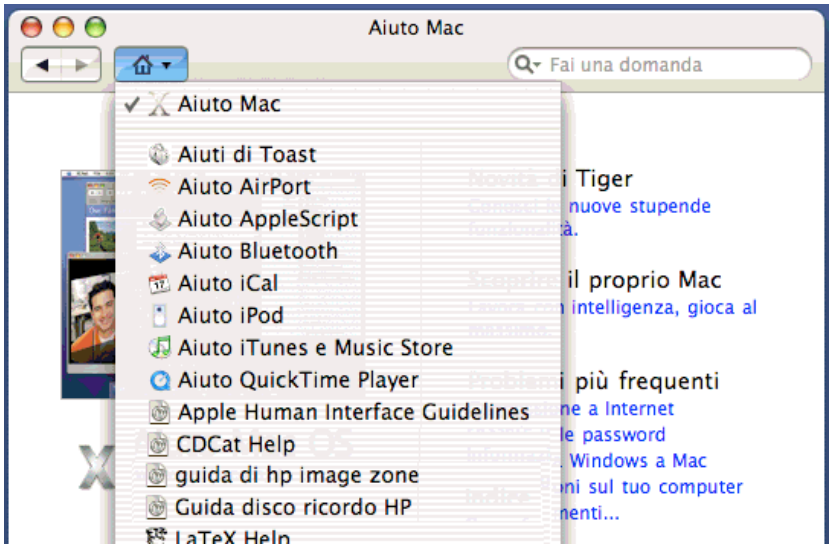




Lo Help è tipicamente formato da una pagina di titolo e da una pagina principale, dalla quale accedere poi a tutte le altre pagine che costituiscono l'indice. Io non mi sono preoccupato troppo di fornire un Help completo, ma, bastandomi per il momento i concetti alla base, mi sono limitato a produrre una pagina di titolo.

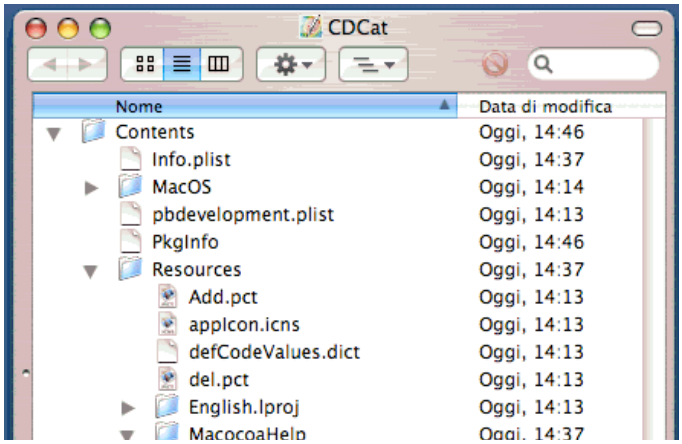
Ed è in questa pagina di titolo che entra il primo trucco per un Help funzionante: occorre aggiungere un meta tag del tipo

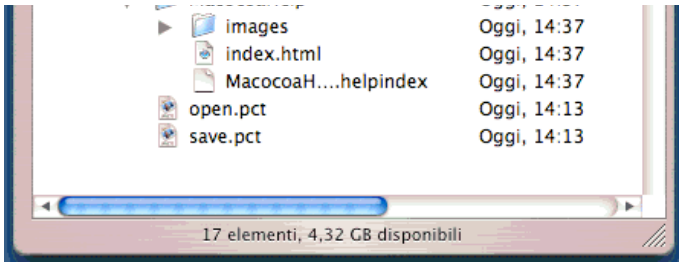
```
<HEAD>
<META NAME="AppleTitle" CONTENT="CDCat Help">
</HEAD>
```



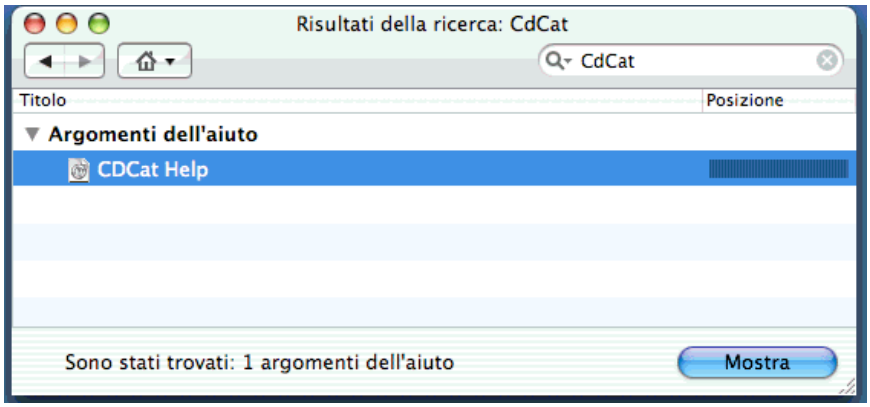
In questo modo, la voce `CDCat Help` sarà mostrata all'interno della lista delle applicazioni con Help presentate dalla pagina principale di `Help Viewer`.

Questa è in effetti l'unica cosa da tenere presente per scrivere le pagine di Help. Le altre direttive (carattere, link, eccetera) sono opzionali, mentre le raccomandazioni per la strutturazione in capitoli sono sensate e ragionevoli, e di nessuna difficoltà.





Il risultato finale della scrittura dello Help comunque sarà bene sia sempre una struttura di questo tipo: tutto lo Help si trova all'interno di una directory, opportunamente chiamata (nel mio caso, `MacocoaHelp`, ma il nome può essere qualunque); all'interno di questa directory, un file, comunque chiamato, che rappresenta la pagina di titolo avente il meta tag sopra indicato; la pagina principale dello Help (che potrebbe essere anche la pagina del titolo); altre sotto directory con dentro le immagini e gli altri file HTML di help.

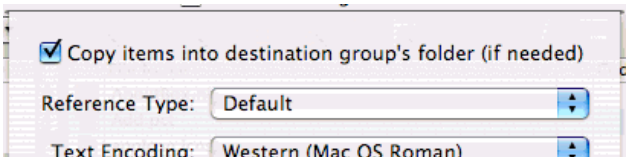


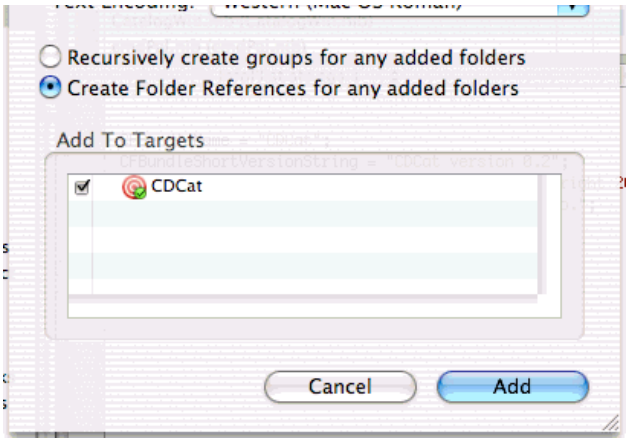
Se proprio si vuole essere professionali, occorre indicizzare lo Help, in modo che quando l'utente dall'interno dello Help Viewer esegue delle ricerche, queste possano produrre risultati anche guardando questi nuovi file di Help. L'operazione è oltre tutto molto semplice: basta droppare l'intera cartella sull'applicazione `Help Indexer` che si trova in `/Developer/Application/Utilities`. Questa operazione produce un file (nel mio caso, `MacocoaHelp.helpindex`) già nel posto corretto.

Tutto ciò funzionava quando scrivevo questa pagina. Tuttavia, nel corso del tempo, le cose si sono modificate, e sembra che la ricerca all'interno del visore aiuto non produca risultati; dovrò indagare più a fondo, ovvero, dovrò leggere la documentazione aggiornata...

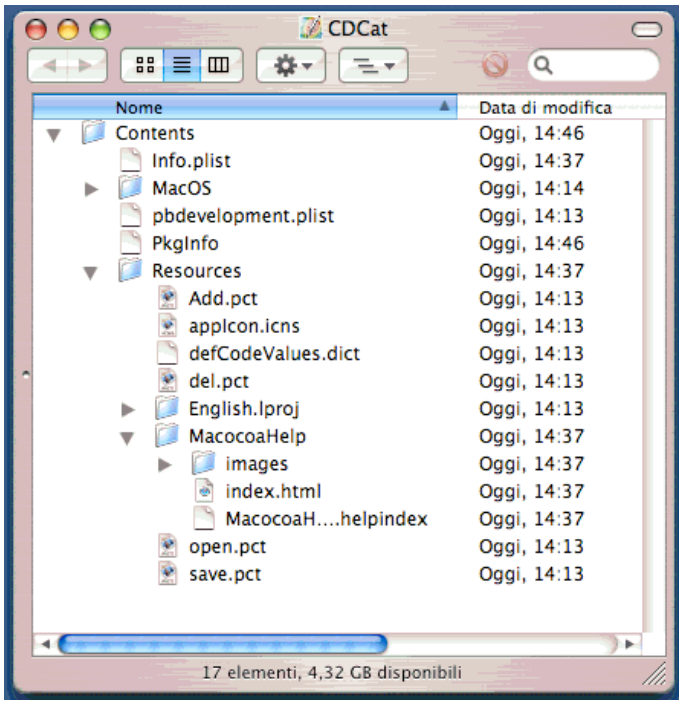
REGISTRAZIONE DELLO HELP

Il passo successivo è informare l'applicazione dell'esistenza di questo Help. Questo si ottiene eseguendo due operazioni: si inserisce la cartella all'interno del progetto, e si aggiungono un paio di voci all'interno della property list. Entrambe le operazioni mi hanno fatto innervosire.

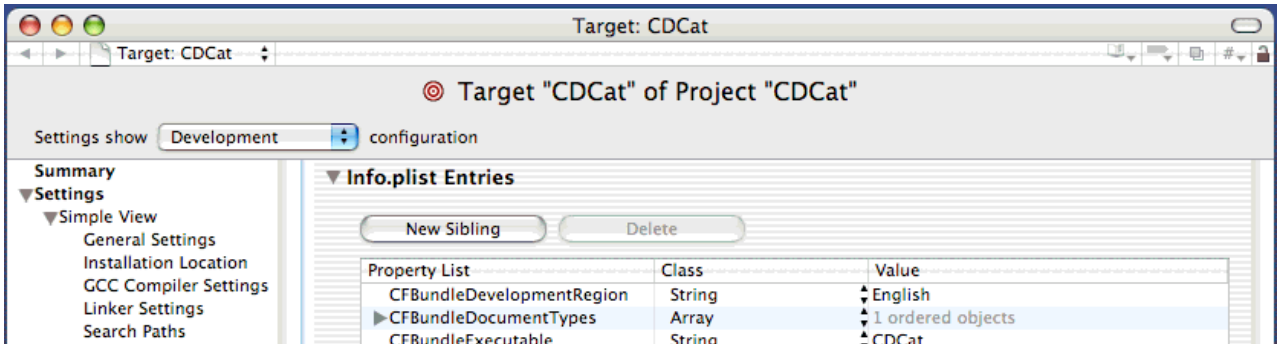


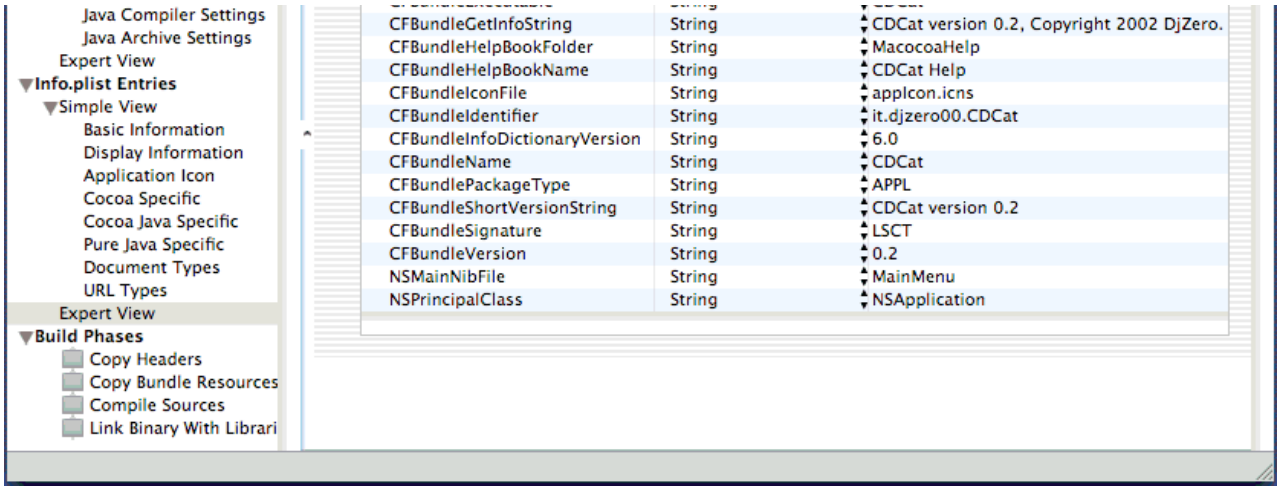


La prima, innocente operazione di aggiungere la cartella dei file al progetto richiede un'avvertenza: nel dialogo che compare ad un certo punto dell'operazione, bisogna selezionare il pulsante `Create Folder References...`; nel mio caso, era automaticamente selezionato l'altro pulsante (che aggiunge i file al progetto,ma non nel posto giusto), cosa che proibiva il corretto funzionamento del meccanismo.



Scopo ultimo di questa operazione è di fare in modo che la cartella e l'intero suo contenuto siano copiate all'interno della cartella `Resources` contenuta all'interno dell'applicazione CDCat finale.





La seconda operazione consiste nell'aggiungere due voci alla `property list` dell'applicazione. Questa property list contiene molte altre informazioni di servizio utili alla normale vita dell'applicazione stessa; ad esempio, il nome del file dove sono contenute le icone. Per impostare valori su questa property list si usa la finestra che riporta le informazioni sul Target. Tuttavia, i campi da aggiungere qui non sono previsti: occorre passare alla modalità `Expert`. Ciò che compare è una tabella che ricorda molto (ma va'?) la finestra dell'applicazione `Property List Editor`. In questa finestra bisogna aggiungere due voci: `CFBundleHelpBookFolder`, con il nome della directory in cui è contenuto l'intero sistema di Help (nel mio caso `MacocoaHelp`), e `CFBundleHelpBookName`, con il content del meta tag `AppleTitle` della pagina di titolo (`CDCat Help`, va da sé). Cocoa Dev Central raccomanda di impostare anche la voce `CFBundleIdentifier`, ma questo era già stato fatto quando ho lavorato con le Preferenze.

Nonostante tutto, le cose non funzionavano ancora, ed un irritante messaggio sul fatto che l'applicazione non aveva un Help associato continuava a saltare fuori. Colto da disperazione, ho spento tutto. Alla riaccensione, tutto funziona. Ho il sospetto che il meccanismo abbia bisogno di pensarci su prima di funzionare (forse basta aspettare un po', senza necessariamente spegnere).

Capitolo 026 - Fritto Misto

Questo capitolo è composto da vari pezzi slegati da loro; ci sono la correzione di alcuni errori, il miglioramento di alcuni comportamenti bizzarri, qualche nuova caratteristica, cose del genere.

Sorgenti: Tanto lavoro di debugger; un esempio di Apple.

Primo inserimento: 2 settembre 2002

Due Errori

Nel codice finora sviluppato, ci sono due errori; meglio, comportamenti non conformi quando si lavora in condizioni eccezionali (sì, mettiamola così...). Entrambi gli errori sono venuti alla luce quando ho tentato di catalogare il contenuto dell'intero hard disk.

Il primo errore è concettuale, ed è dovuto alle caratteristiche di Unix.

Dovete sapere che all'interno del mio hard disk (ma non credo di essere speciale...) c'è una cartella, nascosta, chiamata `/Network` (tra parentesi... ho scoperto una quantità enorme di cartelle nascoste). All'interno di questa cartella sono presenti altre cartelle, ed in particolare c'è una cartella che mantiene collegamenti ai volumi di rete. Poiché con MacOSX lo hard disk è uno dei volumi di rete presenti (si chiama sempre **localhost**), la catalogazione del contenuto dello Hard disk riprendeva daccapo. Questo conduce l'applicazione in un ciclo infinito, che la porta ad esplorare continuamente il contenuto dello hard disk (non è vero: l'applicazione muore clamorosamente dopo un po' di tempo). Il problema sta nella natura dei collegamenti. Quindi, digressione sui collegamenti in Mac OS X.

All'interno del sistema operativo sono possibili tre diversi modi per effettuare un collegamento. Il primo metodo è il classico **Alias**, ereditato dai sistemi operativi precedenti. Altri due collegamenti sono invece ereditati da Unix, e li chiamerò **soft link** e **hard link**. Un hard link (che pare si possa fare solo a file e non a directory) è totalmente indistinguibile dal file collegato. È come avere più copie del file, sempre mantenute sincronizzate ed uguali tra loro. Per effettuare un hard link, avete bisogno del terminale Unix e scrivere l'istruzione `ln <nomefile>`.

Un soft link è invece un collegamento ad un file molto più semplice: creando un soft link viene costruito un file nel cui interno di trova il nome (completo di path) del file collegato. Un soft link è utilizzando anche per effettuare collegamenti verso directory. Anche un soft link si costruisce tramite terminale con il comando `ls -s <nomefile>`. Da evidenziare la differenza tra un soft link ed un alias. Se sposto o rinomino il file puntato, l'alias continua a funzionare, il soft link no. D'altra parte, se sostituisco il file con una nuova versione, l'alias mantiene il riferimento alla vecchia versione, il soft link punta alla nuova. Fine digressione.

Il problema nasce con i soft link che il Mac OS X utilizza a man bassa per rendere le cose più semplici a se stesso, al Finder, all'utente, ai programmatori, insomma a tutti. Ora, il codice di catalogazione non segue i collegamenti dati da un alias perché,

all'interno del codice stesso, quanto si recuperano le informazioni sul file, ho detto di non farlo: è l'istruzione

```
NSDictionary       *fattrs         =  [manager  fileAttributesAtPath:  aFile
traverseLink:NO];
```

presente all'interno del metodo `initWithPath` della classe `LSFileInfo`. Questa istruzione però, segue gli hard link (non può farne a meno) ma soprattutto segue i soft link. Ed è proprio questa cosa che non va bene. Per evitare l'espansione dei soft link mi sono trovato ad usare quel campo delle informazioni di un file che qualche capitolo fa lamentavo di non usare, ovvero `fileType`.

Ho quindi introdotto la seguente condizione

```
if ( [ [self fileType] isEqual: NSFileTypeSymbolicLink ])
    isAdir = FALSE ;
```

all'interno del metodo `initTreeFromPath` della classe `FileStruct`. La condizione dice, molto semplicemente, che se il file è in realtà un soft link, sicuramente non è una directory, o meglio, va trattato come un normale file: insomma, non va espanso. Questo risolve il primo problema.

GIGA GIGA BUM

Il secondo problema si è presentato quando ho catalogato directory il cui contenuto supera i due giga. Anche questo errore è saltato fuori quando ho catalogato l'intero hard disk (dopo aver risolto il problema precedente, ovviamente). Causa dell'errore è che la dimensione prevista dei file era mantenuta all'interno di un `long`, ovvero un numero intero a 32 bit, capace di mantenere una dimensione in byte fino a due giga. In effetti, guardando meglio la documentazione, il metodo che recupera la dimensione di un file non usa un `long`, ma piuttosto un `long long`. A prescindere dal buffo nome, si tratta di un intero a 64 bit, in grado di conservare correttamente dimensioni fino a otto milioni di Terabyte (più che sufficienti per qualche anno a venire). Cambiare la dimensione della variabile d'istanza da `long` a `long long` è parsa la più ovvia e semplice delle soluzioni, se non fosse che si è portata dietro un altro po' di problemi.

In primo luogo, occorre modificare anche i metodi `encodeWithCoder` e `initWithCoder` (cosa che, ancora una volta, rende incompatibili i file salvati su disco in precedenza), per non parlare dei metodi accessor. Poi, nella classe `FileStruct`, c'è da promuovere a `long long` la variabile che tiene conto delle dimensioni della directory man mano che si aggiungono file. Un grosso problema si è presentato nella classe `LSDataSource`, perché, a quanto pare, il meccanismo `valueForKey` non funziona con una variabile di tipo `long long` (o almeno, così sembra a me; mi pare strano...).

```
- (id)
outlineView:                (NSOutlineView *) outlineView
    objectValueForTableColumn:    (NSTableColumn *) tableColumn
    byItem:                (id)item
{
```

```
    NSString * colId ;

    // se l'oggetto e' vuoto, c'e' qualche problema...
    if (item == nil)
        return ( @"????" ) ;
    // recupero l'identificatore della colonna
    colId = [ tableColumn identifier] ;
    // e da qui l'elemento che mi serve
    if ( [ colId isEqual: COLID_FILESIZE ] )
    {
        long long    tmp ;
        tmp = [ item fileSize ] ;
        return ( [ NSNumber numberWithLongLong: [ item fileSize ]] );
    }
    return ( [ item valueForKey: colId ] ) ;
}
```

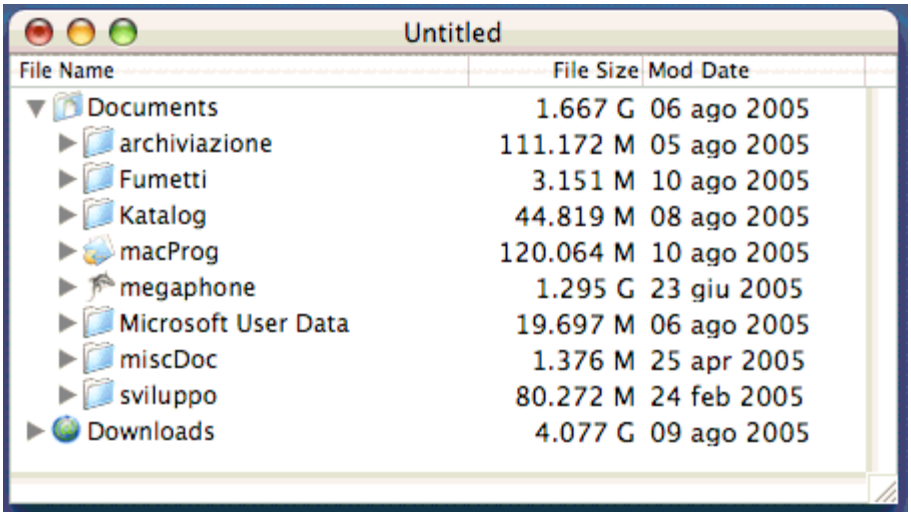
Senza saper né leggere né scrivere (ma soprattutto programmare), ho deciso di non perdere troppo tempo sull'argomento e di inserire il caso specifico della dimensione del file per... per costruire un oggetto di tipo `NSNumber`. Infatti, il metodo serve per restituire il valore di una casella della `NSOutlineView`, valore che poi passa attraverso un formatter. Il formatter, per funzionare, si aspetta un generico oggetto da cui estrarre un valore. Il metodo più spiccio per inviare un valore numerico è allora di produrre un oggetto `NSNumber` con il valore corretto.

Va da sé che anche il formatter va modificato, perché estragga `long long` piuttosto che `long`. E qui, già che c'ero, ho aggiunto anche la visualizzazione in Giga quando la dimensione lo consente.

```
- (NSString *)
stringForObjectValue:    (id)anObject
{
    unsigned long long    fSize, fSizeK ;
    float             fsizeM, fsizeG ;

    // controllo che l'oggetto sia un numero...
    if (![anObject isKindOfClass:[NSNumber class]]) {
        return nil;
    }
    // ricavo la dimensione del file
    fSize = [ ((NSNumber*) anObject) longLongValue ];
    // se il file e' piccolo, mostro byte
    if ( fSize < 1024 )
    {
        long    tmp = fSize ;
        return ( [ NSString stringWithFormat: @" %4d b", tmp] );
    }
    // la dimensione e' in byte, divido per 1024 ed arrotondo, ottengo K
    fSizeK = (long) (( fSize / 1024.0 ) + 0.5 );
    // se il file e' medio, mostro K
    if ( fSizeK < 1024 )
    {
        long    tmp = fSizeK ;
        return ( [ NSString stringWithFormat: @" %4d K", tmp] );
    }
    // se inferiore al giga, restituisco mega con tre decimali
    fsizeM = ( fSizeK / 1024.0 ) ;
    if ( fsizeM < 1024 )
        return ( [ NSString stringWithFormat: @" %6.3f M", fsizeM] );
```

```
    //negli altri casi, ritorno Giga, con tre cifre decimali
    fsizeG = ( fsizeM / 1024.0 ) ;
    return ( [ NSString stringWithFormat: @" %6.3f G", fsizeG] );
}
```

Se poi volete rendermi infelice, chiedetemi perché all'interno di una coppia di parentesi graffe utilizzo una variabile temporanea `long` per la scrittura della stringa. A quanto pare, non esiste una specifica di formato per i `long long` (improbabile), oppure, non sono riuscito a trovarla (probabile).



C'è un'ultima cosa, che non c'entra con l'errore, ma che pertiene alla visualizzazione: quando installo il formatter della dimensione del file, ho pensato di allineare la colonna a destra piuttosto che lasciare l'allineamento a sinistra di default (sono nella funzione `attachFormatter` del file `CatalogDoc.m`):

```
#### codice codice ####
if ( [ colId isEqual: COLID_FILESIZE ] )
{
     FileSizeForm      *myDF2 = [[[ FileSizeForm alloc ] init ] autorelease
];
    [[tc dataCell ] setFormatter: myDF2 ];
    // gia' che ci sono, sbandiero a destra
    [[tc dataCell ] setAlignment: NSRightTextAlignment];
    [[tc headerCell ] setAlignment: NSRightTextAlignment];

    return ;
}
#### codice codice ####
```

L'ICONA DEL FILE

Peregrinando nella documentazione, mi sono imbattuto in una classe e un metodo molto interessante; qualche capitolo fa mi crucciavo di non riuscire a recuperare l'icona del file. Ecco, ho trovato il modo per leggerla e conservarla.

Esiste una classe denominata `NSWorkspace`; con questa classe si può interagire in modo più o meno compiuto con l'ambiente operativo. In pratica l'unico oggetto di questa classe costruito automaticamente al lancio dell'applicazione (e che si trova col

metodo di classe `sharedWorkspace`) è in grado di fornire servizi adatti a:

1. la manipolazione di file (copia, sposta,...) e il recupero di informazioni accessorie sugli stessi;

2. riconoscere eventi globali come modifiche al file system, dispositivi e utenti (ad esempio, si è reso disponibile un nuovo Volume, che so, un CD);

3. lanciare applicazioni dall'interno dell'applicazione.

Di tutto questo, al momento mi interessa un solo metodo,

```
- (NSImage *)iconForFile:(NSString *)fullPath
```

che restituisce, come un oggetto della classe `NSImage`, l'icona posseduta dal file.

Scoperto questo, aggiungere all'interno della classe `LSFileInfo` la variabile d'istanza `fileIcon`, i metodi accessor relativi, il recupero e la memorizzazione dell'icona del file è stato un gioco da ragazzi. Voglio solo riportare qualche riga di codice del metodo `initWithPath:`

```
// prelevo l'icona del file
tmpImg = [ [ NSWorkspace sharedWorkspace] iconForFile: aFile ] ;
// dico di scalarla quando sara' ridimensionata
[ tmpImg setScalesWhenResized: TRUE ];
// perche' adesso la ridimensiono a 16x16
smallIconSize.width = smallIconSize.height = 16 ;
// ecco che la ridimensiono
[ tmpImg setSize: smallIconSize];
[ self setFileIcon: tmpImg ];
```

L'idea è di ridimensionarla subito alla grandezza di 16 bit di larghezza per 16 bit di altezza, perché utilizzerò sempre queste dimensioni. Faccio tutto ciò da programma, dicendo dapprima di mantenerla proporzionata durante le operazioni di ridimensionamento, poi ridimensionandola appunto a 16x16, per infine assegnarla alla variabile d'istanza.

Aggiungo alcune considerazioni a questo argomento nel prossimo capitolo.

Il passo successivo, quello più semplice, consiste nel visualizzare l'icona nella finestra delle Info. Allo scopo, l'ho modificata facendo spazio per un'oggetto della classe `NSImageView`, di dimensioni appunto 16x16, inserendo un apposito outlet, e modificando di concerto il metodo `selectionChanged:`. Non trovate qui il codice perché il metodo subirà fra un momento una profonda revisione.

Arriva adesso la cosa teoricamente più difficile: far comparire l'icona all'interno della `NSOutlineView`, di fianco al nome del file. Eppure, il compito si è rivelato facilissimo, ma non per merito mio. In cerca di ispirazione, mi sono imbattuto nell'esempio `DragNDropOutlineView`, fornito a corredo dell'installazione di XCode. Ebbene, la cosa è già fatta! Nell'esempio, il programmatore ha costruito una classe proprio a questo scopo. Copio pedissequamente proprio i due file, `ImageAndTextCell.h` e `ImageAndTextCell.m`, e li uso brutalmente. Le bellezze della programmazione

object-orientes stanno anche qui, nel fatto che ignoro bellamente il contenuto dei file (la realizzazione della classe) e mi interesso solamente della sua interfaccia come usare la classe).

Il procedimento è il seguente: una `NSOutlineView` contiene diverse colonne; ogni colonna è costituita da un insieme di oggetti `NSCell`, uno per ogni riga, responsabili della visualizzazione del contenuto. L'oggetto `NSCell` utilizzato di default è in grado di visualizzare testo; la classe `ImageAndTextCell` definisce una sottoclasse di `NSCell` (anzi, di `NSTextFieldCell`, la cella che mostra il testo) e fa in modo di visualizzare non solo il testo, ma anche una immagine che qualcuno ha provveduto ad inserire all'interno dell'oggetto come variabile d'istanza.

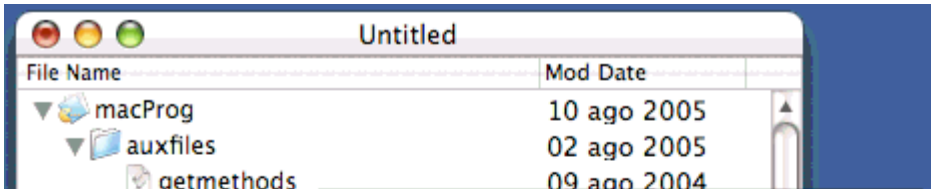
Per effettuare questo scambio di celle, occorre scrivere qualche riga di codice all'interno del metodo `windowControllerDidLoadNib` della classe `CatalogDoc`, lì dove si predispone il funzionamento della `NSOutlineView` (anche questo segmento di codice è stato copiato, mutatis mutandis, dall'esempio predetto):

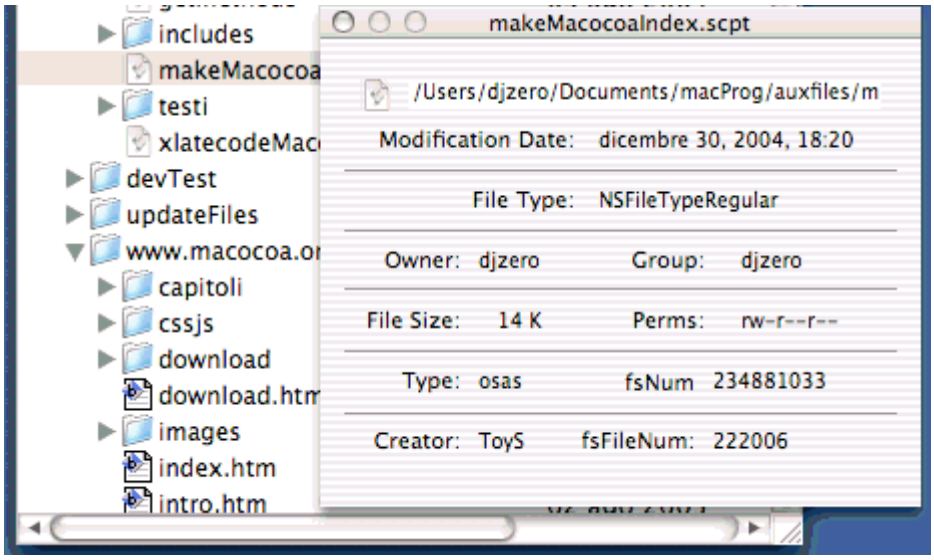
```
tableColumn = [outlineView tableColumnWithIdentifier: COLID_FILENAME];
imageAndTextCell = [[[ImageAndTextCell alloc] init] autorelease];
[imageAndTextCell setEditable: NO];
[tableColumn setDataCell:imageAndTextCell];
```

Si ricava l'oggetto `NSTableColumn` responsabile della visualizzazione della colonna, si costruisce la cella con la nuova classe, si assegna tale cella come deputata alla visualizzazione (di passaggio, si dice anche che la cella non gestisce la modifica del contenuto).

Da qualche porte, infine, bisogna inserire l'immagine all'interno della cella. Per fare ciò, copio ancora una volta dall'esempio. Tra le varie notifiche che la `NSOutlineView` invia, e che la classe delegata può realizzare, ce ne è una che dice: 'guarda che mi sto preparando a visualizzare questa cella...'. Basta intercettare questa notifica, e se il caso impostare l'immagine:

```
- (void)
outlineView:              (NSOutlineView *) outlineView
    willDisplayCell:     (NSCell *)cell
    forTableColumn:         (NSTableColumn *) tableColumn
    item:             (id)item
{
    if ( [[tableColumn identifier] isEqualToString: COLID_FILENAME] )
    {
        // Set the image here since the value returned from
           // outlineView:objectValueForTableColumn:... didn't specify the
image part...
        [((ImageAndTextCell*) cell) setImage: [item fileIcon] ];
    }
    // negli altri casi, faccio nulla
}
```





La cosa è molto facile: se la cella appartiene alla colonna che mostra il nome del file, inserisco nella cella l'immagine corrispondente all'elemento in corso di visualizzazione. Dopo essersi ricordati di impostare la classe delegata della `NSOutlineView` (si può fare direttamente in IB, nel file `CatalogWin.nib`, collegando la finestra col `File's Owner`, che è un oggetto della classe `CatalogDoc`), le cose funzionano a meraviglia.

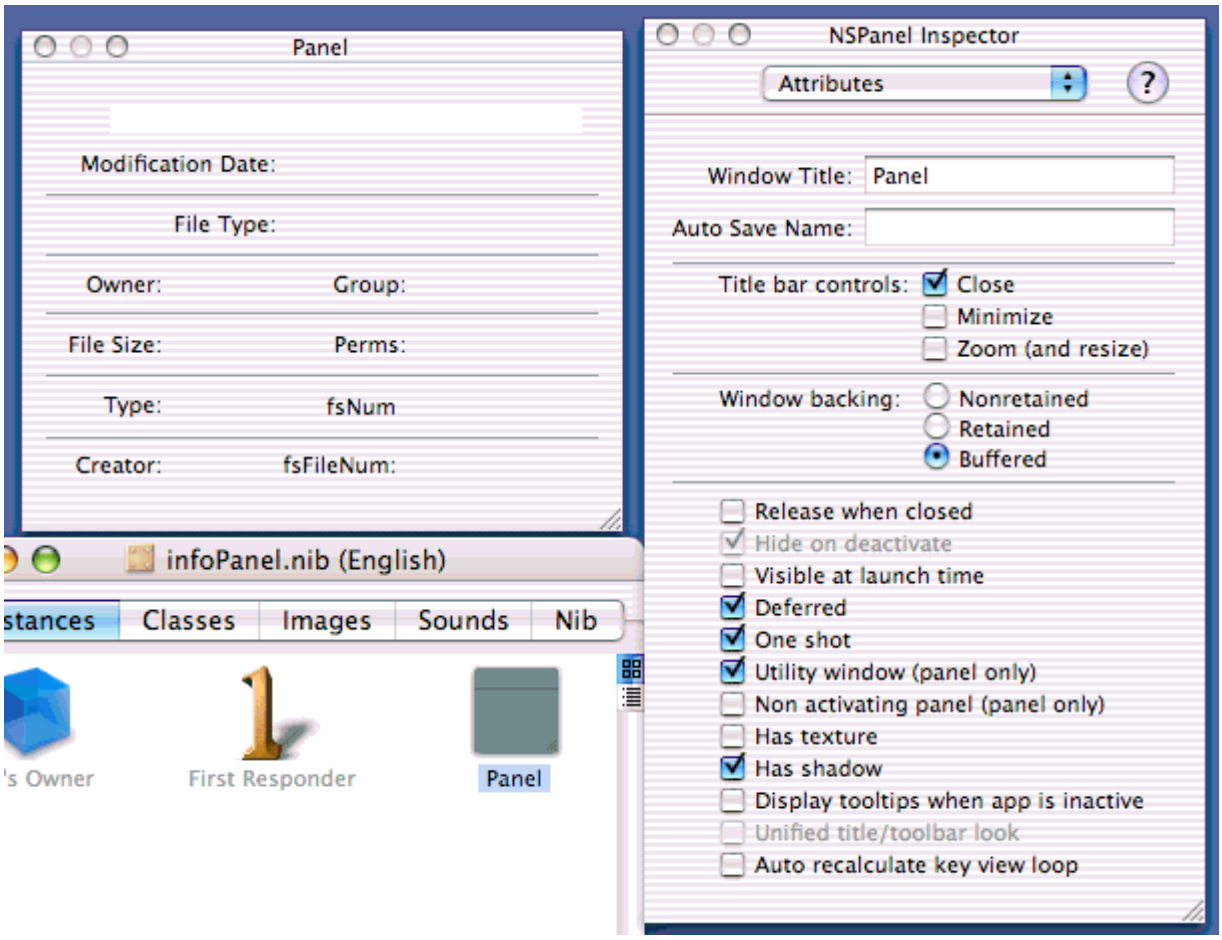
Alternativamente, posso sfruttare il metodo `outlineView:objectValueForTableColumn:byItem:`, già presente all'interno della classe `LSDataSource`. È il metodo che fornisce i valori da visualizzare. Intercettando la richiesta relativa al nome del file, inserisco in quel momento l'immagine come variabile d'istanza:

```
- (id)
outlineView:                    (NSOutlineView *) outlineView
    objectValueForTableColumn:      (NSTableColumn *) tableColumn
    byItem:                     (id)item
{
    NSString * colId ;

    // se l'oggetto e' vuoto, c'e' qualche problema...
    if (item == nil)
        return ( @"????" ) ;
    // recupero l'identificatore della colonna
    colId = [ tableColumn identifier] ;
    // e da qui l'elemento che mi serve
#if    1
    if ( [colId isEqualToString: COLID_FILENAME] )
    {
        // Set the image here since the value returned from
           // outlineView:objectValueForTableColumn:... didn't specify the
image part...
            [((ImageAndTextCell*) [tableColumn dataCell]) setImage: [item
fileIcon] ];
    }
#endif

    if ( [ colId isEqual: COLID_FILESIZE ] )
    {
        long long     tmp ;
        tmp = [ item fileSize ] ;
```

```
        return ( [ NSNumber numberWithLongLong: [ item fileSize ]] );
    }
    return ( [ item valueForKey: colId ] ) ;
}
```

Tutto sommato, preferisco questa seconda versione, che non sparpaglia troppo il codice nei vari metodi.

LA FINESTRA DI INFO



Lavorando con la finestra di Info (ma anche con la palette dei comandi) mi sono accorto di alcune cose piuttosto fastidiose. La prima è che le due finestre rimanevano davanti a tutte anche se l'applicazione non era attiva. Mi sono dimenticato di impostare a TRUE il flag `HideOnDeactivate` su entrambe le finestre. Per far ciò, basta usare IB e visualizzare le informazioni della finestra.

La cosa è ora eseguita in modo automatico se la finestra è dichiarata essere Utility Panel

L'altro problema fastidioso è che la finestra delle Info non reagisce correttamente all'apertura (visualizza sempre nulla, anche se c'è un elemento selezionato nella finestra di catalogo) ed allo scambio di finestra (passando da una finestra di catalogo ad un'altra, la finestra delle info non si adegua di conseguenza, ma mantiene le informazioni dell'ultimo elemento cliccato, e non dell'elemento correntemente selezionato).

La cosa si risolve (mi pare) tenendo conto di alcuni eventi (notifiche) che si verificano

durante la manipolazione delle finestre. Estendo in pratica gli abbonamenti alle notifiche notevoli (sono nel metodo `windowDidLoad` della classe `InfoWinCtrl`)

```
// dico che il winctrl osserva le notifiche riguardanti
// 1. il cambiamento di selezione di una outlineView
// eseguendo il metodo selectionChanged:
[[ NSNotificationCenter defaultCenter] addObserver: self
    selector: @selector( selectionChanged: )
    name: NSOutlineViewSelectionDidChangeNotification object: nil ] ;
// 2. il cambiamento di finestra
// eseguendo il metodo mainWindowChanged:
[[ NSNotificationCenter defaultCenter] addObserver: self
    selector: @selector( mainWindowChanged: )
    name: NSWindowDidBecomeMainNotification object: nil ] ;
// 3. la chiusura di finestra
// eseguendo il metodo mainWindowChanged:
[[ NSNotificationCenter defaultCenter] addObserver: self
    selector: @selector( mainWindowResigned: )
    name: NSWindowDidResignMainNotification object: nil ] ;
```

Nella versione precedente del metodo era presente solamente la prima sottoscrizione: la finestra delle Info reagisce solo quando qualcuno cambia la selezione all'interno della `NSOutlineView` corrente. La seconda sottoscrizione fa in modo che sia attivato il metodo `mainWindowChanged:` quando una finestra di catalogo è portata in primo piano (in particolare ,quanto è aperta nuova nuova). La terza sottoscrizione attiva il metodo `mainWindowResign:` quando una finestra di catalogo non è più la finestra principale, perché lo è diventata un'altra oppure perché è stata chiusa.

I tre metodi sono molto simili tra loro, e svolgono per la gran parte le stsse operazioni. Ho allora raccolto in un unico metodo le operazioni di aggiornamento della finestra:

```
- (void)
updateInfo: (NSOutlineView *) outView
{
    long        row ;
    FileStruct    * locItem ;
    // vedo se ci sono selezioni in corso
    if ( outView != nil )
        row = [ outView selectedRow] ;
    else     row = -1 ;
    // se non ce ne sono, faccio nulla
    if ( row == -1 )
    {
        // ripulisco la finestra
        [ fileType setStringValue: @""];
        [ groupName setStringValue: @""];
        [ ownerName setStringValue: @""];
        [ fullPath setStringValue: @""];
        [ fsFileNum setStringValue: @"" ];
        [ fsNum setStringValue: @"" ] ;
        [ fileSize setObjectValue: nil ];
        [ osCreator setObjectValue: nil ] ;
        [ osType setObjectValue: nil ];
        [ posixPerm setObjectValue: nil ];
        [ modDate setObjectValue: nil ];
        [ fileIcon setObjectValue : nil ];
        [[ self window ] setTitle: @"Info" ] ;
        return ;
    }
```

```
    // altrimenti, uso l'elemento selezionato
    locItem = [ outView itemAtRow: row ] ;
    // se l'oggetto e' reale, aggiorno il contenuto
    [ fileType setStringValue: [ locItem fileType]];
    [ groupName setStringValue: [ locItem ownGroupName]];
    [ ownerName setStringValue: [ locItem ownerName]];
    [ fullPath setStringValue: [ locItem fileFullPath]];
    // piglio l'intero e lo traformo in stringa
      [ fsFileNum setStringValue: [ NSString stringWithFormat: @"%d", [
locItem fsFileNum]] ];
     [ fsNum setStringValue: [ NSString stringWithFormat: @"%d", [ locItem
fsNum]]] ;
     // questi hanno un formattatore appiccicato, gli passo direttamente
l'intero
      [ fileSize setObjectValue: [NSNumber numberWithLongLong: [ locItem
fileSize] ]];
    [ osCreator setIntValue: [ locItem creatorCode]] ;
    [ osType setIntValue: [ locItem typeCode] ];
    [ posixPerm setIntValue: [ locItem filePosixPerm] ];
     // qui, con il formattatoredi data, gli passo direttamente l'oggetto
NSDate
    [ modDate setObjectValue: [ locItem modDate]];
    // recupero l'icona del file
    [ fileIcon setObjectValue : [locItem fileIcon] ];
    // infine, il nome del file e' il titolo della finestra
    [[ self window ] setTitle: [ locItem fileName ]] ;
}
```

Qui non c'è molto da dire (è il vecchio metodo `selectionChanged:`, con l'aggiunta della pulizia della finestra se non sono selezionati elementi) se non notare l'impostazione dell'icona del file.

I tre metodi sottoscrittori sono invece i seguenti.

Per primo il nuovo metodo `selectionChanged:`; estrae il nuovo elemento da visualizzare e poi invoca l'aggiornamento della `NSOutlineView` argomento della notifica:

```
- (void )
selectionChanged: ( NSNotification *) notification
{
    [ self updateInfo: [ notification object ] ];
}
```

Quando invece la finestra scompare o passa in secondo piano, pulisco la finestra delle info:

```
- (void)
mainWindowResigned:(NSNotification *) notification
{
    [self updateInfo: nil];
}
```

Più complicato l'aggiornamento delle Info quando si presenta una nuova finestra. Seguo passo passo la catena dei messaggi: ricavo dapprima quale finestra è passata in primo piano con il messaggio `object` inviato all'argomento `notification`; dalla finestra, recupero l'oggetto controllore (metodo `windowController`) per poi arrivare

finalmente al documento `CatalogDoc`. Il documento mi serve per rintracciare (attraverso un outlet, per il quale ho dovuto definire metodi accessor) la `NSOutlineView`; finalmente, posso utilizzare questo oggetto per invocare l'aggiornamento della finestra delle info:

```
- (void )
mainWindowChanged: ( NSNotification *) notification
{
        [ self updateInfo: [[[ notification object] windowController]
document] outlineView] ];
}
```

In realtà, nella maggior parte dei casi la notifica `resign` è ridondante: se chiudo una finestra di catalogo, normalmente diventa principale un'altra finestra di catalogo, e sarà il metodo `mainWindowChanged:` che aggiorna la finestra delle Info di conseguenza. L'unico caso in cui il metodo `mainWindowResign:` ha un uso preciso è quando non ci sono più finestre presenti.

SALVATAGGIO FILE

Mi sono accorto che manca una delle funzioni caratteristiche della gestione dei documenti, ovvero la richiesta di salvataggio di un file che è stato modificato. Come tutte le buone applicazioni, occorre in qualche modo visualizzare lo stato corrente del documento. All'inizio il documento è creato vuoto, e quindi non richiede alcun salvataggio. Non appena si comincia a lavorarci sopra, aggiungendo o togliendo elementi dal catalogo, il documento si *sporca*; è bene allora, prima di chiudere la finestra o abbandonare l'applicazione, chiedere all'utente cosa fare del documento sporcato. Un documento è sporco se non è mai stato salvato su disco, oppure se il contenuto di ciò che è salvato su disco differisce da quanto rappresentato.

Cocoa fornisce a corredo del paradigma `NSDocument` un meccanismo molto semplice per tenere traccia delle modifiche; si tratta di utilizzare il metodo `updateChangeCount:` passandogli come argomento un valore predefinito che indica come lo stato del documento si è modificato. Nel mio caso, per il momento, utilizzo sempre e solo la seguente istruzione:

```
[ self updateChangeCount: NSChangeDone ] ;
```

all'interno dei metodi della classe CatalogDoc che si occupano di modificare il numero degli elementi presenti all'interno di una finestra di catalogo, ovvero `addFileItem:`, `delItem:` e `performDragOperation:`.

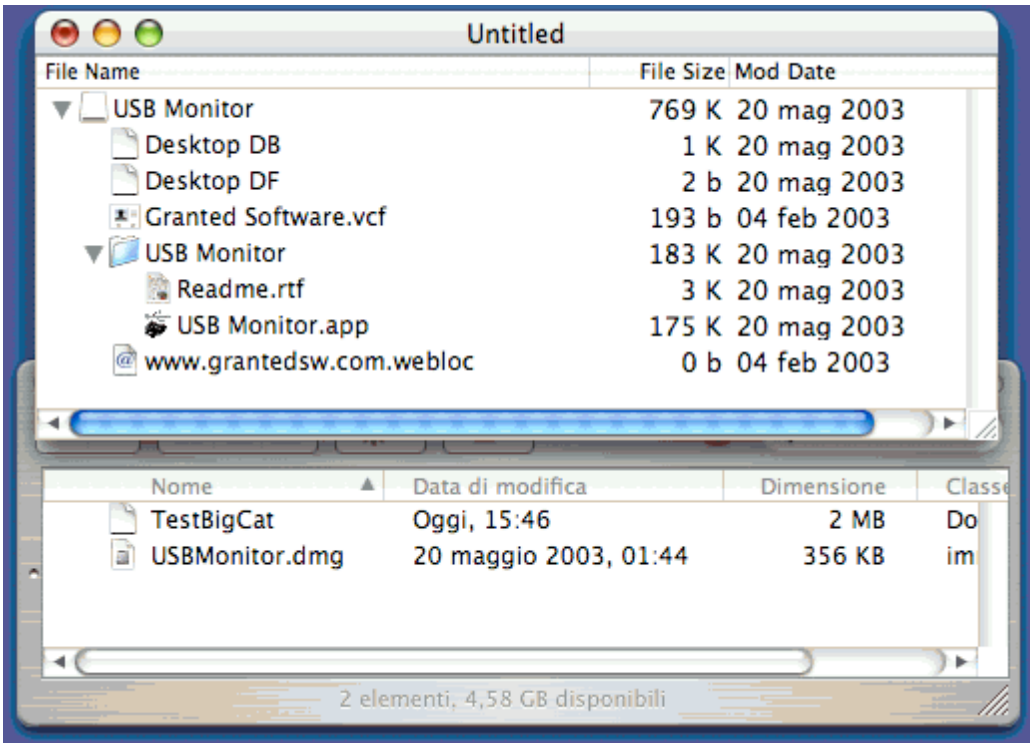
Capitolo 027 - Solo Volumi, per favore

In questo capitolo, comincio con un po' di pulizia, discutendo come mai le dimensioni dei file catalogo sono enormi, e poi ripulisco un po' il codice introducendo (come mio solito) un file dove raccolgo le cose che non so dove mettere altrimenti. Dopo di che, l'argomento principe del capitolo è la limitazione della catalogazione ai soli volumi (ovvero, hard disk, cd ed altri rimovibili in generale), piuttosto che catalogare qualsiasi cosa che somigli ad un file. Nel fare questo, scopro tutta una serie di belle cose, utilizzando ancora classi interessanti come `NSWorkspace`, introducendo il concetto di `sheet` ed utilizzando perfino una barra da barbiere.

Sorgenti: Documentazione Apple, pazienza e fantasia.

Primo inserimento: 2 settembre 2002

Dimensioni Enormi



Quando, dopo l'ultimo capitolo, ho salvato un documento di tipo catalogo, ho scoperto che la dimensione del file era piuttosto grossa. Direi enorme. Ad esempio, catalogando un semplice volume (è un file .dmg di un programmino che ho scaricato da internet...), mi risulta un file di 2 Mega, addirittura di dimensioni maggiori del volume stesso. Dove sta il problema? Dopo lunghe considerazioni che risparmio al lettore, scopro che la quasi totalità della dimensione del file è occupato dalle icone associate ai file (bella scoperta: prima che salvassi anche le icone, le dimensioni dei file catalogo erano ragionevoli). In effetti, quando si recupera l'icona di un file da disco, è creato un oggetto della classe `NSImage`. Questo oggetto non contiene in realtà l'immagine dell'icona; o meglio,

contiene una o più *rappresentazioni* dell'immagine come oggetti della classe `NSImageRep`. Ho scoperto che l'icona raccolta dal metodo `iconForFile:` è composta da più rappresentazioni, tra cui una di 128 pixel per 128; considerato che per ogni pixel ci vogliono 4 byte per specificarne il colore (32 bit), ovvero occorrono 64K di memoria per conservare una singola icona. Catalogate un po' di file, ed ecco che si raggiungono dimensioni di catalogo impressionanti (anche perché le icone non sono riciclate: un file all'interno del catalogo ha sempre la sua icona, anche se del tutto uguale e quella di un altro file). È giocoforza tentare di ridurre lo spazio occupato da un catalogo eliminando le rappresentazioni delle icone che non mi interessano. Allo scopo ho scritto una funzione (non un metodo!) che appunto esamina le varie rappresentazioni di una `NSImage` e tiene buona solo quella di dimensioni minori. Già che ci sono, svolgo qui dentro le operazioni di scalatura a 16x16 pixel:

```
void
reduceImageToIcon( NSImage * img )
{
    NSSize        smallIconSize ;
#if    REDUCE_ICON_ON_FILE
    int        minSize, minImgIndex, i ;
    NSImageRep     * imageRep ;
    // recupero la lista delle rappresentazioni
    NSArray        * tmp = [ img representations ] ;
    // per ora la rappresentazione minima e' la prima
    minImgIndex = 0 ;
    imageRep = [ tmp objectAtIndex: minImgIndex ];
    minSize = [ imageRep pixelsHigh ];
    // poi guardo tutte le altre rappresentazioni
    for ( i = 1 ; i < [ tmp count ] ; i ++ )
    {
        int        hh ;
        imageRep = [ tmp objectAtIndex: i ];
        // se la dimensione della rappresentazione corrente...
        hh = [ imageRep pixelsHigh ];
        // ...e' maggiore
        if ( hh >= minSize )
        {
            // la rappresentazione non mi interessa e la elimino
            [ img removeRepresentation: imageRep];
        }
        else    // ... se invece e' minore
        {
            // ho trovato una nuova rappresentazione minima
            // butto via quella precedente
             [ img removeRepresentation: [ tmp objectAtIndex: minImgIndex ]
];
            // mi segno questa rappresentazione
            minImgIndex = i ;
            minSize = hh ;
        }
    }
    // arrivato qui, dovrei avere una sola rappresentazione, la più piccola
#endif
    // dico di scalarla quando sara' ridimensionata
    [ img setScalesWhenResized: TRUE ];
    // perche' adesso la ridimensiono a 16x16
    smallIconSize.width = smallIconSize.height = 16 ;
    // ecco che la ridimensiono
    [ img setSize: smallIconSize];
}
```

Il codice è una classica ricerca di minimo, con la variante che non appena trovo qualcosa più grande del minimo, lo elimino brutalmente.

Con questo metodo, inserito opportunamente all'interno del metodo `initWithPath:` della classe `LSFileInfo`:

```
// prelevo l'icona del file
tmpImg = [ [ NSWorkspace sharedWorkspace] iconForFile: aFile ] ;
// qui ci sono molte "rappresentazioni" dell'icona...
reduceImageToIcon( tmpImg );
// e poi l'assegno all'interno del file
[ self setFileIcon: tmpImg ];
```

le dimensioni dei file catalogo salvati su disco sono ancora importanti, ma decisamente più piccole (68K vs 2 Mega).

Una volta scritto il metodo, mi sono posto il problema di dove piazzarlo, ovvero, all'interno di quale file mantenerlo; dopo tutto, non si tratta di un metodo di una classe, ma di una funzione nemmeno troppo legata all'applicazione... Per il momento, come faccio di solito, lo piazzo all'interno di un file (una coppia di file, in realtà) che chiamo `djZeroUtils.m` e `djZeroUtils.h`. In questi file metterò tutto quello che non si riferisce ad una classe in particolare, ma è patrimonio comune dell'applicazione. Ad esempio, ho già trovato per un paio di `#define` di compilazione (utili per sperimentare sul codice senza perdere pezzi già funzionanti) ed una macro che utilizzo genericamente all'interno dell'applicazione per la costruzione di metodi accessor:

```
#define        OBJ_ACC_SET( var, new )        \
    { [new retain]; [var autorelease]; var = new; }
```

Infine, leggendo qui e lì la documentazione Apple, scopro che esiste già una funzione per la conversione dei codice tipo/creatore propri di ogni file all'interno dei sistemi Mac OS 9 e precedenti; quindi, ho modificato la classe formatter `TOS9TCForm` come segue:

```
- (NSString *)
stringForObjectValue:    (id)anObject
{
    // controllo che l'oggetto sia un numero...
    if (![anObject isKindOfClass:[NSNumber class]]) {
        return nil;
    }
    // la documentazione Apple afferma che e' meglio usare la funzione
```

```
    // sotto indicata piuttosto che fare da soli...
    return ( NSFileTypeForHFSTypeCode( [ anObject longValue ]) );
}
```

molto più compatta della mia realizzazione.

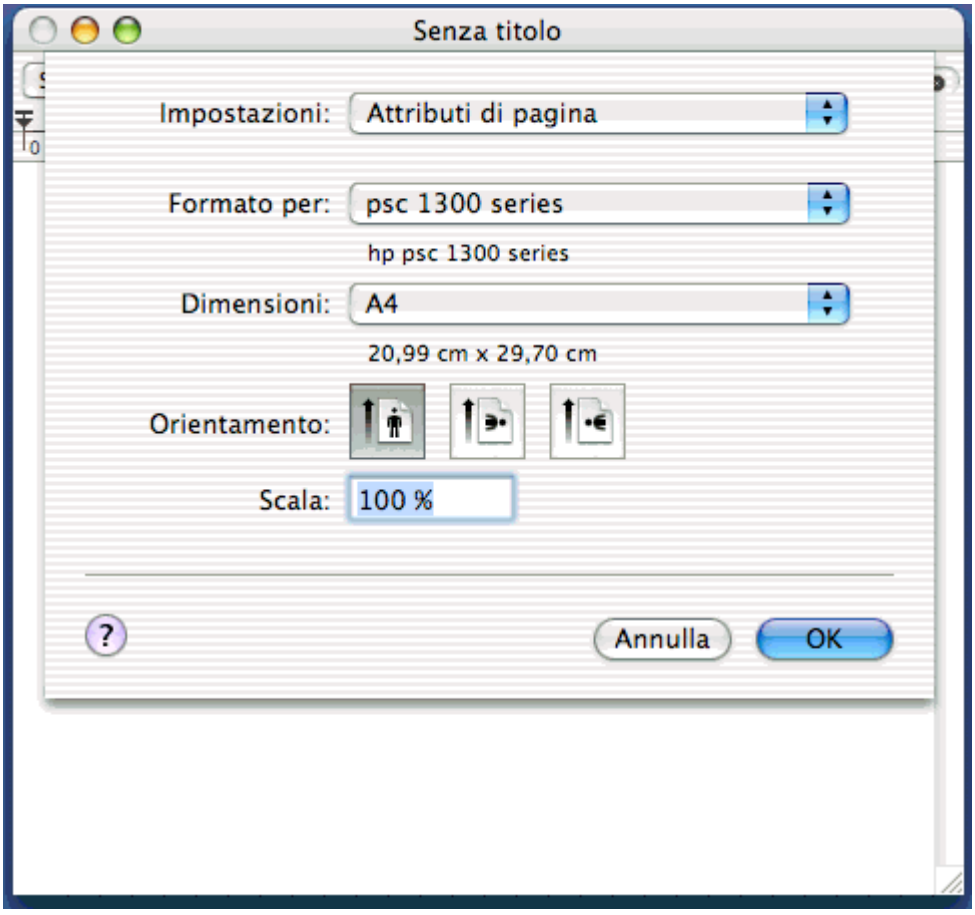
SOLO VOLUMI

L'applicazione si chiama CDCat perché, nell'idea iniziale, intende catalogare CD. In realtà, fino a questo momento, la finestra con la `NSOutlineView` è in grado di conservare informazioni su qualsiasi file e cartella.

Voglio limitare le possibilità di catalogazione permettendo di aggiungere ad un catalogo solamente volumi. Per volume intendo un disco rigido (o meglio, le partizioni di un disco rigido), dischi rimovibili come CD, DVD (nelle varie accezioni: scrivibili, riscrivibili...), dischi ZIP, eccetera. Presumo (ma non sono attualmente in grado di provarlo) che si possano anche catalogare volumi di rete, sia locale sia remota.

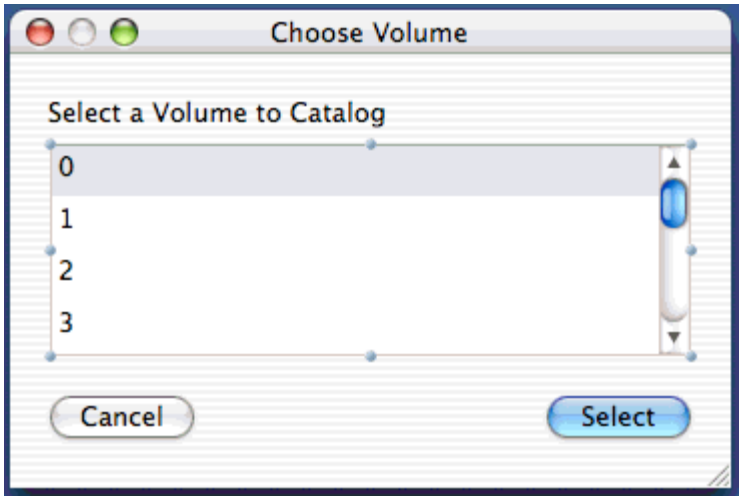
Ora, per aggiungere elementi ad un catalogo, sono possibili due strade: attraverso un dialogo standard di `apri file` oppure attraverso drag'n'drop. Vado per ordine e comincio col dialogo.

Ebbene, non riesco a trovare una caratterizzazione del dialogo standard di apertura file che limiti la scelta ai soli volumi. Anche se sono possibili diverse strade, decido di seguirne una tutta mia.



L'idea è di mostrare una finestra, all'interno della quale presentare, mediante una `NSTableView`, l'elenco dei volumi correntemente presenti. L'utente seleziona da questa lista un elemento, che sarà aggiunto al catalogo. Mi piacerebbe che tale finestra fosse in realtà una *sheet*, ovvero quel tipo di finestra che, in Mac OS X, risulta modale per un documento ma non per l'applicazione (per capire cosa intendo, prendete ad esempio il dialogo di stampa dell'applicazione `TextEdit`). Queste finestre appaiono simpaticamente scivolando da sotto il titolo della finestra, e lì rimangono attaccate anche muovendo la finestra. Nel frattempo, tuttavia, l'applicazione non è bloccata, e si può lavorare su altri documenti.

La Finestra Modale



Per realizzare una finestra modale destinata a diventare una sheet, non seguo particolari precauzioni. Vado in IB, costruisco un oggetto della classe `NSPanel`, lo riempio con del testo, una `NSTableView` e due pulsanti, uno di `Select` e uno di `Cancel`. Nella `NSTableView` mostro l'elenco dei volumi, con `Select` dico di catalogare il volume selezionato. `Cancel`, ovviamente, interrompe la sessione e fa nulla.

Definisco una nuova classe `ChooseVolCtrl`, sottoclasse di `NSWindowController`, e la attribuisco al `File's Owner`. Aggiungo un outlet verso la tabella e due azioni corrispondenti ai due pulsanti.

Ricordo che per la corretta visualizzazione di una tabella, questa ha bisogno di una classe che funzioni da sorgente di dati. Per non moltiplicare il numero delle classi, utilizzo lo stesso `File's Owner` come sorgente dei dati; questo significa che devo fare un collegamento tra la tabella ed il `File's Owner`, e che nella realizzazione della classe `ChooseVolCtrl`, oltre ai propri metodi, ci saranno anche dei metodi per la visualizzazione della tabella. Collego poi un altro po' di cose tra loro (la finestra al proprietario, i pulsanti alle azioni, cose del genere), faccio costruire i file della classe `ChooseVolCtrl`, e torno in XCode a scrivere i metodi.

Volumi ed Enumerator

Il primo problema da risolvere è trovare la lista dei volumi attualmente disponibili. Mi

viene in aiuto la classe `NSWorkspace`, già utilizzata per il recupero delle icone. Esiste un metodo adatto allo scopo denominato `mountedLocalVolumePaths:`. Il metodo restituisce un vettore con il percorso completo dei volumi presenti, cosa non particolarmente bella; infatti, lo hard disk del computer (meglio, la partizione principale dove si trova il sistema operativo) è rappresentato dal path '`/`', effettivamente poco significativo. Di più: un eventuale CD di nome '`pippo`' montato sulla scrivania appare nella lista come '`/Volumes/pippo`'. Per fortuna esiste il metodo `displayNameAtPath:` della classe `NSFileManager` che traduce questi path nel nomi cui siamo abituati.

Prima di passare al metodo che recupera i nomi dei volumi, stabilisco che la tabella della finestra presenterà anch'essa, come la `NSOutlineView` del documento catalogo, il nome del volume e l'icona che lo rappresenta. Mi occorre quindi anche salvare l'icona del volume. Questo spiega finalmente le variabili d'istanza della classe:

```
@interface ChooseVolCtrl : NSWindowController
{
    IBOutlet NSTableView *listaVolumi;
    // vettore con i nomi dei volumi
    NSMutableArray     * volumeNames;
    // vettore con i percorsi dei volumi
    NSMutableArray     * volumePaths;
    // vettore con le icone dei volumi
    NSMutableArray     * volumeIcons;
}
```

Ci sono tre vettori, che conservano i nomi per la visualizzazione, i percorsi completi per accederci quando si dovranno catalogare, e le icone per la visualizzazione.

Ho quindi scritto un metodo che esplora il mondo alla ricerca dei volumi presenti:

```
- (void)
checkLocalVolumes
{
    NSString    * volPath ;
    // recupero l'elenco dei volumi correntemente montati
        NSArray                *  tmp  =  [  [NSWorkspace  sharedWorkspace]
mountedLocalVolumePaths ] ;
    // ne faccio un enumerator
    NSEnumerator     * enumerator = [tmp objectEnumerator];
    // costruisco tre vettori per tenere nomi ed icone
    NSMutableArray    * tmpVolNames = [NSMutableArray array] ;
    NSMutableArray    * tmpVolPaths = [NSMutableArray array] ;
    NSMutableArray    * tmpVolIcons = [NSMutableArray array] ;

    // ciclo sui volumi correntemente montati
    while ((volPath = [enumerator nextObject]))
    {
        // recupero l'icona del volume
           NSImage           * tmpImg  =  [  [  NSWorkspace  sharedWorkspace]
iconForFile: volPath ] ;
        reduceImageToIcon( tmpImg ) ;        // la riduco ai minimi termini
           [ tmpVolIcons  addObject: tmpImg ];      // aggiungo l'icona  al
vettore
          [ tmpVolPaths  addObject: volPath ];     // aggiungo  il  path  al
vettore
        // aggiungo il nome del volume al vettore
```

```
            [ tmpVolNames addObject: [[ NSFileManager defaultManager]
displayNameAtPath: volPath] ];
    }
    // aggiorno i vettori d'istanza coi nomi ed icone
    [ self setVolumeNames: tmpVolNames ];
    [ self setVolumePaths: tmpVolPaths ];
    [ self setVolumeIcons: tmpVolIcons ];
}
```

Un costrutto nuovo (e la classe relativa) che compare qui è lo `enumerator`. Con un oggetto della classe `NSEnumerator` posso esplorare facilmente vettori, dizionari, insomma insiemi più o meno ordinati di oggetti. Si usa in modo molto semplice: dapprima si costruisce un enumerator relativo all'insieme che vi vuole esplorare. A questo enumerator poi, continuo a chiedere il prossimo oggetto dell'insieme, fino a che non restituisce `ni`". È proprio quello che ho fatto per estrarre i path dei volumi presenti, recuperare il nome e l'icona, ed inserire il tutto all'interno delle variabili d'istanza.

A questo punto, i due metodi principali per fornire i dati alla tabella si scrivono presto:

```
- (int)
numberOfRowsInTableView:    (NSTableView *)tableView
{
    return [volumeNames count];
}
```

Il numero di righe da visualizzare è evidentemente pari al numero di elementi di uno qualsiasi dei tre vettori; invece, l'oggetto alla riga `row` indicata è l'elemento `row`-esimo del vettore che mantiene i nomi dei volumi, ma tale valore è restituito solo dopo aver assegnato alla cella l'icona del volume stesso per la visualizzazione.

```
- (id)
tableView:            (NSTableView *)tableView
    objectValueForTableColumn:    (NSTableColumn *)tableColumn
    row:             (int)row
{
    // assegno alla cella l'immagine
      [((ImageAndTextCell*) [tableColumn dataCell]) setImage: [volumeIcons
objectAtIndex: row] ];
    // e poi restituisco il nome del volume
    return ( [volumeNames objectAtIndex: row] );
}
```

MONTAGGI E SMONTAGGI

Uno dei problemi della lista di volumi è che questa lista non è costante nel tempo; in altri termini, la lista cambia se si rende disponibile un nuovo CD, o viene espulso un altro CD, o se in rete si perde una connessione o se acquista una nuova. Si tratta di mantenere aggiornata la lista dei volumi, fotografata ad un certo istante, con la situazione corrente. Ancora una volta mi viene in aiuto la classe `NSWorkspace`, che rende disponibili delle notifiche quando un volume è **montato** (annuncia la sua disponibilità) o **smontato** (non è più presente perché ad esempio espulso). Occorre sottoscrivere un paio di abbonamenti; lo faccio nel metodo `windowDidLoad:`, assieme ad un altro paio di cose:

```
- (void)
windowDidLoad
{
    NSTableColumn *tableColumn = nil;
    ImageAndTextCell *imageAndTextCell = nil;

    // metto a posto l'elenco dei volumi presenti
    [self checkLocalVolumes ];
    // attenzione a quale notification center attaccarsi!!
    // dopo di che, dico che mi segnalino operazioni coi volumi
    [[ [NSWorkspace sharedWorkspace] notificationCenter] addObserver: self
        selector: @selector( situationChanged: )
        name: NSWorkspaceDidMountNotification object: nil ] ;
    [[ [NSWorkspace sharedWorkspace] notificationCenter] addObserver: self
        selector: @selector( situationChanged: )
        name: NSWorkspaceDidUnmountNotification object: nil ] ;
    // l'unica colonna ha come cella una ImageAndTextCell
    tableColumn = [listaVolumi tableColumnWithIdentifier: @"listaVol"];
    // creo l'oggetto autorelease perche' passa in carico alla tableColumn
    imageAndTextCell = [[[ImageAndTextCell alloc] init] autorelease];
    [imageAndTextCell setEditable: NO];
    [tableColumn setDataCell:imageAndTextCell];
    [ listaVolumi reloadData ];
}
```

Al caricamento della finestra, faccio la fotografia dello stato attuale dei volumi presenti con il metodo `checkLocalVolumes`. Poi, sottoscrivo due abbonamenti alle notifiche `NSWorkspaceDidMountNotification` e `NSWorkspaceDidUnmountNotification`. Da notare il fornitore di questi abbonamenti, che non è il nostro solito, ma quello specifico della classe `NSWorkspace` (ci ho messo un bel po' prima di capirlo; poi, è bastato, come sempre, leggere meglio la documentazione). Quando è disponibile un nuovo volume, o uno che era presente non è più in linea, è invocato il metodo `situationChanged:`, all'interno del quale aggiornerò la situazione. Il metodo poi prosegue installando la cella per la visualizzazione di testo ed immagine, e forzando l'aggiornamento della tabella per mostrare l'elenco dei volumi appena rilevato.

Ecco allora il metodo `selectionChanged:`, invocato quando succede qualcosa ai volumi presenti:

```
- (void )
situationChanged: ( NSNotification *) notification
{
    // in realta', nella notifica, c'e' scritto cosa e' successo...
    // ma e' per me piu' comodo aggiornare da zero la lista dei volumi
    [ self checkLocalVolumes ];
    // dico che la lista deve essere rinfrescata
    [ listaVolumi reloadData ];
}
```

Questo metodo, molto semplicemente, si limita ad esplorare nuovamente la situazione e a rinfrescare la tabella.

All'interno di questa classe ci sono altri due metodi, ma li commento dopo aver modificato la classe `CatalogDoc`.

AGGIUNGERE VOLUMI

Una volta predisposta la classe `ChooseVolCtrl` occorre riscrivere il metodo `addFileItem:` della classe `CatalogDoc` per tenere conto di queste nuove esigenze. Questo metodo adesso apre la finestra come una sheet e poi gestisce le operazioni di catalogazione una volta che l'utente ha selezionato qualcosa (ma anche no). Il metodo è molto semplice, solo perché il grosso del lavoro è svolto altrove.

```
- (void)
addFileItem: (id)sender
{
    // se devo aggiungere solamente volumi, costruisco una finestra
    // in cui è presente la lista dei volumi; sara' una sheet
    // costruisco la finestra
    ChooseVolCtrl *tmpCVCtrl = [[ ChooseVolCtrl alloc] init ] ;
    // comincio la procedura: mi metto modale per il documento
    // alla chisura della sheet, sara' invocato il metodo addVolume:...
    [ NSApp beginSheet: [ tmpCVCtrl window ]
        modalForWindow: [ NSApp mainWindow ]
        modalDelegate: self
        didEndSelector: @selector( addVolume2Cat:returnCode:contextInfo: )
        contextInfo: nil
    ];
}
```

Si costruisce la finestra, e poi si lancia una sheet. Questa sheet è una finestra che scivola da sotto il titolo di una finestra, ed impedisce ogni lavorio sulla finestra oscurata (non impedisce però di lavorare su altri documenti all'interno dell'applicazione); la sheet non ha titolo, e rimane appiccicata alla finestra comunque questa si muova.

La costruzione di una sheet è un messaggio inviato all'applicazione nel suo complesso (la cosa mi sembra ragionevole, dal momento che bisogna aggiustare il gestore degli eventi): deve sapere quale finestra è la sheet vera e propria (primo argomento) e quale finestra verrà bloccata da questa sheet (secondo argomento). Il controller della sheet (nel caso, `ChooseVolCtrl`) gestisce l'interazione, e poi, quando l'utente ha risposto ai quesiti posti (nel mio caso, ha selezionato uno dei volumi e ha fatto clic su `Select` o su `Cancel`) restituisce il controllo alla finestra. Se nel metodo `beginSheet:...` è specificato il terzo argomento `didEndSelector:` (come nel mio caso), allora è invocato proprio il metodo specificato da questo argomento. Il metodo, dal nome piuttosto lungo, si aspetta tre parametri: il primo è l'oggetto sheet che è appena terminato; il secondo è un valore di ritorno fornito dalla sheet stessa (ci arrivo in un momento), il terzo sono informazioni che il metodo chiamante vuole passare a questo metodo chiamato, inserite proprio nell'ultimo argomento (nel mio caso, non ho trovato nulla da interessante da dire).

Noto che il metodo `beginSheet:...` non blocca le operazioni del metodo (in altre parole, se dopo l'istruzione con questo metodo ci fosse qualche altra istruzione, questa sarebbe eseguita); tuttavia, poiché la sheet intercetta ogni evento destinato alla finestra, non è che si possa fare molto (se non lavori in background, ma questo è un altro discorso).

In pratica, dopo quest'istruzione il controllo delle operazioni passa alla sheet; all'interno della sheet l'utente seleziona un volume, poi fa clic su `Select` o su `Cancel`. I metodi corrispondenti, della classe `ChooseVolCtrl` sono allora i seguenti:

```
- (IBAction)
selectButton:    (id)sender
{
    // annullo l'abbonamento
      [[ [NSWorkspace sharedWorkspace] notificationCenter] removeObserver:
self ];
    // butto via la finestra
    [[self window] setIsVisible: FALSE ];
    // ho finito di lavorare modale con la sheet
    // il codice di ritorno e' la riga selezionata
     [NSApp endSheet: [self window] returnCode: [ listaVolumi selectedRow ]
];
}
```

Il metodo `cancelButton:` è del tutto simile; l'unica differenza è l'argomento `returnCode`, pari a -1. L'idea infatti è che la sheet comunichi alla finestra la riga selezionata dall'utente come l'argomento `returnCode`, che corrisponde ad un elemento nel vettore variabile d'istanza. La finestra di catalogo preleva poi l'elemento, che è il percorso del volume, e procede alla catalogazione: ecco quindi il metodo `addVolume2Cat:...`, ancora una volta molto semplice perché il grosso del lavoro è svolto altrove:

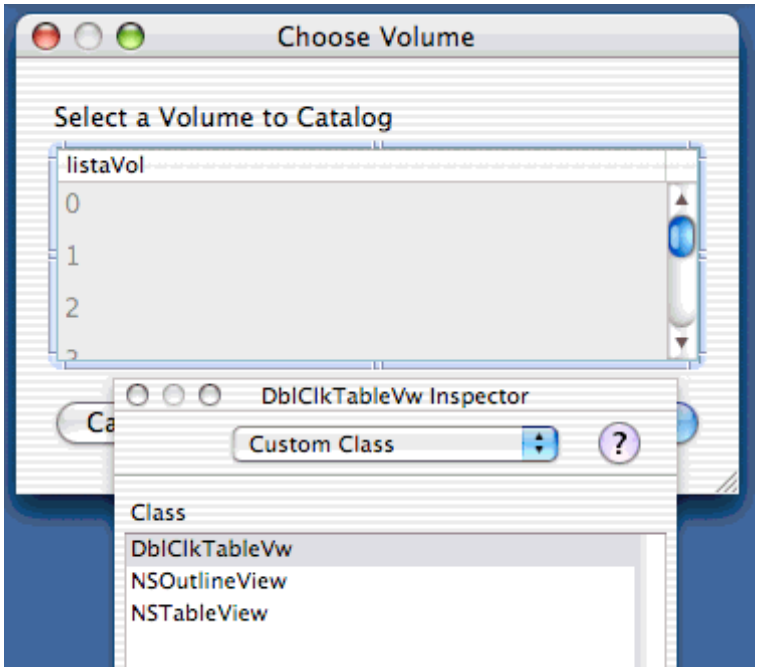
```
- (void)
addVolume2Cat:         (NSWindow *)sheet
    returnCode:    (int)returnCode
    contextInfo:    (void *)contextInfo
{
    NSString    * volName ;

    // se il codice di ritorno e' -1, non ho nulla da fare
    if ( returnCode== -1 )
        return ;
    // se arrivo qui, ho da montare un volume
    // recupero il percorso completo del volume...
      // dalla finestra recupero il controllore, da qui il vettore dei
percorsi, e dal vettore
    // piglio l'elemento di indice returnCode
     volName = [[((ChooseVolCtrl*) [sheet windowController]) volumePaths]
objectAtIndex: returnCode ];
    [ self performAddFilesModal: [ NSArray arrayWithObject: volName]];
    // ovviamente, il documento e' stato sporcato
    [ self updateChangeCount: NSChangeDone ] ;
    // rinfresco la finestra con i nuovi dati
    [ outlineView reloadData ];
}
```

Il metodo sfrutta il primo argomento, la finestra della sheet, per accedere alle variabili d'istanza del controllore di questa finestra; in questo modo recupera l'elemento d'indice indicato da `returnCode`, vale a dire, il percorso del volume da catalogare. Con questo path, costruisce un vettore di un solo elemento che passa al metodo `performAddFilesModal:`, per poi segnalare che il documento si è sporcato e che è il caso di aggiornare il contenuto della finestra.

Ma prima di passare avanti, un momento di pausa con un giochetto interessante.

Doppio Clic e Select

Lavorando con la lista dei volumi, ho trovato noioso selezionare il volume e poi fare clic su `Select`. Mi piacerebbe fare doppio clic sul volume, e lanciare in automatico la catalogazione. In altre parole, mi piacerebbe che la tabella risponda ad un doppio clic su di un elemento come il pulsante `Select`. Per fare questo, ho fatto una cosa semplice (ma non so se efficiente e soprattutto, non so se esiste un meccanismo equivalente già all'interno di Cocoa; non importa, nel farlo ho imparato qualcosa): ho creato una sottoclasse di `NSTableView` e ne ho riscritto un metodo per cambiarne il comportamento.



Torno in IB, e faccio una sottoclasse di `NSTableView` chiamata `DblClkTableVw`. Faccio in modo la tabella diventi di questa classe (si fa nella finestra Info, pannello `Custom class`). Genero i file della classe e torno in XCode.

Qui riscrivo il metodo `mouseDown:` (che in realtà è proprio di `NSControl` di cui la `NSTableView` è comunque una sottoclasse), che è inviato quando si fa clic col mouse sulla tabella:

```
- (void)
mouseDown:    (NSEvent *)theEvent
{
    // processo normalmente il comando
    [ super mouseDown: theEvent] ;
    // poi conto quanti clic ci sono stati
    // se sono due (o piu'), dico che c'e' stato un doppio-clic
    if ( [theEvent clickCount] >= 2 )
    {
        // che e' come aver scelto il pulsante Select
        // per individuare il destinatario, uso un trucco...
        [ self sendAction: @selector( selectButton: ) to: [self dataSource]
] ;
    }
}
```

In primo luogo, faccio in modo che l'evento sia trattato normalmente; poi, aggiungo il

mio codice specifico: conto il numero di clic ravvicinati del mouse. Se due o più, mando il messaggio `selectButton:` al controllore della finestra. Per individuare tale controllore, uso la strada più breve a disposizione, ovvero sfrutto il fatto che il controllore della finestra è anche il delegato della tabella.

NUOVO DROP

Ho estratto il codice dell'inserimento del volume nel catalogo perché utilizzo questo metodo anche all'interno del metodo che realizza le operazioni di drag'n'drop.

C'è una fondamentale modifica da fare: le operazioni di drop si accettano solamente se gli elementi droppati sono dei volumi. Ho concentrato le modifiche all'interno del metodo `prepareForDragOperation:`. Ricordo che questo metodo è invocato quando l'utente rilascia il mouse dopo la draggatura e poco prima della droppatura. Il metodo risponde `YES` se l'operazione si può fare (quindi, se tutti gli elementi draggati sono path corrispondenti a volumi) e `NO` altrimenti (nella vecchia versione rispondeva sempre `YES`, dal momento che accettava qualunque path).

```
- (BOOL)
prepareForDragOperation:(id<nsdragginginfo>)sender
{
    // questo e' il momento di decidere se fare il drop
    NSPasteboard *pboard;

    pboard = [sender draggingPasteboard];
    // in primo luogo, droppo solo path di file
    if ([[pboard types] indexOfObject:NSFilenamesPboardType] != NSNotFound)
    {
        // recupero la lista degli elementi draggati
             NSArray             * pList = [ pboard propertyListForType:
NSFilenamesPboardType ] ;
        // la metto in un enumerator
        NSEnumerator     * enumerator = [pList objectEnumerator];
        // recupero anche la lista dei volumi correntemente mondati
           NSArray           * volumes = [ [NSWorkspace sharedWorkspace]
mountedLocalVolumePaths ] ;
        NSString    * volPath ;
        // dico che accetto il drop solo di volumi
        while ((volPath = [enumerator nextObject]))
        {
            // se cerco di droppare un elemento che non e' un volume
            if ( ! [ volumes containsObject: volPath ] )
                return ( NO );    // il drop non si fa
        }
        // se arrivo qui, tutti gli elementi droppati sono volumi,
        // posso eseguire l'operazione
        return ( YES );
    }
    // se arrivo qui, gli elementi droppati non sono nemmeno path...
    return ( NO );
}
```

La cattiva programmazione mi ha fatto scrivere il metodo qui sopra, che esegue le operazioni in maniera piuttosto pedestre: in pratica verifica che ogni elemento presente nella lista degli elementi draggati si trovi anche all'interno della lista dei volumi correnti. Non appena un elemento non risponde a questo criterio, risponde `NO` e

l'operazione di drop è cancellata. Forse c'è un meccanismo migliore e più efficiente, ma non ho voglia di cercarlo.

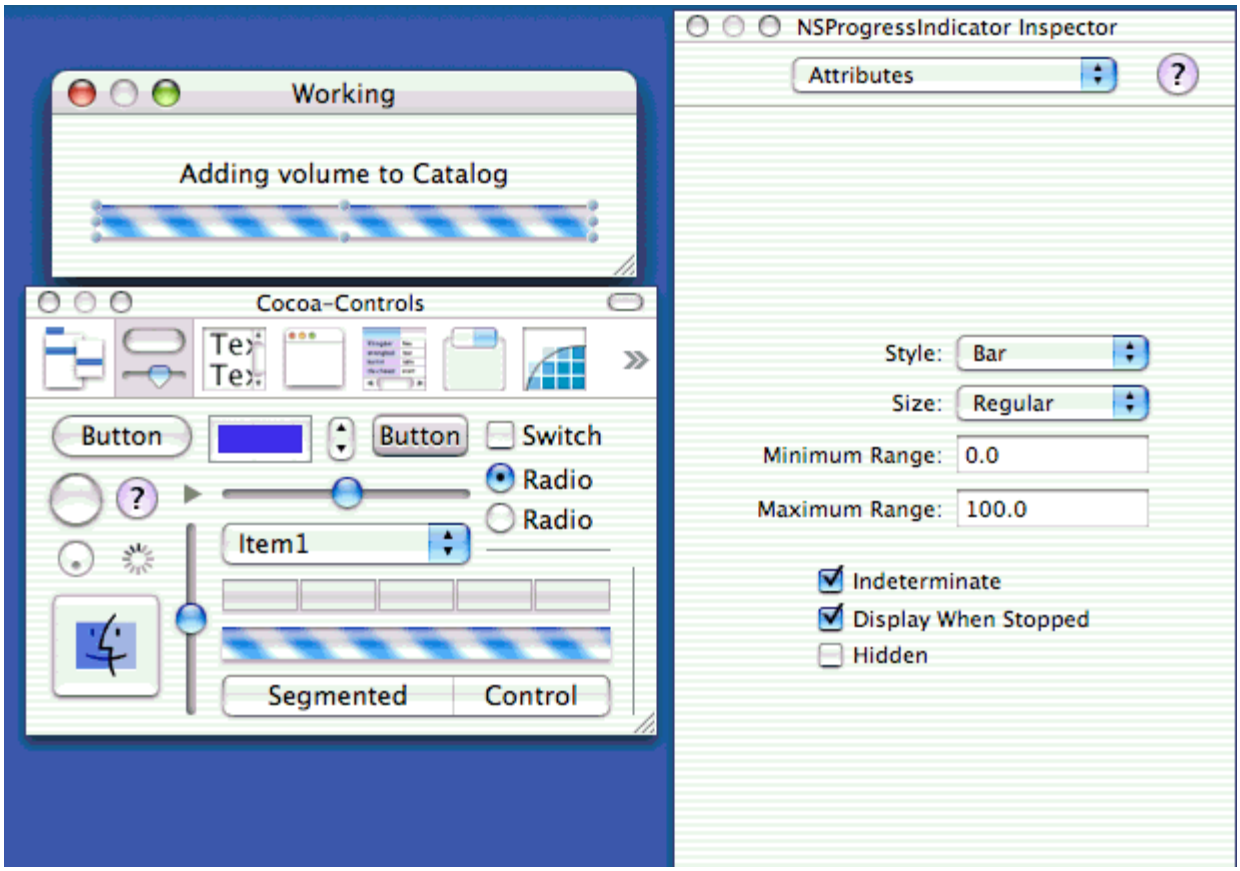
Se il metodo precedente risponde `YES`, si può procedere alla droppatura, e quindi alla catalogazione degli elementi droppati: questo è un compito per...

```
- (BOOL)
performDragOperation:(id<nsdragginginfo>)sender
{
    NSPasteboard     * dragPasteboard;
    NSArray        * pList ;

    // recupero la pasteboard
    dragPasteboard = [sender draggingPasteboard];
    // da qui, la lista degli elementi draggati
    pList = [ dragPasteboard propertyListForType: NSFilenamesPboardType ] ;
    [ self performAddFilesModal: pList ];
    // ovviamente, il documento e' stato sporcato
    [ self updateChangeCount: NSChangeDone ] ;
    return ( YES) ;
}
```

che diventa molto semplice: si piglia la lista degli elementi droppati e la si passa al metodo `performAddFilesModal:`.

La Barra del Barbiere

Sarebbe giunto il momento di vedere il metodo `performAddFilesModal:`, se non fosse che all'interno di questo metodo c'è un altro concetto.



Visto che l'operazione di catalogazione è piuttosto lunga, è grazioso informare l'utente dell'attesa mostrandogli una finestra di cortesia. La cosa più bella sarebbe mostrargli una barra che si colora con l'avanzamento delle operazioni (come molti programmi di installazione ci hanno abituato); tuttavia, nella catalogazione al momento ignoro lo stato di avanzamento. Sfrutto allora una barra tipo barbiere, ovvero una barra con righe animate bianche ed azzurre. Vado ancora una volta in IB e costruisco una finestra, con un testo e un oggetto della classe `NSProgressIndicator` (la citata barra del barbiere). La chiamo `WaitingPanel`, faccio come al solito un controllore `WaitPanCtrl`, outlet e collegamenti vari (vado via veloce, ormai ne ho fatte a bizzeffe di queste finestre).

L'interfaccia verso il mondo di questa finestra è composta da tre metodi (fintamente) di classe: come al solito, ci sarà una sola istanza di questa classe, la finestra riutilizzata da chiunque ne avesse bisogno.

```
+ (void) setText: (NSString*) title ;
+ (void) animate: (id) sender ;
+ (void) showHide: (BOOL) state ;
```

Con il primo metodo imposto la stringa di testo, con il secondo faccio animare la barra, con il terzo controllo la visibilità della finestra.

```
+ (void) setText: (NSString*) title
{
    // chiamo un metodo interno
    [ [ WaitPanCtrl sharedProgress] setMainText: title ];
}

+ (void) animate: (id) sender
{
    // chiamo un metodo interno
    [ [ WaitPanCtrl sharedProgress] animation: sender ];
}

+ (void) showHide: (BOOL) state
{
    // mostro/nascondo direttamente la fienstra
    [[[ WaitPanCtrl sharedProgress] window] setIsVisible: state ];
}
```

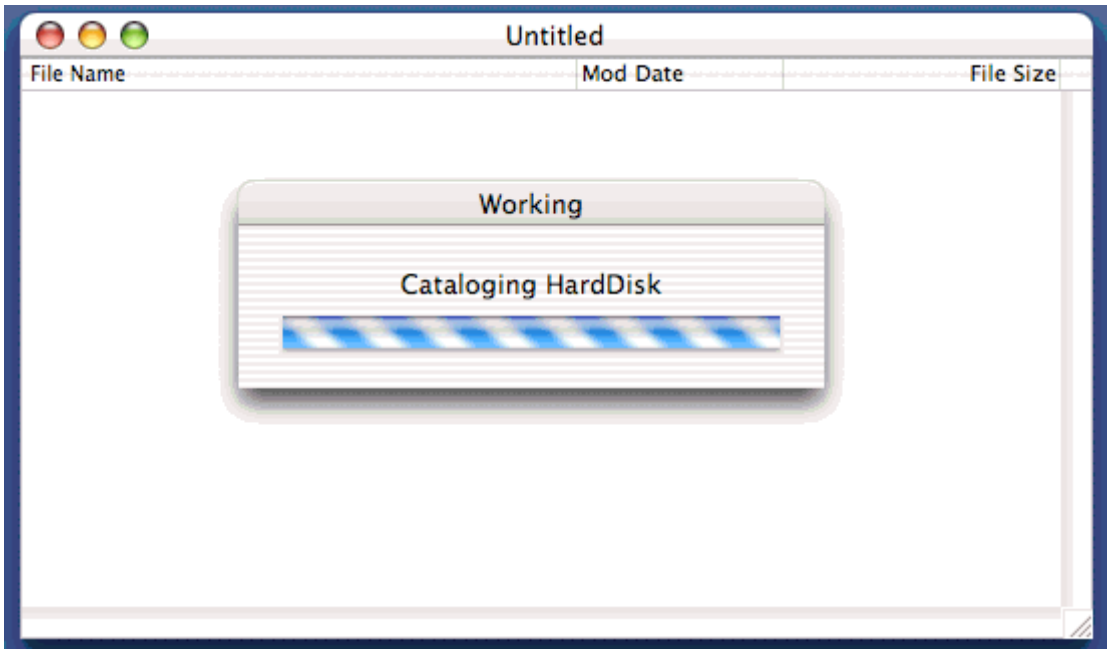
I primi due metodi si appoggiano sui metodi seguenti (non posso raggiungere gli outlet dentro i metodi di classe...):

```
- (void )
setMainText: (NSString*) title
{
    // banale, imposto la stringa
    [ infoText setStringValue: title ] ;
    // forzo il rinfresco della finestra
    [ infoText displayIfNeeded ];
}

- (void)
animation: (id) sender
{
```

```
    // passo di animazione
    [ progressBar animate: sender ] ;
    // forzo il rinfresco della finestra
    [ progressBar displayIfNeeded ];
}
```

È fondamentale il messaggio `displayIfNeeded`, perché normalmente la finestra è rinfrescata ad ogni loop degli eventi, e qui (ho scoperto con fatica) il loop non gira, visto che la barra impedirà la ricezione degli eventi stessi.

AGGIUNGI UN FILE



Finalmente ci siamo; ecco il metodo per aggiungere un elenco di file ad un catalogo. L'idea è di mostrare la finestra di attesa e di bloccare l'intera applicazione fino a che l'operazione non è completata (lo so che non è una bella cosa, ma per il momento si fa così).

```
- (void)
performAddFilesModal: ( NSArray *)volList
{
    NSModalSession     session ;
    NSEnumerator     * enumerator = [volList objectEnumerator];
    NSString    * volPath ;
```

Esamino i path attraverso un enumerator. Estraggo subito il primo path per predisporre la finestra con il testo e per cominciare l'animazione della barra da barbiere:

```
    volPath = [enumerator nextObject] ;
    // mostro la finestra di attesa con apposita stringa
    [ WaitPanCtrl showHide: TRUE ];
    [ WaitPanCtrl setText: [NSString stringWithFormat: @"Cataloging %@",
        [[ NSFileManager defaultManager] displayNameAtPath: volPath] ]];
    // faccio partire l'animazione della barra del barbiere
```

```
    [ WaitPanCtrl animate: self ];
```

Uso i metodi di classe come se fossero variabili globali per accedere alla finestra dell'attesa. Da notare come il testo usa il nome del volume e non il percorso.

```
    // comincio una sessione modale per l'intera applicazione
        session = [NSApp beginModalSessionForWindow: [ [ WaitPanCtrl
sharedProgress] window ] ];
    // eseguo la sessione
    [NSApp runModalSession:session] ;
```

Con queste due istruzioni comincio ed eseguo una sessione modale per l'applicazione, ovvero, tutte le finestre dell'applicazione non ricevono eventi fino a che questa sessione non termina. Tuttavia, l'attività del metodo prosegue (sono gli eventi che sono bloccati, non le operazioni), ed anzi, sfrutto proprio questa caratteristica per procedere con le operazioni di catalogazione

```
    while ( volPath )
    {
        // costruisco l'alberatura dei file a partire dalla scelta
         FileStruct      * fInfo = [[ FileStruct alloc ] initTreeFromPath:
volPath ];
        // aggiungo la cosa al catalogo
        [ dataSource addFileEntry: fInfo ];
        volPath = [enumerator nextObject] ;
        if ( volPath == nil ) break ;
```

Qui ho recuperato le informazioni sul file corrente (e, per la ricorsività della faccenda, di tutto il volume); passo poi al volume successivo, a meno che lo enumerator restituisca nil. In questo caso, ho finito di esaminare i volumi, ed esco dal `while`.

```
        // mostro la finestra di attesa con apposita stringa
           [ WaitPanCtrl setText: [NSString stringWithFormat: @"Cataloging
%@",
               [[ NSFileManager defaultManager] displayNameAtPath: volPath]
]];
        // faccio partire l'animazione della barra del barbiere
        [ WaitPanCtrl animate: self ];
```

Essendo cambiato il file sotto esame, cambio la scritta sulla finestra, e già che ci sono faccio un giro di barbiere.

```
    }
    // finisco la sessione
    [NSApp endModalSession: session];
    // nascondo la finestra di attesa
    [ WaitPanCtrl showHide: FALSE ];
}
```

Al termine, dichiaro chiusa la sessione modale e nascondo la finestra. Adesso le finestre riprendono a ricevere eventi.

Per rendere più amena l'animazione, all'interno del metodo `initWithPath:` della classe `LSFileInfo` ho aggiungo l'istruzione

```
[ WaitPanCtrl animate: self ];
```

per fare in modo che la barra si muova un po'.

Questo lungo capitolo termina qui.

Capitolo 028 - Barra degli Strumenti

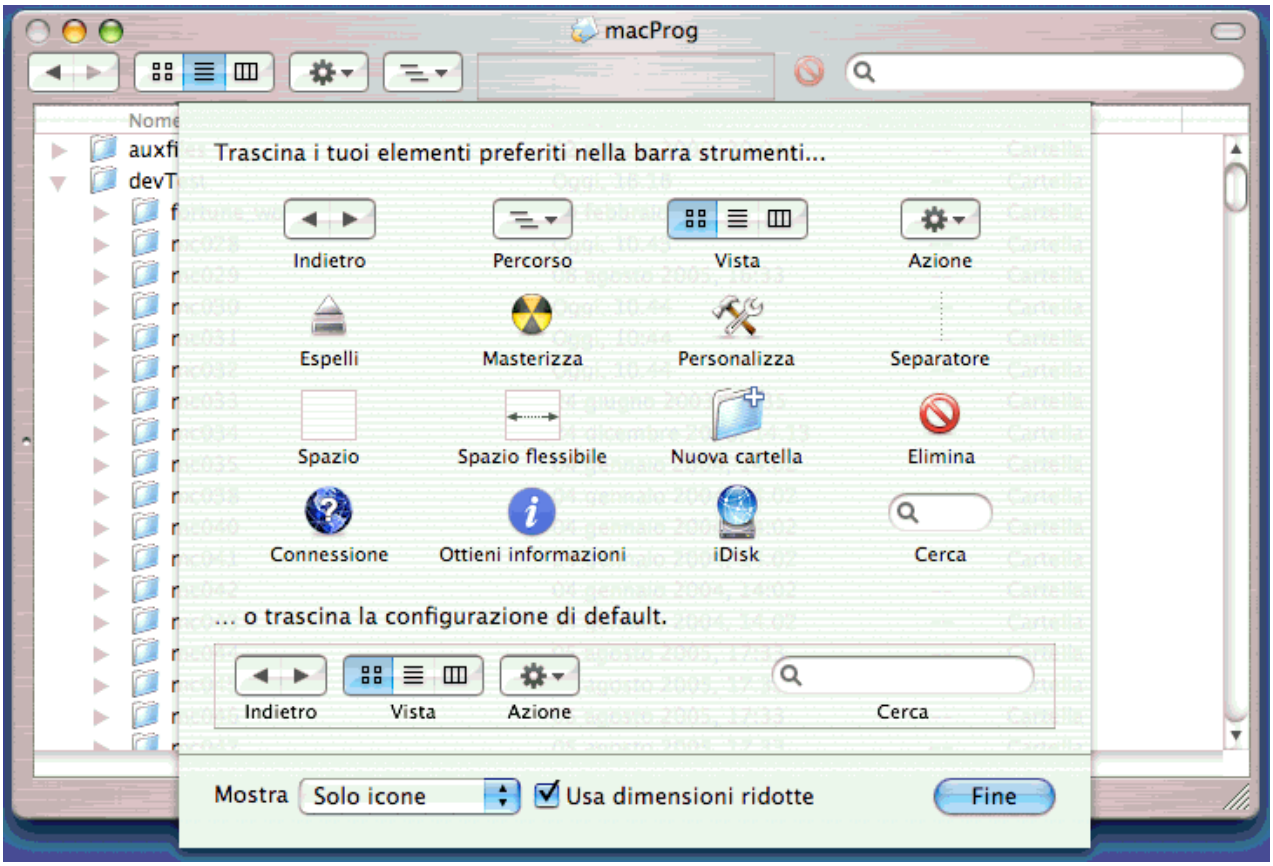
In questo capitolo intendo inserire una toolbar alla finestra di catalogo, duplicando (quasi) tutte le funzioni presenti nella palette dei comandi. La cosa si rivela molto semplice, vuoi per sé, vuoi perché un esempio di Apple fornisce praticamente tutto ciò che mi serve.

Sorgenti: Un esempio di Apple (`/Developer/Examples/AppKit/SimpleToolbar`).

Primo inserimento: 16 settembre 2002

TOOLBAR

Anche al più casuale utente di Mac OS X non può sfuggire la *toolbar* (Barra degli Strumenti) presente nelle finestre del Finder.



Questa toolbar raccoglie alcuni comandi di uso comune, ed ha l'interessante caratteristica di essere personalizzabile dall'utente, aggiungendo comandi propri col drag'n'drop oppure tramite un apposito dialogo.

Pensavo che l'aggiunta di una tale caratteristica alle mie finestre di catalogo fosse un'impresa improba; invece, si è rilevata molto semplice. Non solo: a puro titolo gratuito, Cocoa fornisce anche un menu ed un dialogo di personalizzazione della toolbar

stessa. Anche se il lavoro mi è stato facilitato da un esempio di Apple a corredo dei Developer's Tools, la faccenda è veramente semplice.

Una toolbar normalmente contiene semplici elementi cliccabili che funzionano come dei pulsanti. L'elemento classico di una toolbar è quindi definito da una icona (`image`), un nome corto per mostrarlo nella toolbar (`label`), un nome più lungo per mostrarlo nella finestra di personalizzazione (`paletteLabel`), una descrizione del suo funzionamento mostrata come testo di aiuto (`tooltip`), l'operazione scatenata da un clic (`target` e `action`). Normalmente, una immagine (`tiff` o `pict`) è quanto basta per un elemento della toolbar, ma è possibile definire elementi più complicati, come menu pop-up, immagini speciali, eccetera (in cui tuttavia non mi avventuro).

Prevedo che la mia toolbar avrà elementi per svolgere le seguenti funzioni: aggiungere un volume al catalogo; cancellare un elemento dal catalogo; salvare il catalogo su disco; stampare la finestra del catalogo.

IL DELEGATO ALLA TOOLBAR

Per la realizzazione della toolbar, il concetto base di tutto è ancora una volta la delega. Occorre individuare una classe delegata per la toolbar, che si preoccupi di svolgere i compiti di coordinamento e controllo della toolbar stessa. Poiché la toolbar spesso contiene comandi relativi ad un documento, la classe che gestisce il documento è la naturale candidata al meccanismo di delega.

Dire che la classe `CatalogDoc` è la delegata della toolbar significa che bisogna realizzare un gruppo di metodi ben precisi: li si trova nella documentazione della classe `NSToolbar`, che appunto descrive come si realizza una toolbar. Nella documentazione esiste anche un `Programming Topic` che inquadra meglio l'intero problema.

Benché una toolbar sia un elemento visuale, di interfaccia utente, non si può costruire una toolbar tramite IB, ma bisogna farlo da programma, direttamente in XCode.

Il primo metodo (obbligatorio) realizzato dalla classe delegata fornisce infatti l'elenco degli elementi di default presenti all'interno della toolbar. Questa lista di elementi è utilizzata alla prima creazione della toolbar e per presentare la toolbar di default all'interno del dialogo di personalizzazione.

Un secondo metodo, pur'esso obbligatorio, deve invece fornire la lista di tutti gli elementi che in qualche maniera possono comparire all'interno della toolbar. Questa lista è utilizzata dal dialogo di personalizzazione per consentire la scelta degli elementi da parte dell'utente.

Il terzo ed ultimo metodo obbligatorio, finalmente, deve fornire l'elemento della toolbar.

Per indicare gli elementi di una toolbar, si utilizzano degli identificatori; la toolbar stessa ha un identificatore, che permette di utilizzare la stessa toolbar in posti differenti, mantenendo ovunque lo stesso aspetto.

Alcuni identificatori di toolbar sono predefiniti: il separatore, lo spazio vuoto di

dimensione fissa; lo spazio vuoto di dimensione variabile; elementi per la scelta di caratteri e colori; l'elemento per la stampa; l'elemento per la configurazione della toolbar. Tutti gli altri elementi devono essere definiti dal programmatore.

Io prevedo tre elementi aggiuntivi:

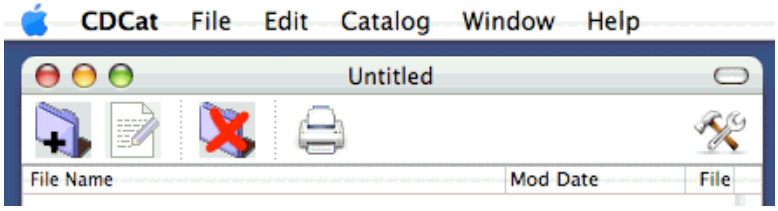
```
#define         TBItId_AddItem          @"tbItem_AddItem"
#define         TBItId_DelItem          @"tbItem_DelItem"
#define         TBItId_SaveItem          @"tbItem_SaveItem"
```

l'elemento per aggiungere un volume al catalogo, un elemento per cancellare una voce di catalogo, l'elemento per salvare il documento corrente.

Detto questo, mi viene facile scrivere i primi due metodi delegati.

```
// questo metodo dice quali sono gli item di default della toolbar
    // sono gli item presenti alla creazione e quando l'utente ripristina
- (NSArray *)
toolbarDefaultItemIdentifiers: (NSToolbar *) toolbar
{
    // la lista e' l'elenco ordinato degli identificatori degli elementi
    return [NSArray arrayWithObjects:
        TBItId_AddItem,             // aggiunta elemento
        TBItId_SaveItem,             // salvataggio catalogo
        NSToolbarSeparatorItemIdentifier,     // separatore
        TBItId_DelItem,                 // cancellazione
        NSToolbarSeparatorItemIdentifier,     // separatore
        NSToolbarPrintItemIdentifier,         // stampa catalogo
        NSToolbarFlexibleSpaceItemIdentifier,     // spazio flessibile
        NSToolbarCustomizeToolbarItemIdentifier,// personalizzazione
        nil];
}

// metodo delegato obbligatorio
// questo metodo dice quali sono gli item possibili della toolbar
    // sono TUTTI gli item che possono entrare nella toolbar, e quindi
    // mostrati nel dialogo di personalizzazione
- (NSArray *)
toolbarAllowedItemIdentifiers: (NSToolbar *) toolbar
{
    // normalmente, nessun elemento e' previsto presente, quindi bisogna
    // inserire tutti gli elementi che si pensa possano essere presenti;
    // in particolare, separatori e spazi vari...
    return [NSArray arrayWithObjects:
        TBItId_AddItem, TBItId_SaveItem, TBItId_DelItem,    // i miei tre
        // e poi tutti gli altri ragionevoli...
        NSToolbarPrintItemIdentifier, NSToolbarShowFontsItemIdentifier,
        NSToolbarCustomizeToolbarItemIdentifier, NSToolbarFlexibleSpaceItemIdentifier,
        NSToolbarSpaceItemIdentifier, NSToolbarSeparatorItemIdentifier, nil];
}
```

Sono piuttosto semplici: si limitano a costruire un elenco degli elementi e di inserirlo all'interno di un array. È da notare l'ordine degli elementi all'interno del metodo `toolbarDefaultItemIdentifiers:`, dal momento che sarà proprio l'ordine con cui gli elementi saranno poi visualizzati (da sinistra verso destra) all'interno della toolbar.

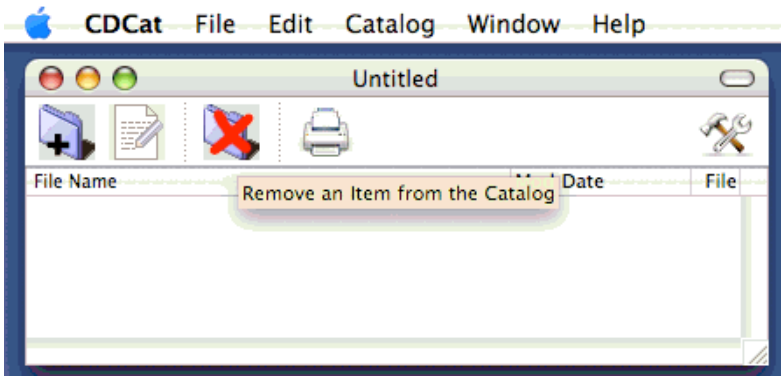


Per completezza, ho inserito nella toolbar anche l'elemento che comanda la personalizzazione della toolbar. L'uso dello spazio di dimensione variabile permette di posizionare questo elemento tutto a destra, e di mantenere tale posizione anche ridimensionando la finestra.

Il terzo metodo delegato è lungo ma piuttosto noioso. Ripeto per tre volta la stessa sequenza di operazioni, una per ogni elemento non predefinito della toolbar.

```
- (NSToolbarItem *)
toolbar: (NSToolbar *)toolbar
    itemForItemIdentifier: (NSString *) itemIdent
    willBeInsertedIntoToolbar:(BOOL) willBeInserted
{
    // costruisco una istanza di item della toolbar
    // come al solito, e' autorelase, verra' ritenuto dalla toolbar
    NSToolbarItem *toolbarItem = [[[NSToolbarItem alloc] initWithItemIdentifier: itemIdent]
autorelease];

    // item della toolbar per aggiungere un volume
    if ([itemIdent isEqual: TBItId_AddItem])
    {
        // do due nomi all'icona di aggiunta volume: nome per la toolbar
        [toolbarItem setLabel: @"Add"];
        // nome per il dialogo di personalizzazione
        [toolbarItem setPaletteLabel: @"Add Volume"];
        // predispongo la scritta di aiuto (il tooltip)
        [toolbarItem setToolTip: @"Add a Volume to the Catalog"];
        // predispongo l'icona di questo item
        [toolbarItem setImage: [NSImage imageNamed: @"Add"]];
        // quando l'item e' cliccato, deve mandare al documento (self)...
        [toolbarItem setTarget: self];
        // ...il messaggio opportuno
        [toolbarItem setAction: @selector(addFileItem:)];
        return toolbarItem;
    }
    // ripeto il giochetto per la cancellazione di un elemento di catalogo
    if ([itemIdent isEqual: TBItId_DelItem])
    {
        [toolbarItem setLabel: @"Delete"];
        [toolbarItem setPaletteLabel: @"Delete Item"];
        [toolbarItem setToolTip: @"Remove an Item from the Catalog"];
        [toolbarItem setImage: [NSImage imageNamed: @"del"]];
        [toolbarItem setTarget: self];
        [toolbarItem setAction: @selector(delItem:)];
        return toolbarItem;
    }
    // e per il salvataggio del catalogo
    if ([itemIdent isEqual: TBItId_SaveItem])
    {
        [toolbarItem setLabel: @"Save"];
        [toolbarItem setPaletteLabel: @"Save Item"];
        [toolbarItem setToolTip: @"Save the Catalog"];
        [toolbarItem setImage: [NSImage imageNamed: @"save"]];
        [toolbarItem setTarget: self];
        // chiamo un metodo di default
        [toolbarItem setAction: @selector(saveDocument:)];
        return toolbarItem;
    }
    // gli altri dovrebbero essere standard
```

```
    return ( nil );
}
```



Per i nomi ed i tooltip, faccio la cosa più semplice (un buon programmatore preleverebbe i nomi da un dizionario, in modo da supportare la localizzazione in lingua dell'applicazione). Per le immagini, riciclo quelle che ho utilizzato per la palette dei comandi (che a questo punto non ha più senso di esistere...). Infine, per ogni elemento, dico di inviare un apposto messaggio (l'argomento di `setAction:`) a `self`, ovvero alla classe documento stessa. Le azioni sono le stesse che erano associate alle icone della palette (quindi, nessun lavoro aggiuntivo...).

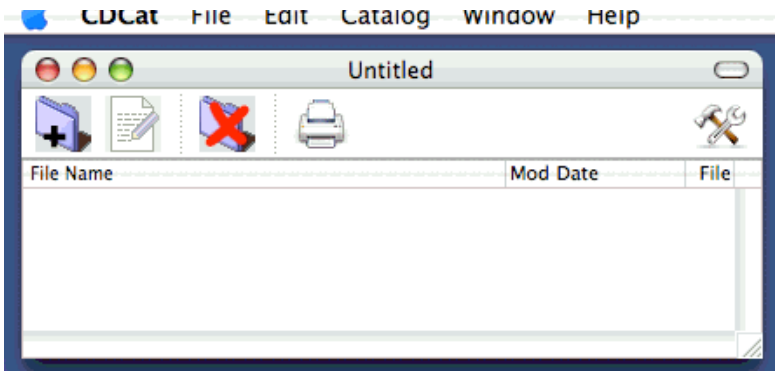
Non rimane altro che attaccare la toolbar alla finestra. Allo scopo ho definito un metodo apposito, `addToolbarToWin:`, da chiamare all'interno del metodo `windowControllerDidLoadNib:` nel seguente modo:

```
...
    // attacco la toolbar alla finestra
    [self addToolbarToWin: [ aController window] ];
...
```

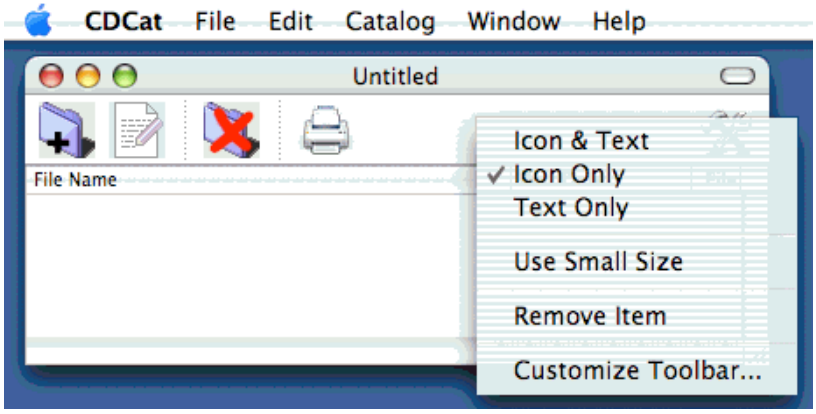
Il metodo utilizza un identificatore della toolbar che mi sono premunito di definire:

```
#define        CatDocToolbarId        @"tbName_CatDoc"

- (void)
addToolbarToWin: (NSWindow *) currWin
{
    // costruisco la nuova istanza della toolbar
    // la faccio autorelease perche' sara' poi ritenuta dalla finestra
        NSToolbar  *toolbar  =  [[[NSToolbar  alloc]  initWithIdentifier:  CatDocToolbarId]
autorelease];

    // imposto le proprieta' di base della toolbar
    // permetto che l'utente la modifichi
    [toolbar setAllowsUserCustomization: YES];
    // lascio che conservi lo stato tra successivi lanci
    [toolbar setAutosavesConfiguration: YES];
    // dico che mostri solo le icone
    [toolbar setDisplayMode: NSToolbarDisplayModeIconOnly];
    // dico che il delegato della toolbar e' il documento stesso
    [toolbar setDelegate: self];
    // attacco la toolbar alla finestra
    [ currWin setToolbar: toolbar ];
}
```

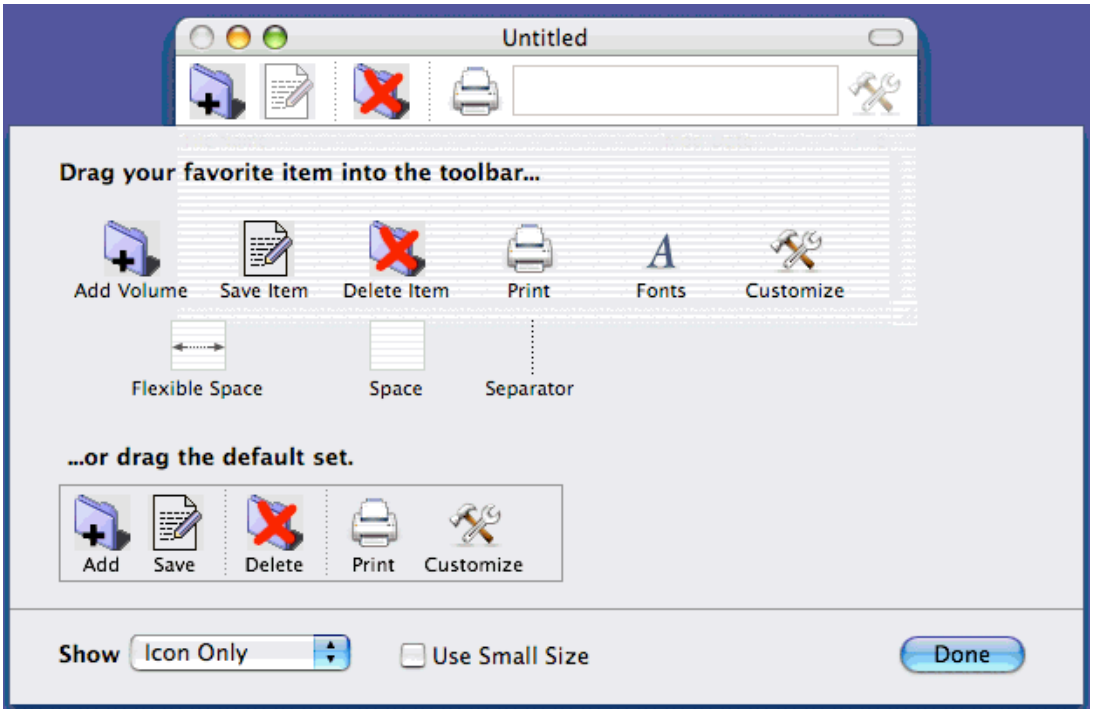
I commenti inseriti nel codice spiegano a sufficienza le operazioni in corso.



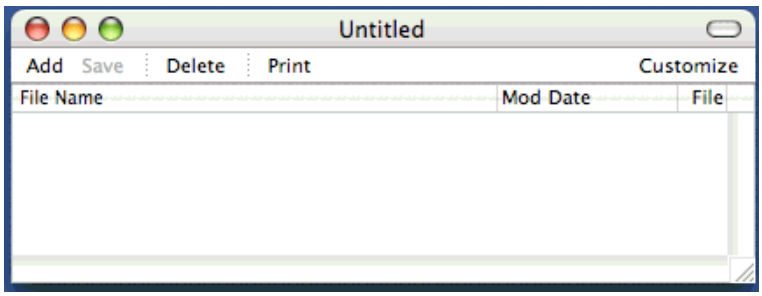
Questi tre metodi già permettono di vedere qualcosa funzionante. Da notare come sia comparsa (automaticamente, dal momento che non ho scritto una riga di codice al proposito) sulla barra del titolo un piccolo pulsante aggiuntivo che permette di mostrare/nascondere la toolbar; e come inoltre sia disponibile un menu contestuale sulla toolbar stessa.



Inoltre, il dialogo di personalizzazione della toolbar è perfettamente funzionante. Di più, la toolbar stessa mantiene l'ultima configurazione utilizzata tra un lancio e l'altro dell'applicazione.



(Nota personale: quanto mi diverto a programmare, quando scopro cose che non mi aspettavo di vedere; ad esempio, non avevo notato che è possibile ridurre tutte le toolbar a solo testo fino a che non ho esplorato questo argomento per aggiungerlo alla mia finestra).



Non tutto è ancora a posto: per questo occorre aggiungere qualche metodo delegato opzionale.

METODI OPZIONALI

Se uno prova a fare clic sull'icona della stampante, succede nulla. In effetti, leggendo la documentazione, scopro che per default, il pulsante di stampa invia il messaggio `printDocument:` al `firstResponder`. Verosimilmente, questo messaggio cade nel vuoto. Piuttosto che mettermi a scrivere questo metodo, preferisco rimandare l'argomento della stampa e copiare ciò che avviene selezionando la voce `Print` dal menu dell'applicazione. Esaminando infatti il file `MainMenu.nib`, il messaggio inviato dalla voce è il più stringato `print:`.

Per cambiare il comportamento dell'elemento, uso il metodo delegato opzionale `toolbarWillAddItem:`, chiamato quando l'applicazione sta per inserire un elemento nella toolbar. È proprio il posto dove inserire codice per predisporre un elemento al suo funzionamento. L'argomento del metodo è una notifica: si recupera l'elemento sotto esame estraendo dal dizionario interno alla notifica l'oggetto avente la chiave `item`.

```
- (void)
toolbarWillAddItem: (NSNotification *) notif
{
    // ricavo l'item che mi interessa
    NSToolbarItem *addedItem = [[notif userInfo] objectForKey: @"item"];
    // se l'elemento e' l'icona per stampare...
    if ([[addedItem itemIdentifier] isEqual: NSToolbarPrintItemIdentifier])
    {
        // aggiungo il tooltip
        [addedItem setToolTip: @"Print the Catalog"];
        [addedItem setAction: @selector(print:)];
    }
}
```

Una volta individuato l'elemento di stampa, ne cambio il tooltip (altrimenti uno scarno `Print`) ed assegno la nuova azione.

Per finire, ho voluto gestire l'elemento che salva il catalogo in modo tale che sia abilitato quando il file è stato modificato, e disabilitato altrimenti. In altre parole, l'elemento è abilitato solo quando ha senso salvare il catalogo su file.

Per realizzare ciò, si sfrutta il fatto che la toolbar esamina ogni elemento presente, ed per ogni elemento avente un valido target, invia a questo il messaggio `validateToolbarItem:`. È il posto adatto dove adeguare lo stato dell'elemento a seconda del documento (ad esempio, potrei pensare di cambiare immagine, cose del genere). A me interessa solamente abilitare o disabilitare l'elemento per la stampa:

```
- (BOOL)
validateToolbarItem: (NSToolbarItem *) toolbarItem
{
    // l'item salva va bene solo se il documento e' stato cambiato
    if ([[toolbarItem itemIdentifier] isEqual: TBItId_SaveItem])
        return ( [self isDocumentEdited] ) ;
    // tutti gli altri sono abilitati
    return ( YES );
}
```

Il metodo `isDocumentEdited` restituisce lo stato corrente del documento. Inizialmente a `NO`, diventa `YES` quando, ad esempio, si aggiunge un volume: il cambiamento di stato è dovuto all'istruzione

```
    [ self updateChangeCount: NSChangeDone ] ;
```

presente, ad esempio, all'interno del metodo `addVolume2Cat:...`.

Capitolo 029 - Ordine, ordine

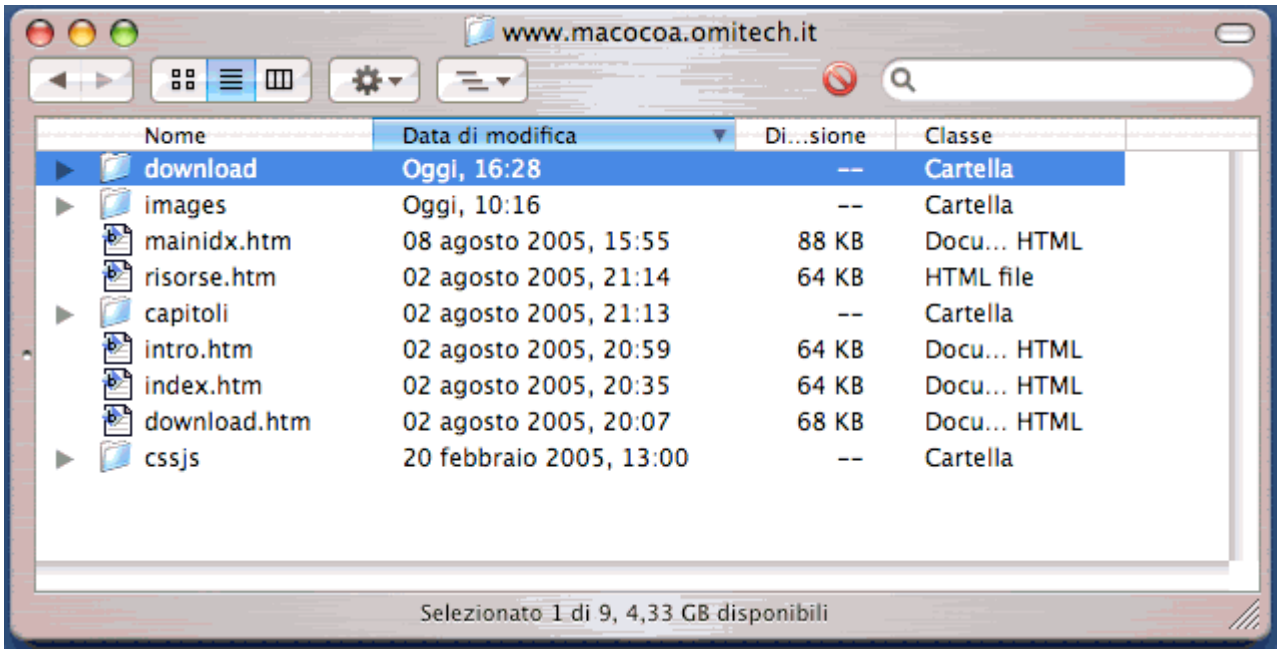
Finora i file all'interno della finestra di catalogo erano rappresentati secondo l'ordine con cui sono catalogati. Mi prefiggo di cambiare le cose, e di ordinare i file secondo i valori di una qualsiasi delle colonne che contengono gli attributi dei file.

Sorgenti: Dalla documentazione Apple.

Primo inserimento: 23 settembre 2002

Ordine ordine

Quando nella finestra del Finder trovate rappresentati i file contenuti in una directory nella vista elenco, è possibile ordinare i file secondo diversi criteri. Fondamentalmente, si utilizza uno degli attributi del file come chiave primaria per stabilire l'ordine, che può essere in una delle due direzioni possibili (per valori crescenti o per valori decrescenti). Ciò avviene selezionando la colonna che si vuole come chiave primaria; in Mac Os X, facendo clic sulla testa della colonna, si cambia la direzione dell'ordinamento. In Mac OS 9 ero invece abituato a fare clic su un riquadrino in alto a destro, all'incrocio tra la riga che contiene le teste delle colonne e la barra di scorrimento verticale.



Quel quadratino lì ha un nome particolare, è detto `corner view` ed è uno degli elementi costituenti la `NSOutlineView`, di cui finora ho tranquillamente dimenticato l'esistenza.

In questo capitolo intendo utilizzare quel quadratino per rappresentare la direzione d'ordinamento corrente, e nel contempo ordinare i file presenti secondo i valori di una delle colonne della `NSOutlineView` stessa.

Ordinare array

Per ottenere i miei scopi, devo modificare la classe `LSDataSource`. Preferisco modificarla piuttosto che farne una sottoclasse, altrimenti moltiplico in modo impressionante le classi presenti, cosa che non è mai pregevole dal punto di vista estetico.

La cosa più evidente è l'aggiunta di due nuove variabili d'istanza; la prima, `orderColumn`, specifica la chiave primaria di ordinamento. È una `NSString` che contiene l'identificatore della colonna prescelta. La seconda, `orderDirection`, è un semplice valore booleano che indica la direzione d'ordinamento.

Ho aggiunto subito i metodi accessor, che servon sempre e non fanno male. Mi sono anche ricordato di assegnare un buon valore iniziale e di deallogare la memoria alla distruzione dell'istanza.

Tutte le modifiche si possono incentrare nel metodo

```
- (id)    outlineView: (NSOutlineView *) outlineView child: (int) index
ofItem: (id) item ;
```

che ha subito una pesante modifica. In effetti, ordinare gli elementi del catalogo va fatto directory per directory; quindi, un posto per intervenire è quando la `NSOutlineView` richiede gli elementi da visualizzare, uno alla volta, procedendo per directory. Il metodo modificato fornisce per ogni elemento il 'figlio' i-esimo. Ero già intervenuto su questo metodo per limitare la visualizzazione dei file, limitandola a seconda delle preferenze espresse dall'utente. Adesso, invece che utilizzare l'ordine predefinito degli elementi, prima di fornire il figlio i-esimo, procedo ad ordinare la directory secondo quanto convenuto.

Agli scopi dell'ordinamento, all'interno della classe `NSArray` esiste un metodo apposito,

```
sortedArrayUsingFunction: <funzione di confronto> context: <contesto>
```

Questo metodo, invocato su di un array, produce un altro array, ordinato con chiave crescente secondo i risultati prodotti della funzione fornita come primo argomento. Questa funzione, detta funzione di confronto, è utilizzata per confrontare due elementi dello array, e deve restituire opportuni valori a seconda che il suo primo argomento sia minore, maggiore o guale al secondo argomento. Ogni volta che questa funzione è chiamata, gli viene passata come argomento il valore dell'argomento context: del metodo di ordinamento... Mi sto un po' pasticciando con la spiegazione, quindi provvedo a fornire l'esempio.

La funzione che effettua l'ordinamento deve essere definita così:

```
int nomeDellaFunzione(id num1, id num2, void *context)
```

ovvero, è una funzione che restituisce un intero (in realtà, un codice tra `NSOrderedAscending`, `NSOrderedDescending` o `NSOrderedSame`). I suoi argomenti sono due generici oggetti, ed il terzo argomento è appunto il valore di 'contesto'. Approfitto proprio di questo parametro per indicare il nome della colonna secondo cui

voglio sia fatto l'ordinamento (per nome, per dimensione, eccetera).

Per ora ho questa realizzazione; la cosa più pedissequa da fare è di esaminare l'argomento, ovvero l'attributo delle informazioni del file secondo cui confrontare i due elementi, estrarre l'attributo stesso e decidere quale dei due elementi è minore dell'altro in base al valore di questo attributo. Tuttavia, sfruttando il fatto che gli attributi sono solo di tre tipi, numeri, stringhe e date, ho raggruppato le operazioni per i primi due tipi. Ho definito un array con i nomi degli attributi di tipo omogeneo, ed ho verificato l'appartenenza dell'argomento a questi gruppi. Ho comunque dovuto esaminare a parte l'unico caso di data ed il confronto sulla dimensione del file, per i soliti problemi del metodo `valueForKey:` con i numeri `long long`. Ciò premesso, la funzione è di scorrevole lettura.

```
int
mySortFunc( id el1, id el2, void * ctx )
{
    // costruisco questi due array per evitare noiosi confronti
    NSArray *id_string = [NSArray arrayWithObjects:
        COLID_FILENAME, COLID_FULLPATH, COLID_GROUPNAME,
        COLID_OWNERNAME, COLID_FILETYPE, nil ];
    NSArray *id_number = [NSArray arrayWithObjects:
        COLID_FILESIZE, COLID_POSIXPERM, COLID_OSCREATOR,
        COLID_OSTYPE, COLID_FSFILENUM, COLID_FSNUM, nil ];
    // devo trattare in maniera differente il confronto tra date
    if ( [ (NSString*)ctx isEqual: COLID_MODDATE ] )
    {
        // ricavo le date
        NSDate    * d1 = [ (LSFileInfo*)el1 modDate];
        NSDate    * d2 = [ (LSFileInfo*)el2 modDate];
        // ed uso il confronto tra date
        return ( [ d1 compare: d2]);
    }
      // il confronto sulla dimensione del file (sempre per il solito
misterioso
    // motivo i long long non funzionano con valueForKey
    if ( [ (NSString*)ctx isEqual: COLID_FILESIZE ] )
    {
        // ricavo le due dimensioni
        long long    tmp1 = [ (LSFileInfo*)el1 fileSize ] ;
        long long    tmp2 = [ (LSFileInfo*)el2 fileSize ] ;
        // e le confronto brutalmente; notare i valori di ritorno
        if (tmp1 < tmp2)
            return NSOrderedAscending;
        else if (tmp1 > tmp2)
            return NSOrderedDescending;
        return NSOrderedSame;
    }
    // poi, per tutte le colonne che sono rappresentate da stringa
    if ( [ id_string containsObject: (NSString*)ctx ] )
    {
        // ricavo le strighe
        NSString *s1 = [ (LSFileInfo*)el1 valueForKey: ctx ];
        NSString *s2 = [ (LSFileInfo*)el2 valueForKey: ctx ];
        // ed uso il confronto tra stringhe
        return ( [ s1 compare: s2]);
    }
    if ( [ id_number containsObject: (NSString*)ctx ] )
    {
        // ricavo i due numeri
        NSNumber * s1 = [ (LSFileInfo*)el1 valueForKey: ctx ];
```

```
        NSNumber * s2 = [ (LSFileInfo*)el2 valueForKey: ctx ];
        // ed uso il confronto tra stringhe
        return ( [ s1 compare: s2]);
    }
    // qui ci arrivo solo se la colonna non e' una di quelle indicate
    // ritorno qualcosa a caso...
    return ( NSOrderedSame );
}
```

A questo punto diventa comprensibile il metodo `outlineView:...` principale. Invece che pigliare direttamente l'elemento i-esimo del vettore che contiene tutti i figli del file in esame, prima costruisco un vettore con gli elementi ordinati secondo il criterio richiesto. Se la direzione di ordinamento è decrescente, mi limito a rovesciare brutalmente lo array stesso, invocando una funzione da me scritta (non ho trovato un metodo apposito nella classe `NSMutableArray`, e tanto meno in `NSArray`).

All'ultimo momento mi sono poi accorto di non poter più utilizzare la funzione che avevo scritto in precedenza quando si devono saltare i dotFiles (si chiamava `getNormalFile`), ed ho dovuto scriverne un'altra, che ho chiamato `getArrayNormalFile`.

```
- (id)
outlineView:        (NSOutlineView *) outlineView
    child:        (int) index
    ofItem:        (id) item
{
    NSArray * newArray ;
    NSMutableArray * tmpArray ;
    if (item == nil)
    {
        // ordino gli elementi secondo criterio
        newArray = [ [self startPoint] sortedArrayUsingFunction: mySortFunc
            context: [ self orderColumn]] ;
        tmpArray = [ NSMutableArray arrayWithCapacity: [ newArray count]];
        [ tmpArray addObjectsFromArray: newArray ];
        // se la direzione e' FALSE, rovescio il vettore
        if ( [ self orderDirection ] == FALSE )
            reverseArray( tmpArray) ;
        return( [ tmpArray objectAtIndex: index ]);
    }
    // negli altri casi, recupero la lista
    newArray = [[ item fileList] sortedArrayUsingFunction: mySortFunc
            context: [ self orderColumn]] ;
    tmpArray = [ NSMutableArray arrayWithCapacity: [ newArray count]];
    [ tmpArray addObjectsFromArray: newArray ];
        if ( [ self orderDirection ] == FALSE )
            reverseArray( tmpArray) ;
    // se visualizzo anche i dotFiles, non c'e' problema
    if ( [[ UserPrefs getPrefValue:keyShowDotFiles] boolValue] )
        return( [ tmpArray objectAtIndex: index ]);
    // altrimenti, e' un bel pasticcio, devo saltare i dotfiles
    return ( getArrayNormalFile ( tmpArray, index ) );
}
```

Mostro adesso le due funzioni citate e non ancora descritte.

Per rovesciare un array, la prima cosa che mi è venuta in mente è stata quella di scambiare tra loro il primo elemento con l'ultimo, il secondo con il penultimo, e così via,

fino a che gli scambi non raggiungevano il centro del vettore. La cosa funziona sia con un numero pari che dispari di elementi, basta solo ricordarsi di fermarsi al momento giusto....

```
void
reverseArray ( NSMutableArray * locArr )
{
    // conto quanti elementi ci sono
    short    numelem = [ locArr count];
    short    i ;
    // e li scambio a coppie
    for ( i= 0 ; i < numelem / 2 ; i ++ )
    {
        // uso il metodo apposito...
                [  locArr  exchangeObjectAtIndex:  (i)  withObjectAtIndex:
(numelem-i-1) ];
    }
}
```

La funzione ha una storia lunga: in primo luogo, il rovesciamento in loco dello array mi costringe a dichiararlo `NSMutableArray`; alternativamente, avrei potuto crearne uno nuovo.... In secondo luogo, avrei potuto fare il tutto in un'altra maniera: se al posto di usare sempre la stessa funzione d'ordinamento ne definivo due, una speculare all'altra (la prima `mySortFunc` restituisce 'maggiore' dove la seconda, diciamo `revSortFunc`, restituisce 'minore') potevo eseguire l'ordinamento in ordine crescente o decrescente scegliendo l'una o l'altra funzione, più o meno come nel frammento di codice seguente:

```
if ( [ self orderDirection ])
    newArray = [ [self startPoint] sortedArrayUsingFunction: mySortFunc
            context: [ self orderColumn]] ;
else         newArray  =  [  [self  startPoint]  sortedArrayUsingFunction:
revSortFunc
            context: [ self orderColumn]] ;
```

Infine, potevo estendere il contesto inserendo nel parametro `context` non solo l'identificatore della colonna con cui ordinare il vettore, ma anche la direzione di ordinamento...

Manca la funzione per estrarre un elemento dal vettore, tenendo eventualmente conto del fatto che si devono saltare i `dotFiles`:

```
FileStruct *
getArrayNormalFile ( NSArray * directory, short index )
{
    short    i, count ;

    count = 0 ;    // contatore corrente
    for ( i = 0 ; i < [directory count] ; i++ )
    {
        FileStruct * currFile ;
        // recupero il file
        currFile = [directory objectAtIndex:i] ;
        // se devo saltarlo, passo avanti
        if ( skipDotFile( [ currFile fileName ] ) )
            continue ;
        // sono arrivato all'indice richiesto ?
        if ( count == index )
```

```
            // si, restituisco l'elemento corrente
            return ( currFile );
        // se arrivo qui, non ho ancora raggiunto l'elemento
        count += 1 ;
    }
    // ovviamente, qui non dovrei mai arrivare...
    return ( nil) ;
}
```

Non c'è molto da dire: in pratica si salta un livello gerarchico, utilizzando direttamente il vettore degli elementi piuttosto che ricavarlo indirettamente.

UN ORDINE ALTERNATIVO

Non sono del tutto soddisfatto di come ho realizzato la cosa. In effetti, la visualizzazione di ogni elemento provoca sempre un nuovo riordinamento, con conseguente creazione di vettori, copiatura, eccetera. Un gran lavorio che forse si può evitare, ma che non ho voglia al momento di modificare.

L'unico miglioramento che mi sono permesso riguarda la funzione di ordinamento, che ho riscritto in questo modo:

```
int
mySortFunc( id el1, id el2, void * ctx )
{
    id    s1, s2 ;
      // il confronto sulla dimensione del file (sempre per il solito
misterioso
    // motivo i long long non funzionano con valueForKey
    if ( [ (NSString*)ctx isEqual: COLID_FILESIZE ] )
    {
        // ricavo le due dimensioni
        long long    tmp1 = [ (LSFileInfo*)el1 fileSize ] ;
        long long    tmp2 = [ (LSFileInfo*)el2 fileSize ] ;
        // e le confronto brutalmente; notare i valori di ritorno
        if (tmp1 < tmp2)
            return NSOrderedAscending;
        else if (tmp1 > tmp2)
            return NSOrderedDescending;
        return NSOrderedSame;
    }
    // per tutte le altre colonne, uso il polimorfismo (late binding)
    // ricavo i due elementi, non meglio specificando la natura
    // possono essere stringhe, numeri o da te
    s1 = [ (LSFileInfo*)el1 valueForKey: ctx ];
    s2 = [ (LSFileInfo*)el2 valueForKey: ctx ];
    // pero' tutti e tre i tipi rispondono al messaggio compare:
    // e quindi, l'istruzione seguente chiama il metodo appropriato
    return ( [ s1 compare: s2]);
    // con questa versione del codice, il compilatore mi da un po'
    // di warning, proprio su questa istruzione...
}
```

Sfrutto il polimorfismo ovvero il late-binding, una delle caratteristiche qualificanti della programmazione object-oriented. Poiché tutti e tre i tipi di dati (numeri, stringhe e date) sono in grado di rispondere correttamente al messaggio `compare:`, non mi preoccupo di stabilire in anticipo la natura degli elementi da confrontare, ma lascio che

sia l'ambiente operativo a decidere quale dei tre metodi utilizzare a seconda della natura dell'oggetto, nota solo al momento dell'esecuzione (va da sé che per il solito motivo estraggo il confronto per le dimensioni dei file...). Non so se questa realizzazione sia computazionalmente più efficiente della precedente; certo, è più compatta.

L'ORDINE DELL'ORDINE

Una volta che ho scritto le istruzioni per l'ordinamento degli elementi da visualizzare, occorre informare la visualizzazione di come procedere all'ordinamento. In altre parole, devo trovare il posto più adatto per assegnare un valore alla variabile d'istanza `orderColumn` della classe sorgente di dati.

La prima soluzione che ho trovato, che non è detto sia la migliore, è di sfruttare una delle tante notifiche cha la `NSOutlineView` produce. In particolare, l'ambiente operativo invoca il metodo `outlineView:shouldSelectTableColumn:` al delegato della `NSOutlineView` ogni volta che l'utente fa clic per selezionare una intera colonna. Scopo del metodo è di decidere se la colonna possa o meno essere selezionata. Nel mio caso, risponde sempre e senz'altro di sì, ma ne approfitto per impostare il criterio di ordinamento:

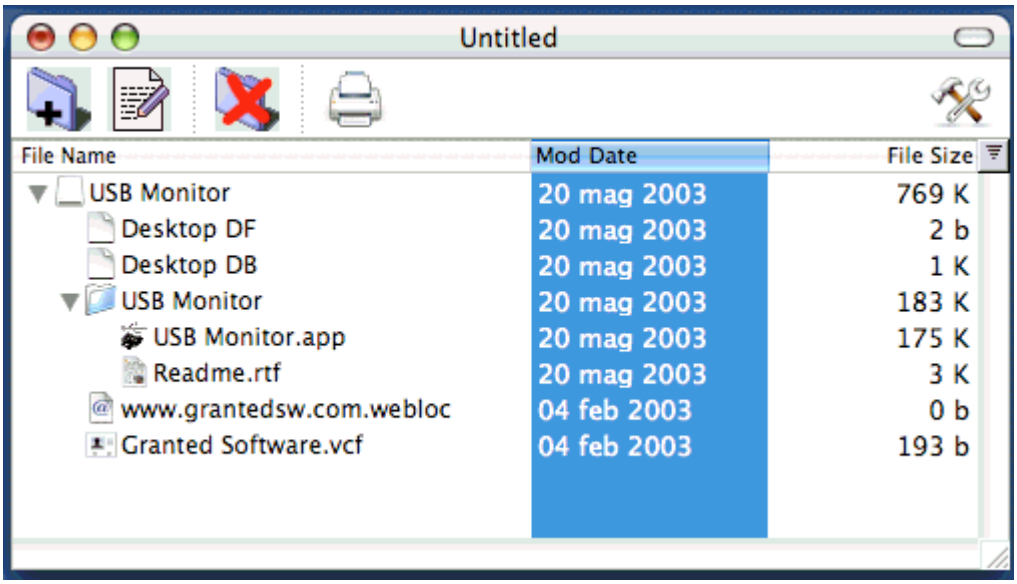
```
- (BOOL)
outlineView:(NSOutlineView *)outlineView
    shouldSelectTableColumn:(NSTableColumn *)tableColumn
{
    // assegno la nuova chiave di ordinamento al dataSource
    [[ self dataSource] setOrderColumn: [ tableColumn identifier]];
    // rinfresco la finestra con i nuovi dati
    [ [self outlineView] reloadData ];
    return ( YES );
}
```

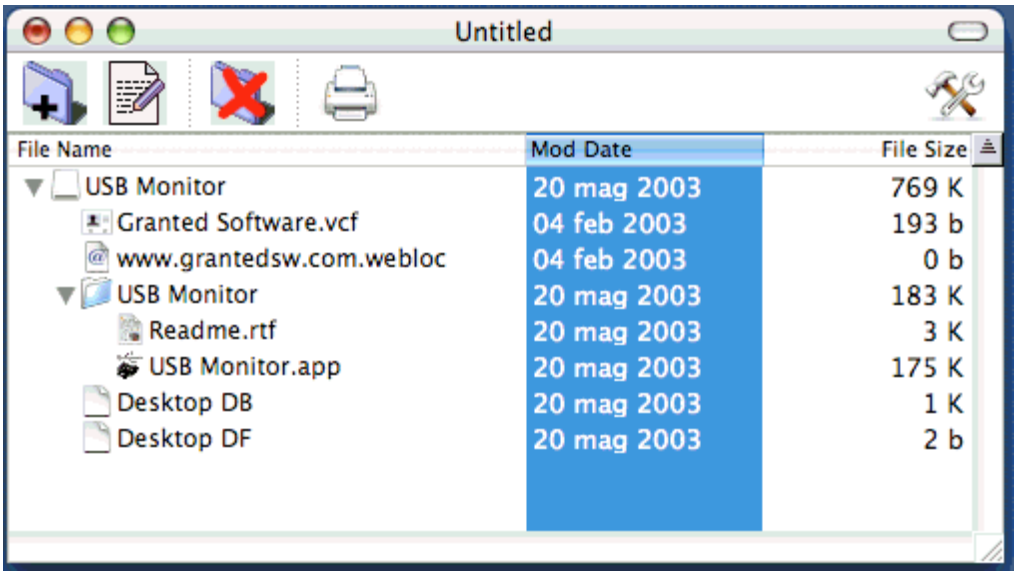
Con queste istruzioni, cambio il criterio di ordinamento, ma non ho ancora un meccanismo che imponga la direzione dell'ordinamento stesso. Dicevo all'inizio che volevo utilizzare la `cornerView`, il quadratino in alto a destra della finestra, all'incrocio tra la riga delle intestazioni e la colonna dove si trova la barra di scorrimento verticale. La `cornerView` è parte strutturale della `NSOutlineView`, e si gestisce passando attraverso di essa.

Decido che questa view è in realtà un pulsante, quindi un oggetto della classe `NSButton`. Non ho la possibilità di impostare questa view all'interno di IB, quindi lo faccio da programma:

```
- (void)
windowControllerDidLoadNib:    (NSWindowController *) aController
{
    NSButton    * sortBtn ;
##codice##
    sortBtn = [[ NSButton alloc] init ] ;
    [ sortBtn setImage: [NSImage imageNamed: @"imgUp"] ];
    [ sortBtn setTarget: self ];
    [ sortBtn setAction: @selector( changeSortOrder:) ];
    [ outlineView setCornerView: sortBtn ];
##codice##
```

```
}
```





Come si può vedere, la cosa è molto semplice: costruisco un oggetto `NSButton`; gli attribuisco una immagine; tramite il meccanismo target/action faccio in modo che ogni clic su di esso invochi un opportuno metodo; dico alla `NSOutlineView` di utilizzare questo pulsante come `cornerView`.

Il metodo invocato è molto semplice:

```
- (void)
changeSortOrder: (id) sender
{
    // rovescio l'ordine: esamino il valore corrente...
    if ( [ [ self dataSource] orderDirection ])
    {
        // ... e lo rovescio
        [ [ self dataSource] setOrderDirection: FALSE ];
        // adeguo l'immagine della corner view
           [[[self outlineView] cornerView] setImage: [NSImage imageNamed:
@"imgDw"]];
```

```
    }
    else
    {
        [ [ self dataSource] setOrderDirection: TRUE ];
        // adeguo l'immagine della corner view
           [[[self outlineView] cornerView] setImage: [NSImage imageNamed:
@"imgUp"]];
    }
    // rinfresco la finestra con i nuovi dati
    [ [self outlineView] reloadData ];
}
```

Si limita infatti a cambiare l'immagine del pulsante a seconda della direzione di ordinamento, di modificare il contenuto delle variabile d'istanza e di forzare il rinfresco dei dati della `NSOutlineView`.

Capitolo 030 - Un ordine migliore

Lo scorso capitolo mi ero lamentato dell'inefficienza dell'ordinamento. Metto assieme questa cosa con altri miglioramenti riguardanti l'interfaccia per dare luogo ad un nuovo ordine mondiale.

Sorgenti: Dalla documentazione Apple

Primo inserimento: 26 settembre 2002

Due funzioni anzi una

In primo luogo, ho deciso di utilizzare due funzioni al posto di una sola per effettuare l'ordinamento di un vettore; avevo già accennato alla cosa come una delle possibilità. Ebbene, neppure questo è vero. In realtà ne uso una sola, anzi tre.

Per ordinare un vettore, si deve definire una funzione ausiliaria di ordinamento che decida quale, fra due elementi di un vettore, venga prima nell'ordine deputato. Nella realizzazione del capitolo precedente, usavo sempre la stessa funzione, a prescindere dalla direzione di ordinamento. Solo dopo aver ordinato il vettore, eventualmente lo rovesciavo se l'ordine non era quello ascendente. Scegliendo invece la soluzione con due funzioni differenti per ordine ascendente e discendente, devo scrivere nel metodo chiamante qualcosa del tipo:

```
if ( dir == TRUE )
      newArray = [ vettore sortedArrayUsingFunction: ascSortFunc context:
chiave] ;
else    newArray = [ vettore sortedArrayUsingFunction: desSortFunc context:
chiave] ;
```

Ora, le due funzioni `ascSortFunc` e `desSortFunc` sono l'una opposta dell'altra. Potrei scrivere la seconda in termini della prima, rovesciando il risultato della prima, oppure, per evitare complicazioni (la funzione è passata come argomento ad un metodo di libreria...), passo per una terza funzione. Ecco la realizzazione finale:

```
int
ascSortFunc( id el1, id el2, void * ctx )
{
    return ( lsSortFunction( el1, el2, ctx ));
}

int
desSortFunc( id el1, id el2, void * ctx )
{
    return ( lsSortFunction( el2, el1, ctx ));
}
```

È facile vedere come le due funzioni forniscano proprio risultati di natura opposta, semplicemente perché scambio tra loro gli oggetti da confrontare (la genialità di questa soluzione mi abbacina ogni volta che la osservo).

E la funzione `lsSortFunction` non è altro che la mia vecchia funzione di ordine crescente, scritta all'origine, e che riporto qui solo per completezza:

```
int
lsSortFunction( id el1, id el2, void * ctx )
{
    id    s1, s2 ;
      // il confronto sulla dimensione del file (sempre per il solito
misterioso
    // motivo i long long non funzionano con valueForKey
    if ( [ (NSString*)ctx isEqual: COLID_FILESIZE ] )
    {
        // ricavo le due dimensioni
        long long    tmp1 = [ (LSFileInfo*)el1 fileSize ] ;
        long long    tmp2 = [ (LSFileInfo*)el2 fileSize ] ;
        // e le confronto brutalmente; notare i valori di ritorno
        if (tmp1 < tmp2)
            return NSOrderedAscending;
        else if (tmp1 > tmp2)
            return NSOrderedDescending;
        return NSOrderedSame;
    }
    // per tutte le altre colonne, uso il polimorfismo (late binding)
    // ricavo i due elementi, non meglio specificando la natura
    // possono essere stringhe, numeri o da te
    s1 = [ (LSFileInfo*)el1 valueForKey: ctx ];
    s2 = [ (LSFileInfo*)el2 valueForKey: ctx ];
    // pero' tutti e tre i tipi rispondono al messaggio compare:
    // e quindi, l'istruzione seguente chiama il metodo appropriato
    return ( [ s1 compare: s2]);
    // con questa versione del codice, il compilatore mi da un po'
    // di warning, proprio su questa istruzione...
}
```

ORDINE IN LOCO

Bene. Ora le cose si complicano. Una pecca della realizzazione del capitolo precedente era costringere l'ordinamento del vettore ad ogni invocazione del metodo `outlineView:child:ofItem:`. Per evitare troppe ordinazioni, ho pensato di ordinarlo una sola volta, la prima, e di utilizzare il vettore già ordinato le volte successive. In pratica, decido di ordinare direttamente in loco il vettore `fileList` della classe `FileStruct`. Allo scopo, aggiungo alla classe altre due variabili d'istanza, le stesse già viste nel caso della `LSDataSource`:

```
// criterio e direzione di ordine
NSString        * orderColumn ;
BOOL            orderDirection ;
```

Inoltre, aggiungo un metodo che realizzi l'ordinamento del vettore secondo chiave e direzione specificate:

```
- (void)
sortFileList: (NSString*) type
    direction: (BOOL) dir
{
    NSArray * newArray ;
```

```
    NSMutableArray * tmpArray ;

    // se non ci sono elementi da ordinare, abbiamo finito
    if ( ! [self fileList] ) return ;
    // il vettore e' gia' ordinato se chiave e direzione coincidono
    if ( [ orderColumn isEqual: type ] && dir == orderDirection )
        return ;
    // altrimenti, bisogna ordinare il vettore; uso una delle due funzioni
    // a seconda della direzione di ordinamento
    if ( dir == TRUE )
         newArray = [ [self fileList] sortedArrayUsingFunction: ascSortFunc
context: type] ;
      else       newArray  =  [  [self  fileList]  sortedArrayUsingFunction:
desSortFunc context: type] ;
    // trasformo il risultato in un NSMutableArray
    tmpArray = [ NSMutableArray arrayWithCapacity: [ newArray count]];
    [ tmpArray addObjectsFromArray: newArray ];
    // assegno le nuove variabili d'istanza
    [ self setFileList: tmpArray];
    [ self setOrderColumn: type ];
    orderDirection = dir ;
    return ;
}
```

A parte il controllo iniziale sul fatto che il vettore sia tale, controllo subito se il vettore non è già ordinato secondo la chiave e la direzione richiesta. Ciò avviene confrontando gli argomenti di invocazione del metodo con le variabili d'istanza. Se il vettore è già ordinato, ho già finito; se invece per qualche motivo occorre riordinare il vettore, procedo appunto al riordino. Una volta effettuata l'ordinazione, impongo il vettore risultante come nuovo valore di `fileList`; completo il tutto assegnando nuovi valori alle variabili d'istanza che memorizzano il criterio d'ordine del vettore.

Questo metodo è chiamato in due posti differenti.

In primo luogo, all'interno del metodo `initTreeFromPath:`, subito dopo la costruzione del vettore `fileList` con il contenuto della directory. Decido di ordinarlo per nome in ordine crescente:

```
##codice##
     fileList = [[ NSMutableArray alloc ] initWithCapacity: [ tmpfileList
count] ];
    [ fileList setArray: tmpfileList ];
    [ self sortFileList: COLID_FILENAME direction: TRUE ];
##codice##
```

Ed in secondo luogo, nel punto chiave dove eseguire l'ordinamento, ovvero all'interno dell'ormai vecchia conoscenza `outlineView:child:ofItem:` (della classe `LSDataSource`), che ha subito una notevole revisione:

```
- (id)
outlineView:        (NSOutlineView *) outlineView
    child:        (int) index
    ofItem:        (id) item
{
    NSArray * newArray ;
    NSMutableArray * tmpArray ;
    if (item == nil)
```

```
    {
        // ordino gli elementi secondo criterio
        if ( [ self orderDirection ] )
                newArray = [ [self startPoint] sortedArrayUsingFunction:
ascSortFunc
                context: [ self orderColumn]] ;
          else     newArray = [ [self startPoint] sortedArrayUsingFunction:
desSortFunc
                context: [ self orderColumn]] ;
        // trasformo il risultato in un NSMutableArray
        tmpArray = [ NSMutableArray arrayWithCapacity: [ newArray count]];
        [ tmpArray addObjectsFromArray: newArray ];
        return( [ tmpArray objectAtIndex: index ]);
    }
    // negli altri casi, recupero la lista
         [ item sortFileList: [ self orderColumn] direction: [ self
orderDirection ] ];
    // se visualizzo anche i dotFiles, non c'e' problema
    if ( [[ UserPrefs getPrefValue:keyShowDotFiles] boolValue] )
        return( [item getFileAtIndex:index]);
    // altrimenti, e' un bel pasticcio, devo saltare i dotfiles
    return ( getNormalFile ( item, index ) );
}
```

Qui ho preso alcune decisioni non banali, con diverse conseguenze. La prima decisione ha a che fare con gli elementi di primo livello, in pratica con l'elenco dei volumi. Questi sono i figli dell'elemento iniziale della `NSOutlineView`, di valore `nil`, e sono trattati dalle istruzioni contenute delle parentesi graffe più interne. Per non complicarmi la vita (dopotutto, dovrebbero essere pochi i volumi contenuti all'interno di un catalogo), ho lasciato che il vettore sia ordinato ogni volta che il metodo è invocato per l'elemento `nil`.

La seconda decisione è automatica, una volta notato che, avendo introdotto il metodo `sortFileList:direction:`, non serve più la vecchia funzione `getArrayNormalFile(.)`, ma si può tranquillamente ritornare alla precedente `getNormalFile(.)` (quando dicevo che avrei fatto e disfatto, non scherzavo...).

Infine, c'è il problema di dove piazzare le funzioni di ordinamento: mi servono sia per la classe `FileStruct`, ma anche per la `LSDataSource`. Trattandosi di funzioni, non ho trovato di meglio che inserirle nel file bidone `djZeroUtils.m`.

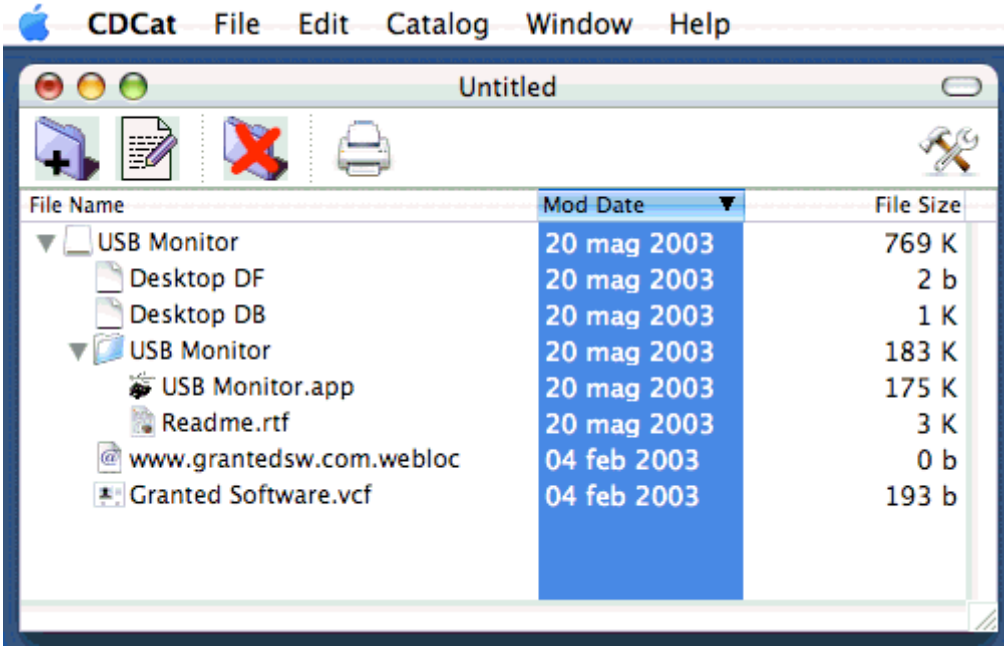
(QUASI) COME IL FINDER

Messo a punto il meccanismo interno, comincio a ripulire l'interfaccia. Diciamolo, l'icona in alto a destra non piace a nessuno. L'idea è di effettuare l'ordinamento degli elementi all'interno di un documento catalogo più o meno come fa il Finder.

La prima cosa che volevo fare era fare in modo che la direzione di ordinamento e la colonna con funzione di chiave fossero individuabili come succede nel Finder, con un bel triangolino di fianco al nome della colonna. Mi stavo avventurando in cose pericolosissime (come far diventare la cella di intestazione una `ImageAndTextCell` modificata) quando mi sono imbattuto, più o meno per caso, in una coppia di metodi della classe `NSOutlineView` che fanno proprio al caso mio: si chiamano `indicatorImageInTalbeColumn:` e `setIndicatorImageInTalbeColumn:`. Traduco testualmente dalla documentazione Apple:

Una **indicator Image** è una qualsiasi (piccola) immagine che è disegnata sul lato destro di una intestazione di colonna. Un esempio del suo uso è nell'applicazione `Mail` per indicare la direzione di ordinamento della colonna corrente che governa l'ordine dei messaggi in una mailbox.

Bingo.



Ecco allora la disciplina per effettuare gli ordinamenti all'interno di una finestra di catalogo. Facendo clic su di una colonna, il catalogo è ordinato secondo quella colonna. Facendo doppio clic su di una colonna, si rovescia la direzione di ordinamento secondo quella colonna.

Quanto detto comporta alcune modifiche nel metodo `windowControllerDidLoadNib:`; scompare la porzione di codice dedicata alla `cornerView` (anzi, scompare la `cornerView`), ed entrano due nuove porzioni di codice, la prima per inizializzare l'ordinamento all'interno della sorgente di dati, e la seconda per introdurre l'immagine nella colonna col nome del file (la sola che sono sicuro esistere all'inizio). Ne approfitto anche per impostare il metodo da chiamare quando si fa doppio clic sulla `NSOutlineView`:

```
    ##codice##
    // imposto l'ordine iniziale
          [ dataSource setOrderColumn: [ NSString stringWithString:
COLID_FILENAME]];
    [ dataSource setOrderDirection: TRUE ];
    ##codice##
    tableColumn = [outlineView tableColumnWithIdentifier: COLID_FILENAME];
    ##codice##
    // assegno alla colonna presente l'immagine che indica l'ordinamento
       [ outlineView setIndicatorImage: [NSImage imageNamed: @"imgUpS"]
inTableColumn: tableColumn ];
    [ outlineView setTarget: self ];
    // facendo doppio clic, si cambia la direzione di ordinamento
    [ outlineView setDoubleAction: @selector( changeSortOrder:) ];
    ##codice##
```

Per inciso, mi sono accorto di un errore che finora mi era sfuggito: il metodo `outlineView:shouldEditTableColumn:item:` fino ad ora era tranquillamente posizionato all'interno della classe `LSDataSource`. Non è quello il suo posto, visto che si tratta di un metodo che la `NSOutlineView` richiede alla classe delegata (che è `CatalogDoc`) e non alla classe sorgente di dati (`LSDataSource`). Il metodo va dunque spostato all'interno del file `CatalogDoc.m`. Ricordo che questo metodo, che si limita a rispondere NO, indica che nessuna cella della `NSOutlineView` è editabile; questo fa sì che un qualsiasi doppio clic all'interno della `NSOutlineView` si traduca automaticamente nell'invocazione del metodo `changeSortOrder:`.

Ciò ha conseguenze appunto sul metodo `changeSortOrder:`, che deve sincerarsi in primo luogo che sia selezionata una colonna affinché il doppio clic abbia senso. Stabilito ciò, determina la colonna, cambia l'indicatore di direzione nella colonna, rovescia la direzione di ordinamento e provoca il rinfresco dell'intera finestra:

```
- (void)
changeSortOrder: (id) sender
{
    NSTableColumn    * tableColumn ;

    // voglio che sia selezionata una intera colonna
    if ( [ sender selectedColumn ] == -1 ) return ;
    // determino allora quale colonna e' selezionata
    tableColumn = [ [outlineView tableColumns] objectAtIndex: [ outlineView
clickedColumn ]];
    // rovescio l'ordine negando il valore corrente
       [  [  self  dataSource]  setOrderDirection:  !  [  [  self  dataSource]
orderDirection ] ];
    // aggiusto l'immagine nello header della colonna
    if ( [ [ self dataSource] orderDirection ])
              [  sender  setIndicatorImage:  [NSImage  imageNamed:  @"imgUpS"]
inTableColumn: tableColumn ];
     else     [ sender setIndicatorImage: [NSImage imageNamed: @"imgDwS"]
inTableColumn: tableColumn ];
    // rinfresco la finestra con i nuovi dati
    [ outlineView reloadData ];
}
```

Infine, ritorno sul metodo `outlineView:shouldSelectTableColumn:`, che scatena generalmente il meccanismo di ordinamento:

```
- (BOOL)
outlineView:(NSOutlineView *) oView
    shouldSelectTableColumn:(NSTableColumn *)tableColumn
{
    NSArray        * tabLists = [ oView tableColumns ];
    NSEnumerator     * enumerator = [tabLists objectEnumerator];
    NSTableColumn    * tc ;

    // in mancanza di metodi migliori, tolgo a tutte le colonne
    // l'indicatore di ordinamento (ho gia' ricavato un enumerator
    // con tutte le colonne al momento presenti)
    while ((tc = [enumerator nextObject]))
        [ oView setIndicatorImage: nil inTableColumn: tc ];
    // imposto l'immagine nello header della colonna
    if ( [ [ self dataSource] orderDirection ])
              [  oView  setIndicatorImage:  [NSImage  imageNamed:  @"imgUpS"]
```

```
inTableColumn: tableColumn ];
      else      [ oView setIndicatorImage: [NSImage imageNamed: @"imgDwS"]
inTableColumn: tableColumn ];
    // assegno la nuova chiave di ordinamento al dataSource
    [ [ self dataSource] setOrderColumn: [ tableColumn identifier]];
    // rinfresco la finestra con i nuovi dati
    [ oView reloadData ];
    return YES;
}
```

Quando l'utente fa clic su di una colonna, occorre ripulire le altre colonna da ogni simbolo di ordinamento, ed inserirlo, nella direzione corretta, sulla colonna prescelta. Poi, si scatena il nuovo ordinamento impostando il nuovo valore della chiave di ordinamento e forzando il ridisegno della finestra.

Ho pasticciato parecchio su i file prima di arrivare a questa situazione, quindi è possibile che mi sia dimenticato di qualche modifica qui e lì...

Capitolo 031 - Ricerca

Questo capitolo racconta di come ho costruito una finestra per effettuare ricerche di elementi all'interno di un catalogo e di come le funzioni di ricerca sono realizzate.

Sorgenti: nessuna

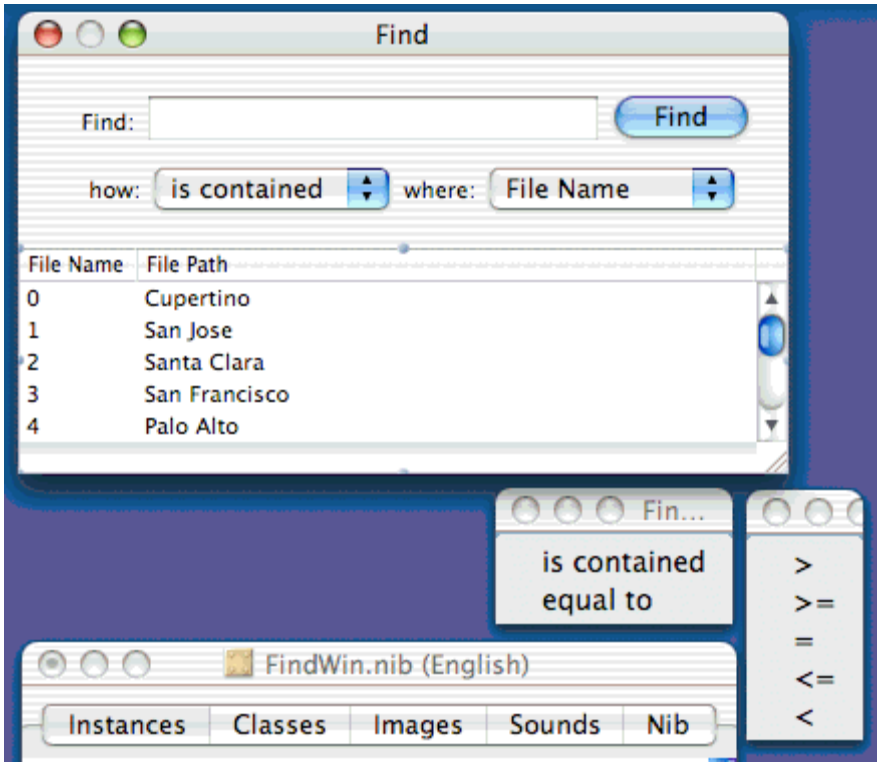
Primo inserimento: 7 ottobre 2002

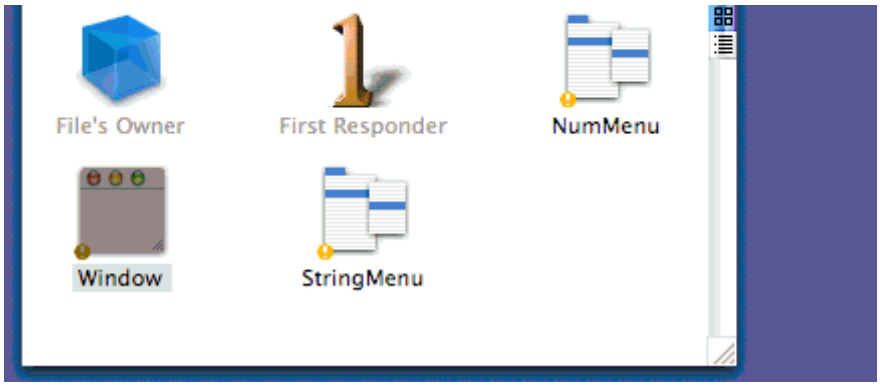
RICERCHE

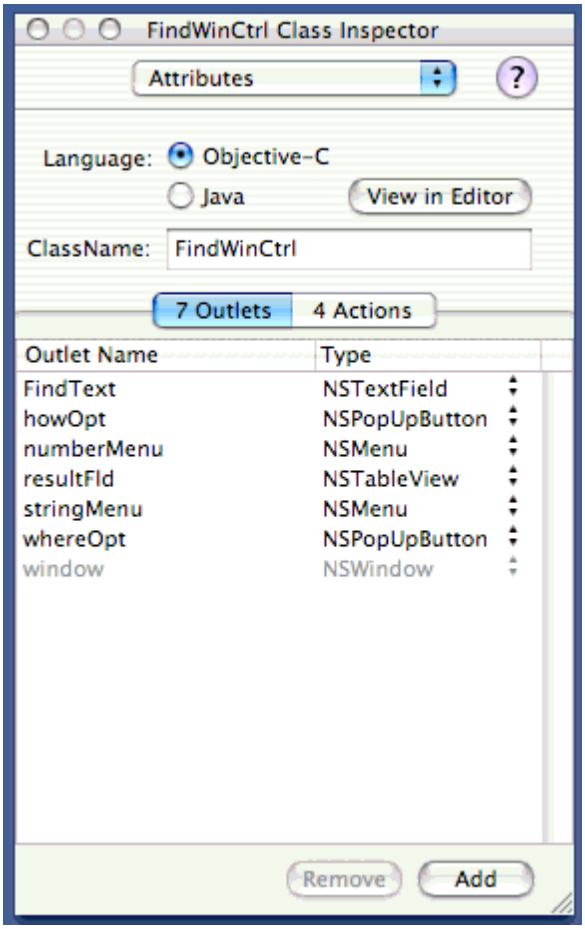
Un catalogo serve a contenere informazioni, ed uno degli scopi della sua esistenza è di facilitare l'estrazione di informazioni utili. Va da sé quindi che una delle funzioni principali dell'applicazione *CDCat* dovrebbe essere la ricerca di elementi all'interno di un catalogo. La ricerca poi è bene che sia parametrizzabile, nel senso che devo poter definire un criterio di ricerca più o meno sofisticato. Per cominciare, mi limito a cercare elementi che rispondano a dei criteri semplici: uno dei campi di un elemento catalogato (ciò che è visualizzato in una colonna) deve essere uguale (o contenere) un termine di riferimento. Per i numeri, posso pensare di cercare valori maggiori, minori o uguali di un numero dato.

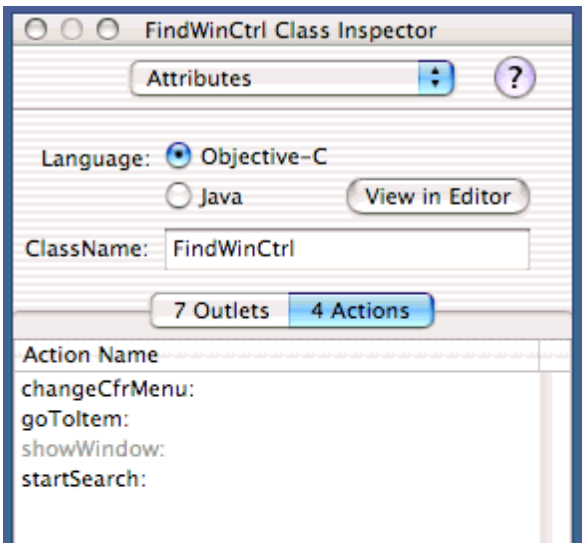
Una volta definito il criterio, passo ad estrarre da un catalogo gli elementi che soddisfano questo criterio; metterò il tutto all'interno di una tabella. Infine, facendo doppio clic sopra uno degli elementi estratti, è visualizzato l'elemento stesso all'interno del catalogo.

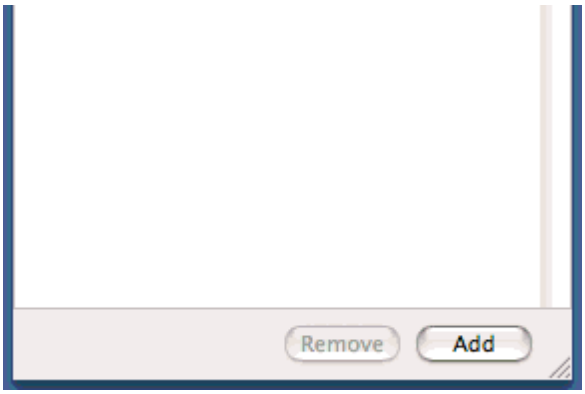
LA FINESTRA

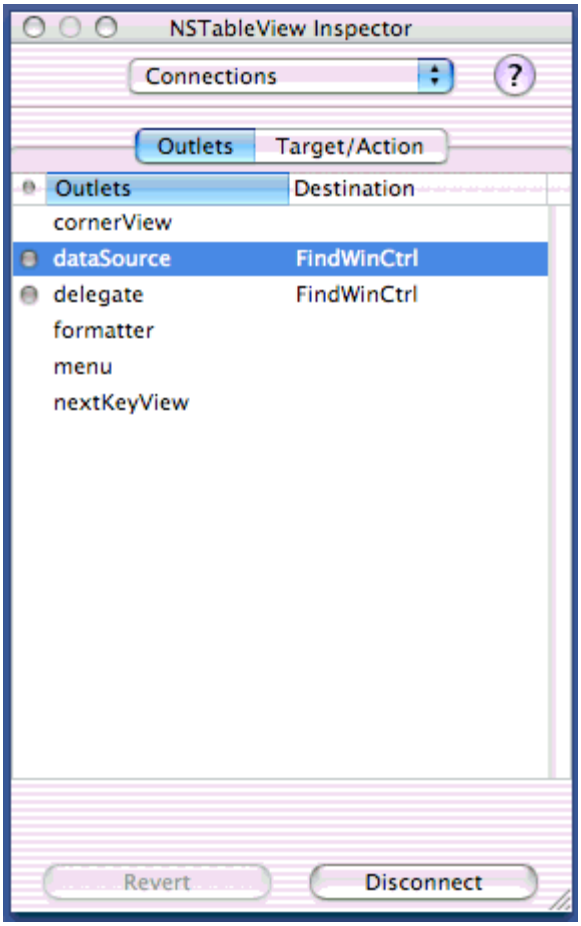



File's Owner
First Responder
NumMenu
Window
StringMenu


FindWinCtrl Class Inspector
Attributes
Language: Objective-C
Java
View in Editor
ClassName: FindWinCtrl
7 Outlets
4 Actions
Outlet Name
Type
FindText NSTextField
howOpt NSPopUpButton
numberMenu NSMenu
resultFld NSTableView
stringMenu NSMenu
whereOpt NSPopUpButton
window NSWindow
Remove
Add


FindWinCtrl Class Inspector
Attributes
Language: Objective-C
Java
View in Editor
ClassName: FindWinCtrl
7 Outlets
4 Actions
Action Name
changeCfrMenu:
goToItem:
showWindow:
startSearch:





Non sono mai stato bravo a disegnare le interfacce utente, e l'interfaccia della finestra di ricerca mostra tutta la sua pochezza. Definisco un campo per contenere il termine della ricerca, e due menu pop-up (oggetti della classe `NSPopUpButton`) per definire il criterio di ricerca. Il menu sulla destra presenta l'elenco dei campi dell'elemento di catalogo (che coincide con le colonne visualizzabili); il menu sulla sinistra (*menu di confronto*) invece dovrebbe essere contestuale alla scelta dell'altro menù. Molto semplicemente, se la scelta del *menudei campi* cade su una colonna che contiene una stringa, il menu di confronto permette di impostare una ricerca di stringa esatta (il campo dell'elemento deve coincidere con il testo impostato come termine della ricerca) oppure su stringa contenuta (il termine della ricerca deve essere contenuto all'interno del campo). Se invece la scelta del campo cade su una colonna che contiene un numero, il menu di confronto permette di impostare ricerche per valori maggiori, minori o uguali. La spiegazione è lunga e noiosa, e credo che chiunque abbia fatto una ricerca in precedenza capirà senza bisogno di ulteriori parole.

La finestra è completata dal pulsante che scatena la ricerca, e da una `NSTableView` destinata a contenere i risultati della ricerca stessa.

Già così, comincio ad avere dei problemi con l'interfaccia. Il primo problema è il comportamento dei vari elementi al ridimensionamento, che non mi piace per nulla. Allora, ho racchiuso il menu pop-up e la sua stringa di riferimento all'interno di una `NSBox`, completamente trasparente e senza bordi (praticamente invisibile). In questo modo, le specifiche di dimensionamento dinamico generano un comportamento che mi piace di più.

Il secondo problema è come associare il giusto menu di confronto; scopro che non si può fare da IB, ma dovrò scrivere un metodo apposta. Tuttavia, mi premunisco ed inserisco nel `nib` i due menu con le voci che mi interessano.

C'è poi il problema di capire quale voce di menu è selezionata correntemente. Scelgo il metodo del `tag`; ad ogni voce di menu assegno in IB un identificatore numerico nel campo `Tag` disponibile nella finestra di informazioni relativa alla voce. Poi, da programma, recupero il tag usando l'apposito metodo.

Concludo l'attività in IB definendo i vari outlet per collegare il controllore della finestra con i vari elementi dell'interfaccia, e dichiarando tre metodi/action: per cambiare il menu di confronto, per lanciare la ricerca, per evidenziare l'elemento trovato nel documento che lo contiene.

L'INTERFACCIA DELLA FINESTRA

Conviene adesso mostrare l'interfaccia della classe controllore della finestra di ricerca, per commentarne il contenuto:

```
@interface FindWinCtrl : NSWindowController
{
    IBOutlet NSTextField    * FindText;
    IBOutlet NSPopUpButton * howOpt;
    IBOutlet NSTableView    * resultFld;
    IBOutlet NSPopUpButton * whereOpt;
    IBOutlet NSMenu     * stringMenu;
    IBOutlet NSMenu     * numberMenu;

    // documento cui si riferisce la ricerca
    CatalogDoc          * currCatalog ;
    // vettore con i risultati
    NSMutableArray      * foundItems;
}
// per cominciare la ricerca
- (IBAction)startSearch: (id)sender;
// cambiamento del menu a seconda del tipo di campo
- (IBAction)changeCfrMenu: (id)sender;
// per evidenziare l'elemento selezionato
- (IBAction)goToItem: (id)sender;

+ (id) sharedFind ;

- (int) numberOfRowsInTableView:(NSTableView *)tv;
- (id) tableView:(NSTableView *)tv objectValueForTableColumn:(NSTableColumn
*)tc row:(int)row;
```

```
- (BOOL) tableView:(NSTableView *)tv shouldEditTableColumn:(NSTableColumn
*)tc row:(int)row ;

- (CatalogDoc *)currCatalog ;
- (void)setCurrCatalog :(CatalogDoc *)newCurrCatalog ;
- (NSMutableArray *)foundItems;
- (void)setFoundItems:(NSMutableArray *)newFoundItems;

@end
```

Dopo i vari outlet, ci sono altre due variabili d'istanza. La seconda è un vettore destinatati a contenere il risultato della ricerca. Nel mio caso, si tratta di una collezione di elementi della classe `FileStruct` (o meglio, di una serie di puntatori agli elementi); il vettore è inizialmente vuoto e sarà riempito dalla procedura di ricerca. La prima variabile invece è (un puntatore a) il documento (l'oggetto della classe `CatalogDoc`) sul quale è stata effettuata la ricerca. Questa variabile sarà utile quando voglio evidenziare l'elemento trovato.

Poi sono elencati i vari metodi; i primi sono le citate azioni associate agli elementi dell'interfaccia; c'è poi il metodo di classe per l'apertura (ed eventuale creazione) della finestra; i tre metodi successivi servono alla visualizzazione degli elementi trovati all'interno della tabella; concludono la classe i metodi accessor standard.

Prima di scrivere codice specifico della ricerca, ne approfitto per scrivere quelle due righe di codice per aprire la finestra. In primo luogo torno in IB; elimino la voce `Show Palette` dal menu `Window`, ed associo il metodo showPalette alla voce `Find...` del menu `Edit` (che ho trovato già predefinito in `MainMenu.lib`). Torno in XCode e cambio il metodo `showPalette:` perché apra la nuova finestra `FindWinCtrl`.

CAMBIO MENU AL VOLO

Devo scrivere il codice per cambiare il menu di confronto a seconda della selezione corrente del menu dei campi di ricerca. In IB mi sono preoccupato di associare al `NSPopUpButton` di destra l'azione `changeCfrMenu:` dall'elenco delle possibili azioni. Il metodo conseguente è molto semplice, avendo fatto un uso accorto dei tag:

```
- (IBAction)
changeCfrMenu:(id)sender
{
    // uso il tag del menu che ho accortamente predisposto in IB
    switch ( [[ whereOpt selectedItem] tag ] ) {
    // questi sono tutte strighe
    case MENUTAG_FILENAME:          case MENUTAG_FULLPATH:
    case MENUTAG_GROUPNAME:         case MENUTAG_OWNERNAME:
    case MENUTAG_FILETYPE:
        [ howOpt setMenu: stringMenu ];
        break ;
    // questi sono numeri, oppure date
    case MENUTAG_MODDATE:          case MENUTAG_FILESIZE:
    case MENUTAG_POSIXPERM:         case MENUTAG_OSCREATOR:
    case MENUTAG_OSTYPE:           case MENUTAG_FSFILENUM:
    case MENUTAG_FSNUM:
        [ howOpt setMenu: numberMenu ];
        break ;
    }
```

```
}
```

Ho predefinito tutti i tag all'interno del file `FindWinCtrl.h` utilizzando il vecchio concetto `#define`.

LANCIO DELLA RICERCA

Il clic sul pulsante `Find` della finestra deve lanciare la ricerca. Il metodo relativo, anche se piuttosto lungo, ricicla in realtà molti vecchi concetti, tra cui la sheet che informa l'utente delle operazioni in corso:

```
- (IBAction)
startSearch:(id)sender
{
    // inizializzo questo array per selezionare poi la colonna
    NSArray    * colIdArr = [NSArray arrayWithObjects:
        COLID_FILENAME, COLID_FULLPATH, COLID_MODDATE,
        COLID_FILESIZE, COLID_GROUPNAME, COLID_OWNERNAME,
        COLID_POSIXPERM, COLID_FILETYPE, COLID_OSCREATOR,
        COLID_OSTYPE, COLID_FSFILENUM, COLID_FSNUM,
        nil];
    // recupero cosa c'e' da cercare
    NSString    * txt2search = [ FindText stringValue];
    // recupero le opzioni di ricerca
    int        how = [[ howOpt selectedItem] tag ] ;
    // ed il campo dove cercare
    int        where = [[ whereOpt selectedItem] tag ] ;
    // mi faccio la finestra di attesa
    WaitPanCtrl     *tmpCVCtrl = [[ WaitPanCtrl alloc] init ] ;
    // faccio comparire la sheet che dice di attendere
    [ NSApp beginSheet: [ tmpCVCtrl window ]
        modalForWindow: [ NSApp mainWindow ]
        modalDelegate: self
        didEndSelector: nil
        contextInfo: nil
    ];
    // imposto il contenuto della shett stessa
    [ tmpCVCtrl setMainText: @"Searching" ];
    [ tmpCVCtrl animation: self ];
    // recupero il catalogo dove effettuare la ricerca
    // piglio tutti i documenti dell'applicazione, e scelgo quello che sta
    // davanti a tutti gli altri... non che questo meccanismo mi piaccia...
        [  self  setCurrCatalog:  (CatalogDoc*)[[  NSApp  orderedDocuments]
objectAtIndex: 0] ];

    // pulisco i risultati della ricerca precedente
    [self setFoundItems: [ NSMutableArray array ]];
    // e li tolgo dalla finestra di ricerca
    [ resultFld reloadData ];
    // adesso eseguo la ricerca, in maniera brutale, sulla sorgente dati
    [ [ [self currCatalog] dataSource ] searchFor: txt2search
        column: [ colIdArr objectAtIndex: where]
        options: how placeIn: foundItems];
    // forzo il rinfresco della finestra dei risultati
    [ resultFld reloadData ];
    // nascondo e chiudo la sheet
    [ [ tmpCVCtrl window ] setIsVisible: FALSE ];
    [ NSApp endSheet: [ tmpCVCtrl window ] ];
}
```

All'inizio, ho definito un vettore per tenere gli identificatori dei vari campi; l'identificatore utilizzato è estratto utilizzando come indice il valore del tag del menu prescelto. Il giochetto riesce perché ho definito i tag in maniera accorta (il tag Zero al nome del file, il tag 1 al path completo, e così via). Ci sono poi un po' di istruzioni per la visualizzazione della sheet `WaitPanCtrl`, molto simili a quelle già viste in un capitolo precedente. Altre istruzioni sono essenzialmente cosmetiche, per pulire la finestra dei risultati e per costringere il rinfresco della stessa una volta terminata la ricerca.

In effetti, in questo metodo, di funzioni di ricerca non ce ne sono. C'è solo una istruzione che invia un messaggio alla sorgente dei dati perché esegua la ricerca. È lì il cuore del problema.

CHI CERCA...

Dal momento che il catalogo è organizzato ad albero, ovvero in una struttura dati che si presta alla ricorsività, ancora una volta la procedura di ricerca avviene in maniera ricorsiva. La ricerca è lanciata sempre dalla sorgente dati:

```
- (void)
searchFor: (id) item2search
    column: (NSString*) colId
    options: (int) opt
    placeIn: (NSMutableArray*) foundItems
{
    // piglio un elenco di tutti i volumi presenti
    NSEnumerator     * enumerator = [[self startPoint] objectEnumerator];
    FileStruct    * item ;
        // li esamino uno ad uno
    while (item = [enumerator nextObject])
    {
        // e faccio una ricerca in profondita' sugli elementi contenuti
        [ item searchItemFor: item2search column: colId
            options: opt placeIn: foundItems ];
    }
}
```

Il metodo si limita ad estrarre un elenco dei volumi presenti nel catalogo e di eseguire la ricerca su ciascuno di essi. La ricerca è realizzata dal seguente metodo della classe `FileStruct`:

```
- (void)
searchItemFor: (id) item2search
    column: (NSString*) colId
    options: (int) opt
    placeIn: (NSMutableArray*) foundItems
{
    // piglio un elenco di tutti gli elementi presenti
    NSEnumerator     * enumerator = [[self fileList] objectEnumerator];
    FileStruct    * item ;
    // prima verifico se l'elemento corrente mi va bene
    if ( checkSearch( self, item2search, colId, opt) )
        [ foundItems addObject: self ] ;
    // poi esamino tutti i file contenuti
    while (item = [enumerator nextObject])
    {
        [ item searchItemFor: item2search column: colId
```

```
            options: opt placeIn: foundItems ];
    }
}
```

Ancora una volta, sto spostando il cuore del problema. In effetti questo metodo chiama la funzione `checkSearch`, che verifica se un dato elemento centra il criterio di ricerca. Poi, gestisce la discesa ricorsiva all'interno della struttura dati.

Insomma, finalmente, la funzione che realizza materialmente la ricerca, lunga e noiosa, ma molto semplice:

```
BOOL
checkSearch( LSFileInfo * item, id searchItem, NSString * colId, int opt )
{
    // colonne che contengono stringhe
    NSArray *id_string = [NSArray arrayWithObjects:
        COLID_FILENAME, COLID_FULLPATH, COLID_GROUPNAME,
        COLID_OWNERNAME, COLID_FILETYPE, nil ];
    // colonne che contengono numeri
    NSArray *id_number = [NSArray arrayWithObjects:
        COLID_FILESIZE, COLID_POSIXPERM, COLID_OSCREATOR,
        COLID_OSTYPE, COLID_FSFILENUM, COLID_FSNUM, nil ];

    // tratto a parte le date
    if ( [ colId isEqual: COLID_MODDATE ] )
    {
        // confronto tra data
        return ( [ [ item modDate] isEqualToDate: searchItem]);
    }
    // confronto sulla dimensione del file (sempre per il solito misterioso
    // motivo i long long non funzionano con valueForKey )
    if ( [ colId isEqual: COLID_FILESIZE ] )
    {
        // ricavo le due dimensioni
        long long    tmp1 = [ item fileSize ] ;
        long long    tmp2 = (long long) [ searchItem doubleValue ] ;
        // e le confronto brutalmente secondo l'opzione
        switch ( opt ) {
        case MENUTAG_NUM_GREAT :
            return ( tmp1 > tmp2 );
        case MENUTAG_NUM_GREATEQ :
            return ( tmp1 >= tmp2 );
        case MENUTAG_NUM_EQ :
            return ( tmp1 == tmp2 );
        case MENUTAG_NUM_LESSEQ :
            return ( tmp1 <= tmp2 );
        case MENUTAG_NUM_LESS :
            return ( tmp1 < tmp2 );
        default :
            return ( FALSE );
        }
    }
    // poi, per tutte le colonne che sono rappresentate da stringa
    if ( [ id_string containsObject: colId ] )
    {
        // ricavo la striga
        NSString *s1 = [ item valueForKey: colId ];
        // a seconda delle opzioni, faccio adeguato confronto
        if ( opt == MENUTAG_STR_ISEQUAL )
        {
            // confronto esatto tra stringhe
```

```
            return ( [ s1 isEqualToString: searchItem ]);
        }
        else if ( opt == MENUTAG_STR_ISCONTIANED )
        {
            NSRange    xx = [ s1 rangeOfString: searchItem ];
            return ( xx.location != NSNotFound ) ;
        }
    return ( FALSE );
    }
    // poi, per tutti gli altri numeri
    if ( [ id_number containsObject: colId ] )
    {
        // ricavo il numero
        NSNumber    * n1 = [ item valueForKey: colId ];
        NSComparisonResult    xx ;
        // a seconda delle opzioni, faccio il confronto
        switch ( opt ) {
        case MENUTAG_NUM_GREAT :
        case MENUTAG_NUM_GREATEQ :
            xx = [ n1 compare: searchItem ];
            return ( xx == NSOrderedDescending ) ;
        case MENUTAG_NUM_EQ :
            // confronto esatto
            return ( [ n1 isEqualToNumber: searchItem ]);
        case MENUTAG_NUM_LESSEQ :
        case MENUTAG_NUM_LESS :
            xx = [ n1 compare: searchItem ];
            return ( xx == NSOrderedAscending ) ;
        default :
            return ( FALSE );
        }
    }
    return FALSE ;
}
```

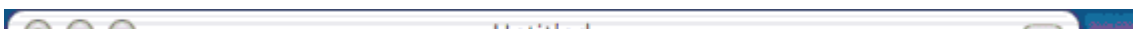
Ci sono essenzialmente quattro grosse ripartizioni della funzione, corrispondente alle quattro categorie di campi sui quali si può effettuare la ricerca. Le date e la dimensione del file sono trattate a parte, la prima per via della natura dell'oggetto, la seconda per il solito motivo dell'estrazione dei numeri `long long`. Il grosso del lavoro coinvolge le restanti due categorie, di ricerca in un testo e di ricerca in un numero.

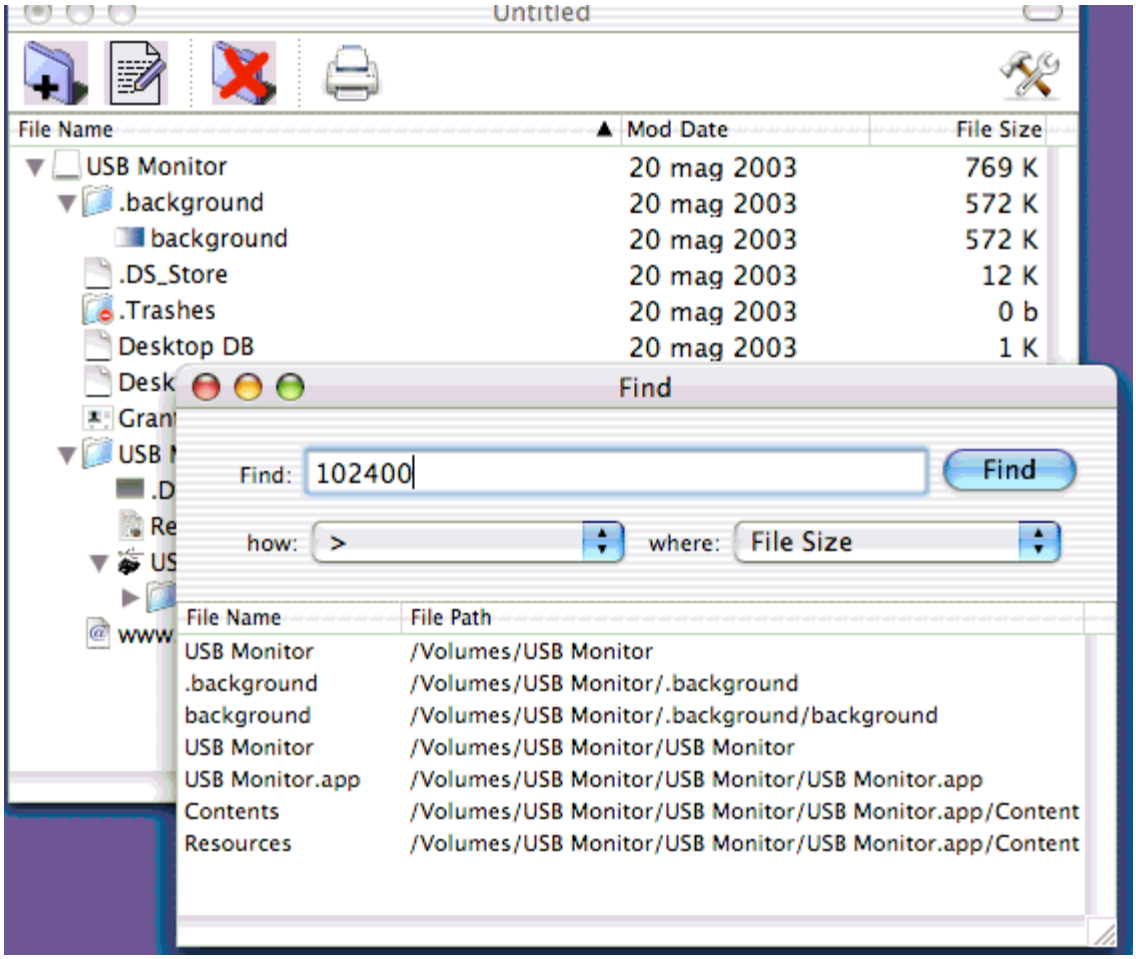
Per ciascuna categoria (a dire il vero, ho trascurato le date...) si considera il criterio di confronto, e si restituisce il valore Vero se il criterio è soddisfatto, Falso altrimenti. Con questo valore di ritorno, il metodo `searchForItem:...` decide se aggiungere o menu l'elemento in esame alla lista dei risultati della ricerca.

La combinazione dei metodi e della funzione sopra descritti esamina l'intero catalogo ed aggiunge al vettore dei risultati tutti gli elementi che soddisfano al criterio scelto. In realtà, ci sono ancora alcune cose da perfezionare (le date, e il fatto che le directory sarebbe bene non considerarle in certe serie di confronti), ma l'impianto mi sembra funzionare.

MOSTRARE I RISULTATI

Avendo fatto molta pratica con le `NSOutlineView` e con le `NSTableView`, predisporre i metodi per la visualizzazione dei risultati della ricerca è un gioco da ragazzi.





Sono da scrivere i soliti tre metodi per fornire i dati, già presenti in forma di vettore all'interno della classe `FindWinCtrl` (che ho provveduto in IB a specificare come sorgente di dati della tabella). Li riporto giusto per completezza, in quanto non c'è nulla di nuovo:

```
- (int)
numberOfRowsInTableView:    (NSTableView *)tableView
{
    return [foundItems count];
}

- (id)
tableView:          (NSTableView *)tableView
    objectValueForTableColumn: (NSTableColumn *)tableColumn
    row:              (int)row
{
    LSFileInfo      * item = [foundItems objectAtIndex: row] ;
    NSString    * colId = [ tableColumn identifier] ;

    // ho al momento due colonne: il nome o l'intero percorso
    if ( [ colId isEqual: COLID_FILENAME ] )
        return ( [ item fileName] );
    return ( [ item fileFullPath] );
}

- (BOOL)
tableView:              (NSTableView *)aTableView
    shouldEditTableColumn: (NSTableColumn *)aTableColumn
    row:              (int)rowIndex
```

```
{
    return NO ;
}

- (void)
windowDidLoad
{
    // un doppio clic su di una riga mostra l'elemento selezionato
    // (risultato di una ricerca) all'interno del documento che lo contiene
    [ resultFld setDoubleAction: @selector( goToItem:) ];
}
```

È da notare solo l'ultimo metodo eseguito al caricamento della finestra, che associa alla tabella una azione da eseguire quando si fa doppio clic su di essa. Lo scopo del metodo `goToItem:` è di evidenziare nella finestra di catalogo l'elemento trovato presente nella lista dei risultati e sul quale si è fatto clic.

...TROVA

Ho impiegato più tempo a sviluppare questa parte di ritrovamento piuttosto che la parte di ricerca precedente. Sembrava una sciocchezza (avevo dopotutto tutte le informazioni che mi servivano), ma poi ho dovuto adottare una soluzione grezza e non so fino a che punto corretta.

Il punto di partenza è il metodo indicato da un doppio clic sulla tabella dei risultati. Come sempre, il metodo svolge una serie di compiti estetici, ma non fa molti passi in avanti. In primo luogo verifica che ci sia una riga selezionata, poi che esista ancora il catalogo sul quale è stata eseguita la ricerca (nel frattempo l'utente potrebbe averlo chiuso). In caso di problemi, mostra un bel dialogo di avvertimento, che la funzione `NSBeginAlertSheet` mette a disposizione.

```
- (IBAction)
goToItem:(id)sender
{
    FileStruct    * item ;
    // verifico che ci sia una riga selezionata
    if ( [ sender selectedRow ] == -1 ) return ;
    // recupero l'elemento selezionato
    item = [ foundItems objectAtIndex: [ sender selectedRow ]];
     // controllo che il documento cui si riferisce la ricerca sia ancora
presente
    if ( ! [ [ NSApp orderedDocuments] containsObject: [self currCatalog] ]
)
    {
        // il documento non c'e' piu', informo l'utente con un alert
        NSBeginAlertSheet(@"Missing Catalog", @"Ok, ok", nil, nil,
            [self window], nil, nil, nil, nil,
            @"The Catalog Document is no longer present") ;
        // non ho altro da fare...
        return ;
    }
      // qui abbiamo il catalogo, ed abbiamo l'item; lo cerchiamo e lo
apriamo
    [[ [self currCatalog] dataSource ] expandDStoItem: item
        inOutlineView: [ [self currCatalog] outlineView ] ];
    // forzo il redisplay della finestra del catalogo
    [ [ [self currCatalog] outlineView ] reloadData ];
```

```
    // metto la finestra del catalogo davanti alle altre
    [ [self currCatalog] showWindows ] ;
}
```

Poi c'è la consueta invocazione di un metodo alla sorgente dati; di seguito, forza il rinfresco della finestra di catalogo e la porta davanti a tutte.

Ora ho un bel problema: l'idea è di rendere evidente l'oggetto selezionato nella finestra di catalogo. Il più delle volte, questo oggetto sarà nascosto, in quanto non necessariamente la vista `NSOutlineView` è espansa fino all'elemento richiesto. Occorre quindi espandere, con l'apposito metodo `expandItem:`, tutti gli elementi dell'alberatura che portano fino all'oggetto trovato.

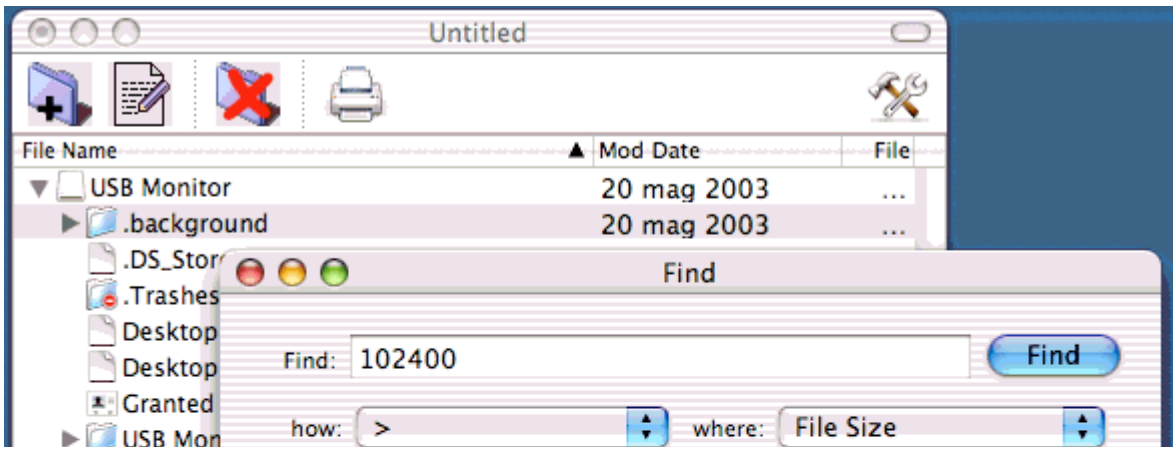
All'inizio pensavo di fare come al solito, con una procedura ricorsiva, ma così non funziona. L'approccio da seguire deve essere diretto: espandere il volume, e poi le directory in ordine di percorso. Purtroppo, non sono riuscito a trovare niente di meglio di una procedura ad hoc, che esamina il percorso completo del file ed espande gli elementi che trova coincidenti. Come al solito, la procedura si svolge in due passi. Il primo è il metodo `expandDStoItem:` sopra invocato:

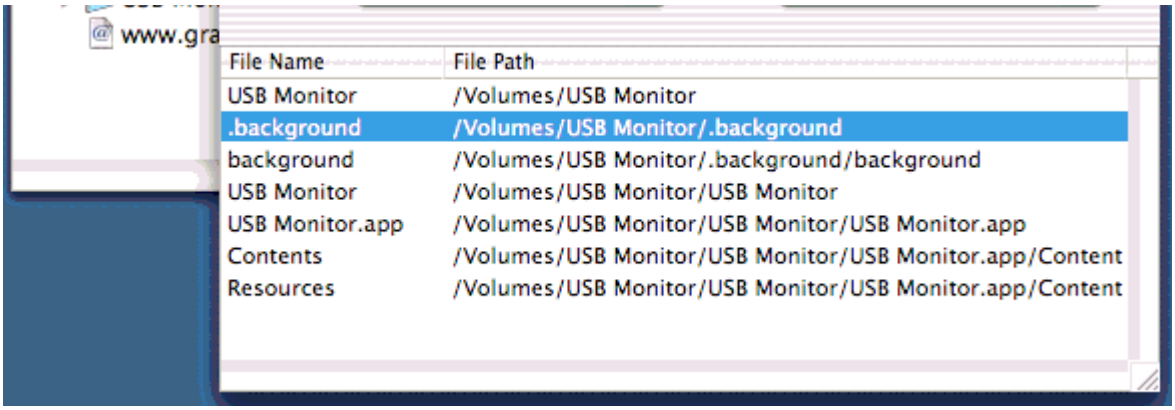
```
- (void)
expandDStoItem: (FileStruct*) expItem
    inOutlineView: (NSOutlineView *) outView
{
    // recupero il path completo dell'elemento obiettivo
    NSString    * fPath = [expItem fileFullPath] ;
    // lo spezzo ordinamente nei vari elementi
          NSEnumerator          *   pathComps   =   [[fPath   pathComponents]
objectEnumerator];
    // piglio anche la lista dei volumi
    NSEnumerator    * listavolumi = [[self startPoint] objectEnumerator];
    FileStruct     * item ;
    // pulisco i primi elementi del percorso completo dell'obiettivo
    NSString    * name = [pathComps nextObject] ;     // /
    name = [pathComps nextObject] ;            // Volume
    name = [pathComps nextObject] ;            // nome volume

    // cerco nella lista dei volumi il volume che mi serve
    while (item = [listavolumi nextObject])
    {
        // quando ho trovato il volume...
        if ( [[item fileName] isEqual: name] )
        {
            int    thisRow = [outView rowForItem: item];
            // dico che questo item va espanso perche' fa parte della
            // catena che porta all'obiettivo
            [ outView expandItem: item ];
            // lo seleziono
            [ outView selectRow: thisRow byExtendingSelection: FALSE ];
            // e lo porto in bella vista
            [ outView scrollRowToVisible: thisRow ];
            // poi proseguo nell'espansione
            [ item expandFStoItem: pathComps inOutlineView: outView] ;
            return ;
        }
    }
}
```

Dopo aver recuperato il path completo dell'elemento da evidenziare, lo divido nei vari nomi parziali che lo compongono (la classe `NSString` fornisce vari metodi appositi per la manipolazione di path). Il primo passo termina cercando all'interno della lista dei volumi contenuti nel catalogo il volume interessato. Una volta trovato il volume, espando l'elemento, lo seleziono e poi porto la selezione all'interno della parte visibile della `NSOulineView`; poi, proseguo nell'espansione con il secondo passo. Da notare come, una volta trovato l'elemento, non vengano esaminati i successivi: è questa la funzione del return subito dopo la chiamata del metodo `expandFStoItem`:

```
- (void)
expandFStoItem: (NSEnumerator*) pathComp
    inOutlineView: (NSOutlineView *) outView
{
    // piglio l'elenco dei file contenuti
    NSEnumerator    * enumerator = [[self fileList] objectEnumerator];
    // questo e' il (pezzo di) nome dell'elemento obiettivo
    NSString    * name = [ pathComp nextObject ];
    FileStruct     * item ;
    // esamino tutti gli elementi presenti
    while (item = [enumerator nextObject])
    {
        // se il nome coincide con quello cercato
        if ( [[item fileName] isEqual: name] )
        {
            int    thisRow = [outView rowForItem: item];
            // dico che questo item va espanso perche' fa parte della
            // catena che porta all'obiettivo
            [ outView expandItem: item ];
            // lo seleziono
            [ outView selectRow: thisRow byExtendingSelection: FALSE ];
            // e lo porto in bella vista
            [ outView scrollRowToVisible: thisRow ];
            // poi proseguo nell'espansione
            [ item expandFStoItem: pathComp inOutlineView: outView] ;
            // non devo esplorare gli altri elementi...
            return ;
        }
    }
}
```

Il secondo passo è piuttosto simile: si cerca fra gli elementi della directory corrente quello che corrisponde al pezzo di percorso cercato; una volta trovato, si espande l'elemento, lo si seleziona e lo si porta in evidenza. Ancora una volta non si prosegue con gli elementi della directory una volta trovato quello giusto.





Il meccanismo funziona, anche se espande una volta di troppo. C'è infatti il problema di terminare l'espansione al posto giusto. Con i due metodi sopra descritti, il termine dell'espansione si ha quanto, all'interno di `expandFStoItem`, l'istruzione che assegna un valore alla variabile name restituisce `nil` (fine del percorso). Tuttavia, questo avviene troppo tardi, ad un livello inferiore a quello dell'elemento trovato (per cui questo è stato già espanso).

Avevo quasi rinunciato a risolvere questa imperfezione quando mi è venuta l'ispirazione: basta aggiungere come ultima istruzione del metodo `expandDStoItem:` il collassamento dell'elemento selezionato!

```
    [ item expandFStoItem: pathComps inOutlineView: outView] ;
    [ outView collapseItem: expItem ];
    return ;
```

In effetti, al termine della discesa dell'alberatura generata da `expandFStoItem:`, l'elemento da evidenziare, appunto `expItem`, si trova bellamente in vista all'interno della `NSOutlineView` (cosa non sempre vera all'inizio, prima di effettuare l'evidenziazione del risultato), per cui il metodo `collapseItem:` trova sicuramente questo elemento, e chiude la visualizzazione degli elementi interni.

Capitolo 032 - Barra a Dritta

Questo capitolo riprende il discorso interrotto con una pagina leggera per ricominciare senza troppi traumi.

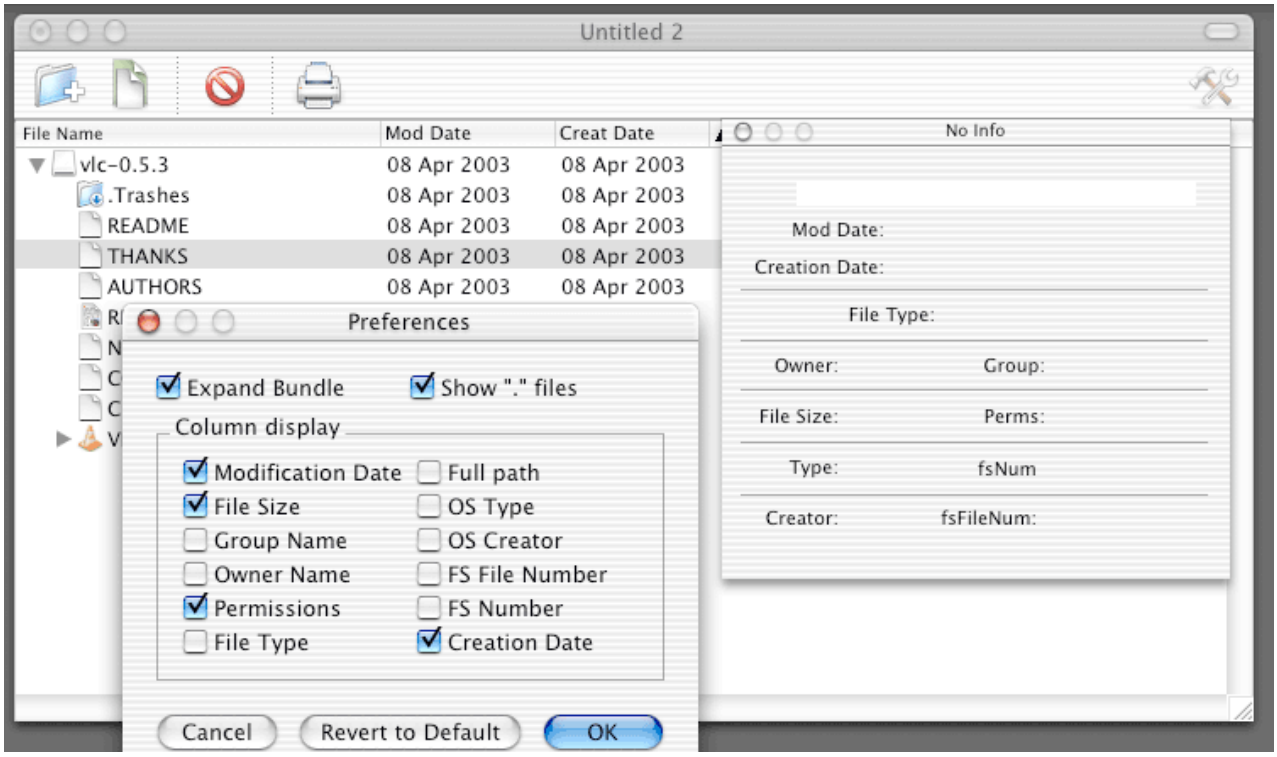
Sorgenti: Documentazione Apple.

Primo inserimento: 5 giugno 2003

NUOVA DOCUMENTAZIONE

Parto dall'aggiornamento dei Developer Tools del Dicembre 2002. Più che le funzioni aggiunte, le possibilità esotiche di compilazione e quant'altro, mi interessa verificare la documentazione in linea. In effetti vedo che ci sono molte informazioni aggiuntive (magari erano anche nei rilasci precedenti, ma le viste solo ora...). Comincio quindi da lì, per scoprire finalmente che, tra le informazioni associate ad un file, compare anche la data di creazione (e non solo quella di ultima modifica). È giocoforza aggiungere anche questa informazione a tutto il mio castello di oggetti ed interfaccia.

Sicuramente dimentico qualche modifica: aggiungo una variabile d'istanza all'oggetto `LSFileInfo`; di conseguenza, i metodi accessor, qualche istruzione per salvare il dato su file e recuperarlo (ancora una volta, il formato del file catalogo cambia, ma è storia già vista); ci sono da cambiare alcuni file nib che contengono l'interfaccia, sicuramente la finestra della Preferenze (c'è appunto da aggiungere la visualizzazione o meno della colonna con la data di creazione); ma allora c'è da cambiare anche la gestione delle preferenze, della visualizzazione nella tabella; c'è infine (forse...) la finestra per le ricerche, dove adesso c'è da aggiungere nell'interfaccia una voce di menu pop-up, e poi anche la gestione delle ricerche.



Insomma, molte piccole modifiche.

Ne ho allora approfittato anche per eliminare codice obsoleto, ovvero ho eliminato dal file `DjZeroUtils.h` le define di compilazione (ne avevo parlato in precedenza), con la conseguente rimozione di tutto il codice interessato.

VOLUMI

Sempre frugando nella documentazione, trovo che l'oggetto `NSFileManager` possiede un metodo per recuperare alcune informazioni relative ai volumi nel loro complesso.

Ora, in Unix e Mac OS X, un volume (disco) è considerato come un file. In effetti, ho sfruttato tale fatto in modo naturale; con l'apposita finestra, scelgo un volume da catalogare. Una volta ottenuto il path, lo considero come un file, ne leggo le informazioni relative, e da qui, essendo a tutti gli effetti una directory, procedo alla costruzione dell'albero dei file contenuti nel volume (di più: all'inizio dei tempi, questo programma ha cominciato catalogando file e directory piuttosto che volumi).

Ci sono però alcune informazioni relative ad un volume che non sono disponibili (né pensabili) se si considera il volume come un file. Ad esempio (cosa che mi rattristava particolarmente), un volume ha una dimensione fisica (per un cd, ad 700 Mega) ben diversa dallo spazio occupato (la dimensione ricorsivamente calcolata sommando lo spazio occupato da tutti i suoi file).

Ecco quindi che è disponibile un metodo solo per i file che sono volumi:

```
attributi = [[NSFileManager defaultManager]fileSystemAttributesAtPath: volume];
```

che in piena similitudine coll'ormai noto

```
attributi = [[NSFileManager defaultManager]fileAttributesAtPath: filename ];
```

produce una serie di informazioni utili: la dimensione del volume, che si spera essere in byte (data l'ampia varietà di formati di volumi letti da Mac Os X, c'è da sperarsi, ma non è garantito) come l'attributo `NSFileSystemSize`; all'opposto, lo spazio libero sul volume `NSFileSystemFreeSize`, nella stesse unità di misura. Sottraendo i due numeri, dovrei avere lo spazio occupato sul volume, che dovrebbe coincidere con quello calcolato sommando le dimensioni di tutti i file (ma c'è il problema della resource fork, che non è considerata...). Poi ci sono due numeri, che dicono quanti *nodi* sono disponibili sul volume, e quanti di questi sono disponibili (`NSFileSystemNodes` e `NSFileSystemFreeNodes`). Cosa siano con precisione i nodi, non lo so, e lascio volentieri indietro ogni spiegazione. Infine, il numero `NSFileSystemNumber` dovrebbe coincidere col sempre misterioso campo `fsNum` delle informazioni relative ad un file (un codice identificativo del volume, suppongo, ma ignoro qualsiasi criterio di produzione di tale numero).

UNA NUOVA CLASSE

Per mantenere le informazioni relative ai volumi, decido di costruire una nuova classe. Dal momento che un volume è un file (`LSFileInfo`) che è una directory (`FileStruct`) con qualcosa di speciale, la cosa da fare è una sottoclasse di `FileStruct`. Battezzata subito la classe `VolInfo`, ecco l'interfaccia:

```
@interface VolInfo : FileStruct {
    // dimensione del volume, forse in byte
    unsigned long long    volSize ;
    // dimensione dello spazio libero, forse in byte
    unsigned long long    volFreeSize ;
    // numero di nodi presenti, qualunque cosa siano
    unsigned long long    volSystemNodes ;
    // numero di nodi liberi, qualunque cosa siano
    unsigned long long    volSystemFreeNodes ;
    // numero del Fyle system
    unsigned long long    volFileSystemNumber ;
}
```

completata dalla solita pletora di metodi accessor (più facili del solito, dal momento che si tratta di grandezze scalari e non di oggetti).

```
- (unsigned long long)volSize
    { return volSize; }
- (void)setVolSize :(unsigned long long)xx
    { volSize = xx ; }
```

Al momento, la raccolta delle informazioni è l'unico scopo dell'oggetto. È tuttavia interessante l'uso che se ne fa, nella parziale riscrittura del metodo `performAddFilesModal:`. Il codice definitivo è diverso da quello seguente, ma per aspetti che spiegherò poi:

```
- (void)
performAddFilesModal: ( NSArray *)volList
{
    NSModalSession     session ;
    NSEnumerator     * enumerator = [volList objectEnumerator];
    NSString    * volPath ;
    NSDictionary *fsattrs ;
    unsigned long long    totalSpace, freeSpace, usedspace ;

    volPath = [enumerator nextObject] ;
    // mostro la finestra di attesa con apposita stringa
    [ WaitPanCtrl showHide: TRUE ];
    [ WaitPanCtrl setText: [NSString stringWithFormat: @"Cataloging %@",
        [[ NSFileManager defaultManager] displayNameAtPath: volPath] ]];
    // faccio partire l'animazione della barra del barbiere
    [ WaitPanCtrl animate: self ];
    // comincio una sessione modale per l'intera applicazione
     session = [NSApp beginModalSessionForWindow: [ [ WaitPanCtrl sharedProgress] window ]
];
    // eseguo la sessione
    [NSApp runModalSession:session] ;
    while ( volPath )
    {
        VolInfo     * fInfo ;
        fsattrs = [[NSFileManager defaultManager] fileSystemAttributesAtPath: volPath];
        totalSpace = [[fsattrs objectForKey:NSFileSystemSize] unsignedLongLongValue] ;
        freeSpace = [[fsattrs objectForKey:NSFileSystemFreeSize] unsignedLongLongValue] ;
        usedspace = totalSpace - freeSpace ;
        // costruisco l'alberatura dei file a partire dalla scelta
        fInfo = [[ VolInfo alloc ] initTreeFromPath: volPath ];
        // metto a posto le altre informazioni
        [ fInfo setFileSize: totalSpace ];
```

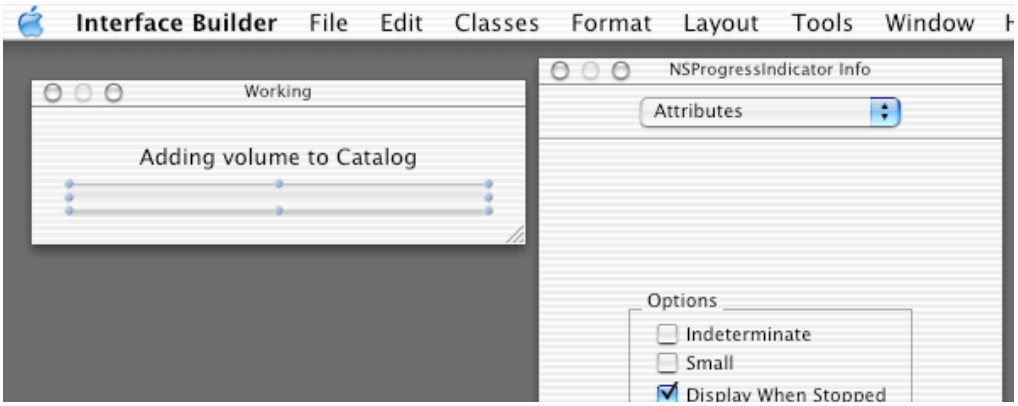
```
        [ fInfo setVolSize: totalSpace ];
        [ fInfo setVolFreeSize: freeSpace ];
                    [  fInfo  setVolSystemNodes:  [[fsattrs  objectForKey:NSFileSystemNodes]
unsignedLongLongValue] ];
              [ fInfo setVolSystemFreeNodes: [[fsattrs objectForKey:NSFileSystemFreeNodes]
unsignedLongLongValue] ];
                [  fInfo  setVolFileSystemNumber:  [[fsattrs  objectForKey:NSFileSystemNumber]
unsignedLongLongValue] ];
        // aggiungo la cosa al catalogo
        [ dataSource addFileEntry: fInfo ];
        volPath = [enumerator nextObject] ;
        if ( volPath == nil ) break ;
        // mostro la finestra di attesa con apposita stringa
        [ WaitPanCtrl setText: [NSString stringWithFormat: @"Cataloging %@",
            [[ NSFileManager defaultManager] displayNameAtPath: volPath] ]];
        // faccio partire l'animazione della barra del barbiere
        [ WaitPanCtrl animate: self ];
    }
    // finisco la sessione
    [NSApp endModalSession: session];
    // nascondo la finestra di attesa
    [ WaitPanCtrl showHide: FALSE ];
}
```

Il metodo esamina tutti i volumi contenuti nel vettore passato come parametro; di ciascuno preleva gli attributi notevoli, e li conserva nelle giuste variabili. È da notare come la costruzione dell'alberatura della directory avviene come in precedenza. Questa volta il metodo inizializzatore `initTreeFromPath:` invece che essere applicato ad un oggetto `FileStruct` lo è ad un oggetto `VolInfo`, ma funziona lo stesso.

In più dico che la dimensione del volume non è quella calcolata sommando le dimensioni di tutti i file, ma proprio la dimensione fisica del volume (mi pare una cosa più sensata).

ATTESE PIU' MISURATE

Uno dei motivi di cruccio per cui ho cercato spasmodicamente la dimensione complessiva di un volume prima di esaminare tutti i file è legata alla finestra di attesa `WaitPanCtrl`. Fino ad ora la barra del barbiere si limita a mostrare attività in corso, ma non riporta alcuna indicazione sulla percentuale di lavoro compiuto. Adesso però, conoscendo la dimensione totale del volume e la quantità di file già catalogati, è possibile lavorare in altro modo. Intendo modificare la finestra in modo che mostri una barra che avanza progressivamente con il procedere delle operazioni; in particolare, la barra esprime percentualmente la dimensione complessiva dei file catalogati nei confronti del totale (ci si mette più a spiegarlo che a farlo).



In primo luogo, occorre modificare la finestra. Sull'interfaccia, con IB, si tratta solo di spuntare la voce `intedeterminate` dalle caratteristiche della barra. Con questa operazione diventano significative due altre caratteristiche, il range minimo ed il range massimo, che lascio rispettivamente a zero ed uno.

Scrivo poi un po' di metodi della classe `WaitPanCtrl` per manipolare opportunamente la barra. Ci sono tre metodi di classe:

```
+ (void) setMaxValue: (double) maxval;
+ (void) setCurrentValue: (double) currval;
+ (double) getCurrentValue ;
```

per permettere la gestione tramite istanza condivisa, cui corrispondono nascostamente altri metodi più efficaci:

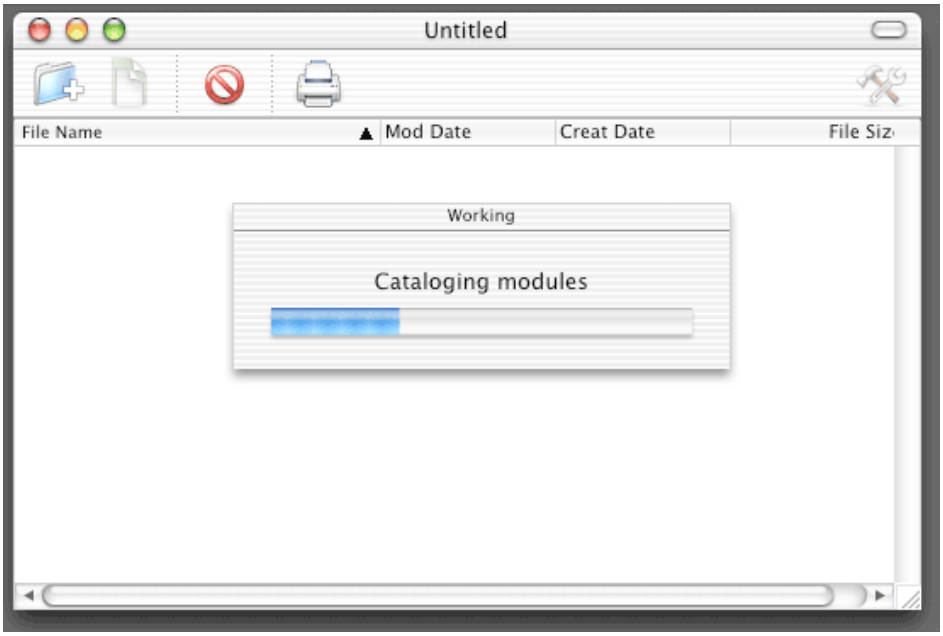
```
- (void) localSetMaxValue: (double) maxval;
- (void) localSetDoubleValue: (double) currval;
- (double) localDoubleValue;
```

La loro realizzazione è talmente semplice da non meritare commenti:

```
+ (void) setMaxValue: (double) maxval
{
    [ [ WaitPanCtrl sharedProgress] localSetMaxValue: maxval ];
}

+ (void) setCurrentValue: (double) currval
{
    [ [ WaitPanCtrl sharedProgress] localSetDoubleValue: currval ];
}

+ (double) getCurrentValue
{
    return( [[ WaitPanCtrl sharedProgress] localDoubleValue] );
}

- (void) localSetMaxValue: (double) maxval
{
    [ progressBar setMaxValue: maxval ];
    maxValue = maxval ;
}

- (void) localSetDoubleValue: (double) currval
{
    [ progressBar setDoubleValue: currval ];
    currValue = currval ;
    [ progressBar displayIfNeeded ];
}
```

```
- (double) localDoubleValue
{
    return( currValue );
}
```

Dotato di questo armamentario, mi accorgo che il metodo `performAddFilesModal` va un po' stravolto per tenere conto delle mutate esigenze. Infatti, finora, quando si deve aggiungere al catalogo una lista di volumi, le cose funzionavano in questo modo (vedi il codice riportato in precedenza):



Mostro la finestra di attesa, apro la sessione modale e poi esamino ordinatamente tutti i volumi. Man mano che trovo un file, faccio fare un giro alla barra.

Adesso le cose cambiano. La finestra con la barra è re-inizializzata ad ogni cambio di volume, pur rimanendo attiva la sessione modale. Quindi la procedura diventa:

Mostro la finestra di attesa, apro la sessione modale e poi esamino ordinatamente tutti i volumi. Ogni volta che considero un nuovo volume, predispongo la barra; man mano che trovo un file, aggiorno la barra considerando la percentuale di file già esaminati.

A questo punto il metodo diventa

```
- (void)
performAddFilesModal: ( NSArray *)volList
{
    NSModalSession     session ;
    NSEnumerator     * enumerator = [volList objectEnumerator];
    NSString    * volPath ;
    NSDictionary *fsattrs ;
    unsigned long long    totalSpace, freeSpace, usedspace ;

    [ WaitPanCtrl showHide: TRUE ];

    volPath = [enumerator nextObject] ;
    // comincio una sessione modale per l'intera applicazione
     session = [NSApp beginModalSessionForWindow: [ [ WaitPanCtrl sharedProgress] window ]
];
    // eseguo la sessione
```

```
    [NSApp runModalSession:session] ;
    while ( volPath )
    {
        VolInfo     * fInfo ;
        fsattrs = [[NSFileManager defaultManager] fileSystemAttributesAtPath: volPath];
        totalSpace = [[fsattrs objectForKey:NSFileSystemSize] unsignedLongLongValue] ;
        freeSpace = [[fsattrs objectForKey:NSFileSystemFreeSize] unsignedLongLongValue] ;
        usedspace = totalSpace - freeSpace ;
        [ WaitPanCtrl setMaxValue: usedspace ];
        [ WaitPanCtrl setCurrentValue: 0 ];
        // mostro la finestra di attesa con apposita stringa
        [ WaitPanCtrl setText: [NSString stringWithFormat: @"Cataloging %@",
            [[ NSFileManager defaultManager] displayNameAtPath: volPath] ]];
        // costruisco l'alberatura dei file a partire dalla scelta
        fInfo = [[ VolInfo alloc ] initTreeFromPath: volPath ];
        // metto a posto le altre informazioni
        [ fInfo setFileSize: totalSpace ];
        [ fInfo setVolSize: totalSpace ];
        [ fInfo setVolFreeSize: freeSpace ];
                [  fInfo  setVolSystemNodes:  [[fsattrs  objectForKey:NSFileSystemNodes]
unsignedLongLongValue] ];
             [  fInfo  setVolSystemFreeNodes:  [[fsattrs  objectForKey:NSFileSystemFreeNodes]
unsignedLongLongValue] ];
              [  fInfo  setVolFileSystemNumber:  [[fsattrs  objectForKey:NSFileSystemNumber]
unsignedLongLongValue] ];
        // aggiungo la cosa al catalogo
        [ dataSource addFileEntry: fInfo ];
        volPath = [enumerator nextObject] ;
        if ( volPath == nil ) break ;
    }
    // finisco la sessione
    [NSApp endModalSession: session];
    // nascondo la finestra di attesa
    [ WaitPanCtrl showHide: FALSE ];
}
```

Devo anche modificare il metodo `initWithPath` della classe `LSFileInfo` in modo che rifletta il nuovo aspetto della barra: al posto della istruzione

```
[ WaitPanCtrl animate: self ];
```

presente all'inizio, inserisco l'istruzione

```
[ WaitPanCtrl setCurrentValue: ([ WaitPanCtrl getCurrentValue ] + [self fileSize])] ;
```

verso la fine (dopo che ho aggiornato la variabile di istanza `fileSize`).

La barra funziona in questo modo: ho impostato il valore massimo, all'interno del metodo performAddFilesModal, pari allo spazio occupato da tutti i file all'interno del volume. Nello stesso momento ho anche assegnato zero al valore corrente della barra. Con l'istruzione precedente assegno un nuovo valore al valore corrente, pigliando il valore precedente ed aggiungendo la dimensione del file appena esaminato. La barra (l'oggetto dell'interfaccia, i metodi messi a disposizione dal kit) si occupa in autonomia di calcolare in che percentuale il valore corrente si pone nei confronti del massimo, e disegna la barra più o meno piena.

Se le cose sono state fatte correttamente, alla fine, quando sono stati esaminati tutti i file, le due dimensioni dovrebbero coincidere, e la barra essere totalmente piena (anche se tale situazione rimane per un solo istante,che poi la finestra di attesa scompare).

UN'ALTRA DIFFERENZA E' POSSIBILE

In tutta questa (mia) confusione, è certo che mi sono dimenticato qualche altra modifica apportata al codice. Piuttosto che avermene a male, si utilizza la simpatica applicazione `FileMerge` (la trovate nella directory `/Developer/Applications`) che permette, tra le altre cose, di confrontare due intere directory (e sottodirectory). Operate il confronto fra le due directory che contengono tutti i file del progetto, quella relativa al precedente capitolo e quella del capitolo corrente, ed il gioco è fatto.

Capitolo 033 - Un pezzo di Carbon

Mi tolgo un sassolino dalla scarpa, ma sono costretto ad utilizzare Carbon. Nessuna paura, è tutto molto semplice.

Documentazione Apple (Carbon e Cocoa).

Primo inserimento: 25 giugno 2003

Dimensione e Risorse

Giocando con la barra dello scorso capitolo, qualcuno si sarà accorto che l'avanzamento lavori non è proprio perfetto. Il problema è molto più evidente in volumi che contengono molti file ed applicazioni di vecchio tipo. Ritorna, ormai è una sorta di incubo, il problema della resource fork.

Ricordo per i più distratti. I file all'interno del Macintosh nascono bifidi. Il loro contenuto è diviso in due parti: la *data fork*, dove sono contenuti i dati, e la *resource fork*, dove sono contenute le risorse. Questa divisione si riflette abbastanza bene nei file di dati (costituiti spesso in prevalenze dalla data fork) e nei file che contengono applicazioni (costituiti massimamente da risorse). Con Mac OS X invece, i file sono solitamente costituiti dalla sola data fork, dal momento che le risorse sono trattate in altro modo (i bundle dell'applicazione, ad esempio). Sto semplificando, mi rendo conto, ma è solo per tirarla breve. Quando con Cocoa si recuperano le informazioni relative ad un file, si ottiene la dimensione della sola data fork. Quindi, sono persi tutti i numeri relativi alla resorce fork. Ciò spiega le discrepanze tra le dimensioni riportate dal terminale, dalla finestra delle info del Finder, eccetera.

È giunto allora il momento di risolvere la questione una volta per tutte. Non c'è altra possibilità che ricorrere a Carbon. Ricordo (ancora una volta) per i più distratti. Sopra il sistema operativo dal cuore Unix, sono disponibili diversi ambienti all'interno dei quali è possibile costruire applicazioni per Macintosh. Uno di questi è Cocoa, argomento di questo sito. Un altro è appunto Carbon. Carbon è costituito in pratica da una serie di funzioni (una libreria, direbbe qualcuno) utilizzabili da ogni linguaggio di programmazione per scrivere applicazioni Macintosh. Le funzioni sono quelle ereditate (ma con la faccia ripulita per l'occasione) dalle precedenti versioni del sistema operativo (fino alla 9 compresa). Chi, come me molti anni fa, si dilettava di programmazione su Macintosh, riconosce molti vecchi amici e può riutilizzare (con debita attenzione) i vecchi volumi di Inside Macintosh.

Risorse con Carbon

La sta tirando per le lunghe, in quanto in realtà le cose sono piuttosto semplici. Cominciamo col recuperare un po' di informazioni da un file; mi interessa in pratica solo la dimensione della resource fork, visto che le altre informazioni utili sono recuperate con il metodo `fileAttributesAtPath:` della classe `NSFileManager`.

Il primo passo è specificare quale file mi interessa. Con `NSFileManager` un file è

identificato dal suo path completo. Con Carbon, si usa tutt'altro: un `File System Representation`, o `FSRef`, una struttura dati che identifica in maniera completa ed equivalente un file. Piuttosto che perdere tempo a spiegare come sia fatta questa struttura, conviene utilizzare una funzione Carbon adatta a nascondere ogni dettaglio inutile. La funzione `FSPathMakeRef` richiede in ingresso una stringa che contiene il percorso completo del file secondo la rappresentazione del file system considerato, e restituisce due dati: la struttura `FSRef` completa delle informazioni necessarie, ed un valore che dice se si tratta di una directory o meno. La funzione ritorna un codice di errore, che vale la costante `noErr` se tutto è andato bene.

Con la struttura `FSRef` ottenuta è a questo punto facile recuperare le informazioni che mi servono: mi giovo della funzione `FSGetCatalogInfo`, che fa al caso mio.

La funzione sembra avere un brutto aspetto ad una prima occhiata, ma poi in realtà è un pezzo di pane. Gli si deve fornire una `FSRef` per specificare un file, ed un codice che individua quali informazioni recuperare. Da tutto ciò produce una notevole quantità di dati; in primo luogo, raccolte in una struttura `FSCatalogInfo` i dati richiesti, e poi, opzionalmente, il nome del file e tante altre cose che in questa applicazione non ci interessano. Mi sto dilungando troppo; ecco quindi il codice da aggiungere al metodo `initWithPath:` della classe `LSFileInfo`.

```
// se il file e' effettivamente un vero file
if ( [[self fileType] isEqualToString: NSFileTypeRegular ])
{
    OSStatus         status ;
    FSRef             myFSRef;
    FSCatalogInfo        catinfo ;
    unsigned long long    tmpSize ;
    float            d1, d2, d3 ;
    // mi faccio qualche scrupolo in piu'
    // potrebbe esserci una resource fork
    // costruisco una FSRef
      status = FSPathMakeRef ([aFile fileSystemRepresentation], &myFSRef,
NULL);
    // e poi indago con Carbon le dimensioni della resource
    if (status == noErr)
        status = FSGetCatalogInfo (&myFSRef, kFSCatInfoRsrcSizes,
            & catinfo, NULL, NULL, NULL);
    tmpSize = [self fileSize] + catinfo.rsrcLogicalSize ;
    [ self setFileSize: tmpSize ];
}
```

Da notare che tutte queste operazioni sono svolte solo se il file in esame è un vero e proprio file, e non qualcos'altro (una directory, o altre diavolerie tipiche di Unix). Ecco quindi che chiamo le due funzioni citate prima, per ottenere solo il campo `rsrcLogicalSize` della struttura dati `FSCatalogInfo` prodotta. Piuttosto che mantenere separata questa dimensione (che andrebbe gestita, nella struttura dati, nelle preferenze, nella finestra delle info, eccetera), ho preferito aggiungerla alla dimensione della data fork ed ottenere un dato globale della dimensione del file.

CONTO I FILE

A questo punto sembrerebbe tutto a posto; tuttavia, avendo cominciato con Carbon,

tanto vale andare avanti, e fare qualcosa di più corretto con la barra di avanzamento lavori. Utilizzare la dimensione dei file come criterio sullo stato di catalogazione non è proprio la cosa più giusta da farsi: se ci sono tanti piccoli file ed un solo grande file, la barra non è equilibrata. La cosa migliore sarebbe lavorare col numero di file. Eppure, con Cocoa, il numero di file presenti su di un volume non è disponibile all'inizio delle operazioni. Invece, con Carbon, tale dato è disponibile: basta utilizzare la funzione giusta, `FSGetVolumeInfo`. Anche qui, sono necessarie due informazioni in ingresso: la specifica del volume e quella delle informazioni desiderate. La funzione produce molte informazioni, ma a me interessa un solo campo della struttura `FSVolumeInfo`, quello che appunto riporta il numero di file contenuti nel volume.

Il problema nasce nella specifica del volume, che deve essere data attraverso un codice identificativo (il `volume referente number`). Dopo una rapida (si fa per dire) indagine, scopro che tale codice è possibile ricavarlo come un sottoprodotto della funzione `FSCatalogInfo`, in uno dei campi delle strutture in precedenza trascurate. Ancora una volta, mi disinteresso dei dettagli e procedo spedito secondo i miei scopi. Modifico il metodo `performAddFilesModal:` di `CatalogDoc`.

```
while ( volPath )
{
    VolInfo     * fInfo ;
    // il volPath ho il nome completo del volume
    FSSpec myFSSpec ;
    FSRef myFSRef;
    FSVolumeInfo myInfo;
    FSCatalogInfo    catinfo ;
    OSStatus status = FSPathMakeRef ([volPath fileSystemRepresentation],
            &myFSRef, NULL);
    if (status == noErr)
          status = FSGetCatalogInfo (&myFSRef, kFSCatInfoNone, & catinfo,
NULL,
                             & myFSSpec, NULL);
    // adesso in myFSSpec ho il mio FSSpec
             status   =   FSGetVolumeInfo   (   myFSSpec.vRefNum,   0,   NULL,
kFSVolInfoFileCount, & myInfo,
        NULL, NULL );
     fsattrs = [[NSFileManager defaultManager] fileSystemAttributesAtPath:
volPath];
              totalSpace    =    [[fsattrs    objectForKey:NSFileSystemSize]
unsignedLongLongValue] ;
             freeSpace    =    [[fsattrs    objectForKey:NSFileSystemFreeSize]
unsignedLongLongValue] ;
    usedspace = totalSpace - freeSpace ;
    [ WaitPanCtrl setMaxValue: myInfo.fileCount ];
    [ WaitPanCtrl setCurrentValue: 0 ];
    // eccetera
}
```

C'è una catena di tre funzioni Carbon il cui unico scopo è di produrre la variabile `myInfo.fileCount`; questa è utilizzata direttamente per dare il valore massimo della barra di scorrimento.

Non rimane altro che effettuare una modifica all'interno di `initWithPath:` della classe `LSFileInfo`:

```
    [ WaitPanCtrl setCurrentValue: ([ WaitPanCtrl getCurrentValue ] + 1)] ;
```

Così facendo è possibile gestire in maniera un po' più pulita la barra che indica lo stato avanzamento lavori.

Capitolo 034 - Prima della pioggia

Dopo lungo tempo, ripiglio in mano Macocoa, e mi trovo a lavorare con Xcode e Panther. Prima che si scateni il diluvio, metto a posto un problema, capisco meglio una cosa, ed aggiungo una piccola cosa. Tuttavia, come si vedrà, la mia ignoranza rimane alta.

Documentazione Apple

Primo inserimento: 27 dicembre 2003

XCode

XCode è il nuovo ambiente di programmazione di Apple, che sostituisce Project Builder. Nonostante le roboanti notizie stampa, non è che all'apparenza sia cambiato molto. C'è però da dire che non ricordo molto i dettagli delle versioni precedenti. È quindi possibile che alcune funzioni e caratteristiche qui date per scontate non siano in realtà presenti sulle versioni precedenti. Non leggo neppure le note di rilascio, ma, nella migliore tradizione degli utenti Macintosh, comincio subito a lavorarci, lasciandomi guidare dall'intuizione. E intendo continuare così, dal momento che finora non ho ancora letto una riga di documentazione su XCode e non ho ancora affrontato grossi problemi.

Ovviamente, non appena apro il mio vecchio progetto CDCat, Xcode mi avvisa che intende convertirlo nel nuovo formato, eccetera. Lo lascio fare. C'è una caratteristica piuttosto interessante; l'indicizzazione del codice, che ha subito profonde modifiche, tanto che adesso propone anche la caratteristica **code completion**, che spesso viaggia assieme. Detto in parole povere, la code completion è un meccanismo che, mentre state scrivendo del codice, piglia possesso della finestra dell'editor e vi propone il completamento di ciò che state scrivendo; ad esempio, se cominciate a scrivere il nome di un metodo, l'editor vi propone in una bella lista ordinata i vari metodi che potrebbero rispondere al bisogno. Ci si mette di più a spiegarlo che a vederlo in azione. Io l'ho disattivato, per non infierire sul mio povero computer (a corto di megahertz) ed anche perché sono abituato a programmare in altro modo (paradigma copia-incolla). Se avessi una macchina più potente, troverei la cosa molto utile (in altri momenti, su altri sistemi operativi, il meccanismo di code completion è inestimabile).

Dicevo che ho importato il vecchio progetto CDCat nel nuovo XCode. Compilo tutto quanto, nessun problema. Faccio partire l'applicazione, e ci sono, ovviamente, devastanti problemi. Cosa è successo? Chi lo sa. Approfondisco l'esecuzione del codice utilizzando il nuovo debugger (che è sempre gdb con una bella faccia, un po' rifatto rispetto alla vecchia versione, ma è sempre il buon vecchio gdb).

Scopro dopo lungo penare che il problema risiede all'interno della procedura `reduceImageToIcon` (file `djZeroUtils.m`). Ad un certo punto, eseguo la chiamata `removeRepresentation:` che produce una morte spettacolare. Piuttosto che stare lì ad indagare oltre (sono pigro), decido di riscriverlo completamente, in maniera migliore. Ricordo che la procedura cerca di ridurre le dimensioni di un oggetto `NSImage`

rimuovendo tutte le rappresentazioni dell'immagine che non interessano ai miei scopi (in pratica intende mantenere solo quella più piccola). Trasformo la procedura in una funzione, che cerca la rappresentazione più piccola, e con questa produce un nuovo oggetto `NSImage` ai minimi termini.

La funzione è in realtà molto somigliante alla procedura originale. Ancora una volta, c'è una ricerca della rappresentazione minima. Questa rappresentazione è utilizzata per riempire un oggetto `NSImage`, che è restituito dalla funzione.

```
NSImage *
reduceImageToIcon( NSImage * img )
{
    NSImage        * reducedImg ;
    NSSize        smallIconSize ;
    int        minSize, minImgIndex;
    unsigned int    i ;
    NSImageRep     * imageRep ;
    // recupero la lista delle rappresentazioni
    NSArray        * tmp = [ img representations ] ;
    // per ora la rappresentazione minima e' la prima
    minImgIndex = 0 ;
    imageRep = [ tmp objectAtIndex: minImgIndex ];
    minSize = [ imageRep pixelsHigh ];
    // poi guardo tutte le altre rappresentazioni
    for ( i = 1 ; i < [ tmp count ] ; i ++ )
    {
        int        hh ;
        imageRep = [ tmp objectAtIndex: i ];
        // se la dimensione della rappresentazione corrente...
        hh = [ imageRep pixelsHigh ];
        // ...e' maggiore
        if ( hh < minSize )
        {
            // ho trovato una nuova rappresentazione minima
            // me la segno
            minImgIndex = i ;
            minSize = hh ;
        }
    }
    reducedImg = [ [ [NSImage alloc] init ] autorelease ];
    // arrivato qui, ho trovato al rappresentazione piu' piccola
    [ reducedImg addRepresentation: [ tmp objectAtIndex: minImgIndex ]];
    // dico di scalarla quando sara' ridimensionata
    [ reducedImg setScalesWhenResized: TRUE ];
    // perche' adesso la ridimensiono a 16x16
    smallIconSize.width = smallIconSize.height = 16 ;
    // ecco che la ridimensiono
    [ reducedImg setSize: smallIconSize];
    return (reducedImg);
}
```

Con questa modifica, l'applicazione CDCat continua a funzionare.

FINESTRA MODALE

C'è un comportamento dell'applicazione che non è particolarmente corretto (anzi, è sbagliato). Quando si aggiunge un volume al catalogo compare una finestra di attesa

(`WaitPanCtrl`) che mostra il progredire delle operazioni. Questa finestra è modale per l'applicazione: quando c'è questa finestra in primo piano, nessuna altra finestra dell'applicazione è in grado di funzionare (anzi, se fate clic fuori della finestra ottenere un beep di sistema). Ciò è ragionevole dal momento che l'applicazione non sarebbe in grado di compiere altre operazioni. È però perfettamente lecito passare ad un'altra applicazione, e nell'attesa fare qualcos'altro. In effetti, facendo clic altrove, un'altra finestra si porta davanti, e si può lavorare con l'applicazione cui appartiene questa finestra. Tuttavia, la finestra di attesa rimane proditoriamente in primo piano. Non impedisce il lavoro con le altre applicazioni, ma è li fastidiosamente in primo piano a coprire spazio del monitor. Ovviamente, così non va.

Non sapendo cosa fare, leggo la documentazione (è un vecchio trucco degli informatici esperti). Ed infatti qui trovo una risposta (o almeno, un indizio piuttosto grosso). Si dice di invocare spesso il metodo `runModalSession:` in modo che la finestra risponda correttamente agli eventi. Ecco il problema. Chiamando il metodo solo all'inizio del procedimento di aggiunta file, la finestra non risponde a tutti gli eventi successivi, fino al momento in cui l'operazione non è terminata. In particolare, non risponde agli eventi che le imporrebbero di andare in secondo piano.

Una volta trovato il problema, la soluzione è spesso ovvia. Ma qui c'è il problema che l'avanzamento lavori è nascosto all'interno di un metodo `initTreeFromPath:` dell'oggetto `FileStruct`, o meglio nel metodo `initWithPath:` dell'oggetto `LSFileInfo`. Una volta partito l'aggiornamento, non è possibile intervenire sulla sessione modale; è invece possibile intervenire sull'aspetto della finestra `WaitPanCtrl` con il metodo `setCurrentValue:`, il quale si occupa di aggiornare la barra di avanzamento lavori.

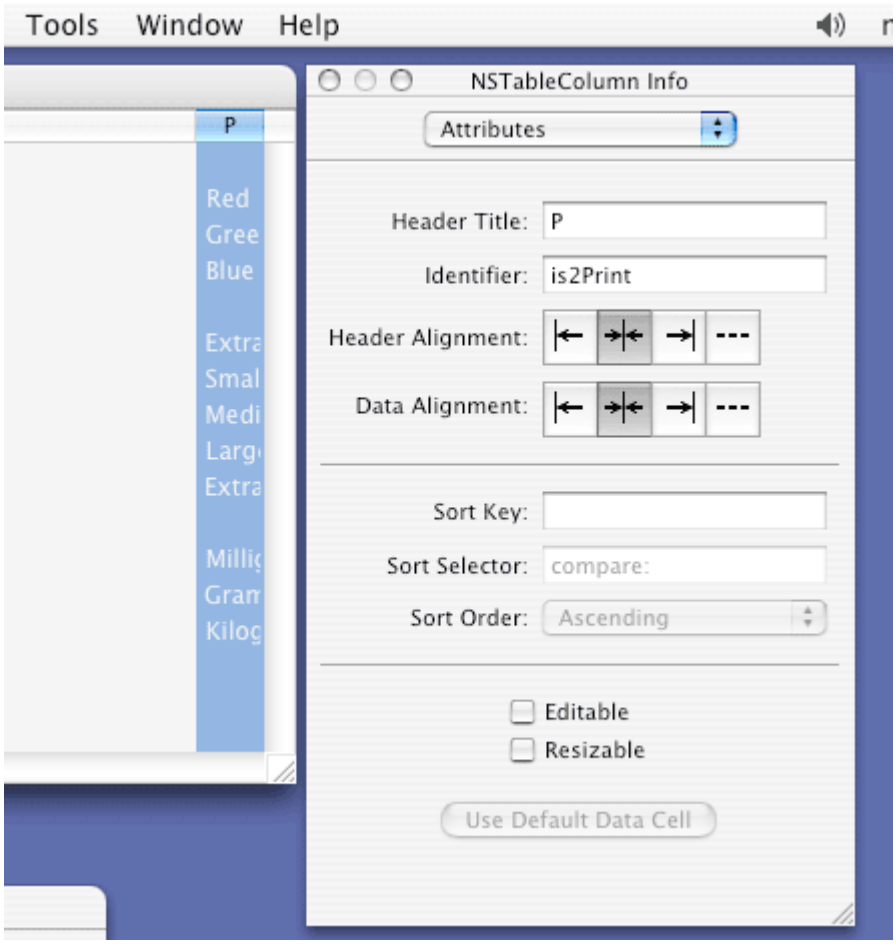
La soluzione più pigra che ho trovato è di informare la classe `WaitPanCtrl` di quale sia la sessione in corso, e di invocare il metodo `runModalSession:` per la sessione stessa. Modifico quindi la classe aggiungendo una variabile d'istanza dall'ovvio nome `session`; aggiungo un paio di metodi accessor. Adesso, all'apertura della sessione, all'interno del metodo `performAddFilesModal:` di `CatalogDoc`, informo la classe `WaitPanCtrl` del valore della variabile. Questo valore è utilizzato poi all'interno del metodo `setCurrentValue:` per invocare il metodo `runModalSession:`. Insomma, un brutto trucco di cui non vado orgoglioso, ma che raggiunge lo scopo prefisso. Adesso, non appena si passa ad un'altra applicazione, la finestra di avanzamento lavoro passa diligentemente in secondo piano.

UN NUOVO CAMPO

L'ultima modifica per questo capitolo è l'aggiunta di una variabile d'istanza booleana alle informazioni del file. Intendo utilizzare tale variabile successivamente, quando mi avventurerò nelle funzioni di stampa delle copertine per i CD. Col il valore della variabile indico se il file contribuisce alla stampa del catalogo, oppure no. A prescindere dal significato della variabile, introduco alcune modifiche interessanti. In primo luogo, voglio visualizzare il valore di questa variabile non con una stringa, ma attraverso un pulsante di spunta. Inoltre, l'utente deve poter essere in grado di modificare il valore della variabile operando direttamente sul pulsante, spuntando o meno il valore.

Questo implica la modificabilità dei dati visualizzati all'interno della outlineView, cosa finora esclusa. Ma andiamo per ordine.

Aggiungere una variabile d'istanza `is2Print` all'oggetto `LSFileInfo` è una cosa facile ma noiosa. ci sono da modificare parecchie cose: aggiungere la variabile e i metodi accessor, poi c'è da tenere conto dei metodi per il salvataggio e ripristino dati, insomma tanti piccoli interventi qui e lì. Decido tuttavia di non far intervenire questa variabile nelle preferenze, nell'ordinamento e nella ricerca.



Poi arriva la parte divertente. In primo luogo stabilisco che la colonna che visualizza la variabile è sempre presente (come il nome del file). Per fare questo, il metodo più spiccio è di intervenire con Interface Builder ed aggiungere direttamente nell'interfaccia la finestra. Bisogna fare attenzione al come si attribuisce il nome della colonna, visto che sfrutto un meccanismo per ricavare il valore di una cella a partire dal nome della colonna. Dico poi che la colonna avrà una dimensione fissa: per ottenere ciò, spunto l'apposito pulsante nella finestra di informazioni relativa alla colonna scelta.

Arrivo ora al nocciolo del problema. Così com'è, l'applicazione funziona, ma nella nuovo colonna è mostrato Zero oppure Uno a seconda dell'inizializzazione (sempre Uno, perché ho deciso di inizializzare a Vero la nuova variabile d'istanza). Devo fare in modo che la colonna non sia visualizzata come testo, ma come un pulsante di spunta. Ho già fatto (copiato) qualcosa di simile quando ho aggiunto l'icona al nome del file, costruendo una nuova classe `ImageAndTextCell`. All'inizio, ho provato a fare una cosa simile, utilizzando una cella del tipo `NSButtonCell`. Ho installato la cella al posto della cella standard alla costruzione della finestra (metodo `windowControllerDidLoadNib:` di

CatalogDoc):

```
...
NSButtonCell        * add2print = nil ;
...
// una nuova colonna con la visibilita' nella stampa
tableColumn = [outlineView tableColumnWithIdentifier: COLID_ADD2PRINT ];
add2print = [[[ NSButtonCell alloc] init ] autorelease ];
[ add2print setEditable: YES ];
[ add2print setButtonType: NSSwitchButton ];
[ add2print setTitle:@""] ;
[ tableColumn setDataCell:add2print];
...
```

In piena similitudine, ho cercato di impostare il corretto stato del pulsante all'interno del metodo

```
- (id) outlineView: objectValueForTableColumn: byItem:
```

Però, non funziona. Come al solito, ho cercato ispirazione nella documentazione, ed ho ri-scoperto un metodo della outlineView che permette di intervenire all'ultimo momento possibile sulla visualizzazione di una cella.

```
- (void)outlineView: (NSOutlineView *) oView willDisplayCell:(id)aCell
forTableColumn:(NSTableColumn *)aTableColumn item:(id)item
```

Si tratta di un metodo delegato della outlineView: si trova quindi all'interno dell'oggetto CatalogDoc. Il metodo è chiamato poco prima che sia visualizzata la cella aCell della colonna aTableColumn con i dati dell'elemento item. Con queste informazioni è estremamente facile impostare lo stato coretto del pulsante che rappresenta la cella:

```
- (void)
outlineView: (NSOutlineView *) oView
    willDisplayCell:(id)aCell
    forTableColumn:(NSTableColumn *)aTableColumn
    item:(id)item
{
    if ( [[ aTableColumn identifier] isEqualToString: COLID_ADD2PRINT ] )
    {
        BOOL    state = [ item is2Print];
        [aCell setState: ( state ? NSOnState: NSOffState) ];
    }
}
```

Se sono nella colonna giusta, leggo lo stato corrente della variabile; a seconda del suo valore, imposto lo stato del pulsante di spunta.

Il passo successivo è di permettere all'utente di modificare lo stato del pulsante. Ora, all'inizio dei tempi, alla prima realizzazione della outlineView, avevo utilizzato il metodo

```
outlineView: shouldEditTableColumn: item:
```

per impedire ogni intervento dell'utente sui valori della tabella. Pensavo di dover

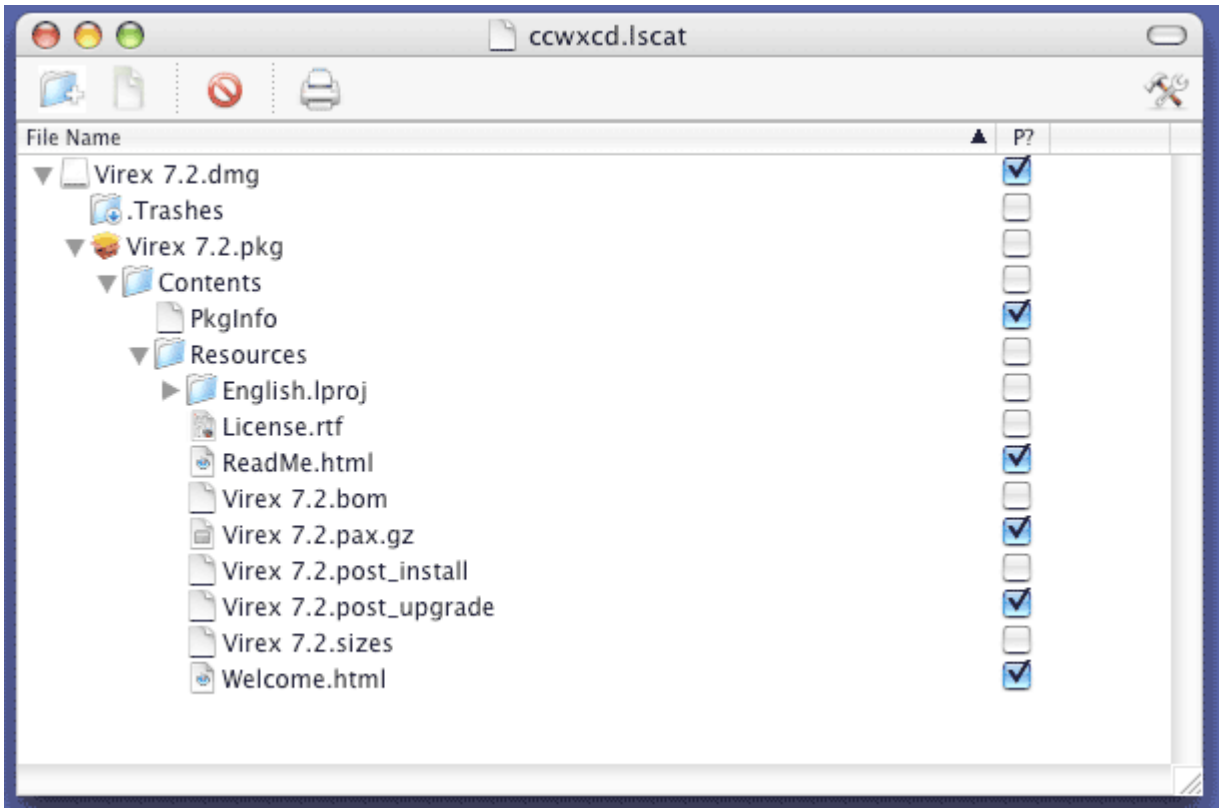
modificare tale metodo in modo da permettere la modifica della sola colonna coi pulsanti di spunta in questo modo:

```
- (BOOL)
outlineView:            (NSOutlineView *) outlineView
    shouldEditTableColumn:    (NSTableColumn *) tableColumn
    item:             (id) item
{
    if ( [[ tableColumn identifier] isEqualToString: COLID_ADD2PRINT ] )
        return YES ;
    return NO;
}
```

In realtà, non serve a nulla (e vai a capire perché). Mi è bastato aggiungere il seguente metodo nella classe sorgente di dati `LSDataSource`:

```
- (void)
outlineView:(NSOutlineView *) oView
    setObjectValue:(id)object
    forTableColumn:(NSTableColumn *)tableColumn
    byItem:(id)item
{
    if ( [[ tableColumn identifier] isEqualToString: COLID_ADD2PRINT ] )
    {
        BOOL    state = [ item is2Print];
        [ item setIs2Print: ! state ] ;
    }
}
```

Questo metodo, molto semplicemente, nel caso in cui si debba modificare il valore di una cella della colonna che mi interessa, rovescia lo stato della variabile booleana.



Posso azzardare una spiegazione del meccanismo: normalmente, quando si intende modificare il contenuto di una cella, si fa doppio clic. Se è permesso modificarla, è possibile manipolare il testo scrivendo ciò che si vuole. Una volta terminato, si esce in qualche modo (tabulatore, ritorno carrello, clic altrove) e si cerca di eseguire la modifica. Ora, il metodo `shouldEditTableColumn` dovrebbe consentire le modifiche in seguito al doppio clic. La cosa è vieppiù complicata dalla presenza in Interface Builder del pulsante di spunta `Editable` per ogni colonna, e dalla possibilità di modificare lo stato di editabilità del campo da programma, attraverso il metodo `setEditable:` relativo alla cella. In effetti, lasciando editabile il campo `is2Print`, un doppio clic permette la modifica del testo (di default vuoto, mi interessa solo lo stato del pulsante).

Quando si è terminato di editare il campo, interviene il metodo `setObjectValue:...`; con questo metodo è possibile cambiarne effettivamente il valore, aggiornando la sorgente dei dati. Se tale metodo non è fatto intervenire, alla visualizzazione successiva (praticamente da subito) la cella riprende il valore precedente, senza alcuna modifica.

C'è ancora una cosa da mettere a posto: quando si modifica lo stato di uno dei pulsanti, è in pratica cambiato lo stato del documento. Chiudendo la finestra del documento, sarebbe una buona cosa chiedere se si vogliono salvare le modifiche. Purtroppo, ancora una volta, la modifica avviene all'interno di un oggetto `LSDataSource`, che nulla conosce del documento associato. Ancora una volta, ricorro all'aggiunta di una variabile d'istanza che contenga un riferimento al documento cui la sorgente di dati è associata. Qui bisogna fare attenzione ad entrare in un circolo vizioso: il documento `CatalogDoc` ha una variabile d'istanza di tipo `LSDataSource`, ed adesso inserisco in `LSDataSource` una variabile d'istanza di tipo `CatalogDoc`. Per poter utilizzare un oggetto occorre prima dichiararlo, ad esempio importando il file .h relativo. Se in `CatalogDoc` includo `LSDatasource.h`, in quest'ultimo file non posso includere `CatalogDoc.h` (che altrimenti ci si morde la coda). Allora utilizzo la dichiarazione di diverso tipo:

```
@class    CatalogDoc ;
```

che informa dell'esistenza di una classe di nome `CatalogDoc`.

Fatto questo, aggiunta la variabile d'istanza ed i metodi accessor, la cosa si risolve facile aggiungendo un'assegnazione al momento del collegamento della sorgente dati al documento:

```
...
// e' la sorgente di dati di outlineView
[ outlineView setDataSource: dataSource];
// imposto l'ordine iniziale
[ dataSource setOrderColumn: [ NSString stringWithString: COLID_FILENAME]];
[ dataSource setOrderDirection: TRUE ];
[ dataSource setRefDoc: self ];
// carico i dati nella finestra
...
```

Il metodo che permette la modifica dei valori dello stato del pulsante diventa così

```
- (void)
```

```
outlineView:(NSOutlineView *) oView
    setObjectValue:(id)object
    forTableColumn:(NSTableColumn *)tableColumn
    byItem:(id)item
{
    if ( [[ tableColumn identifier] isEqualToString: COLID_ADD2PRINT ] )
    {
        BOOL    state = [ item is2Print];
        [ item setIs2Print: ! state ] ;
        // qui dovrei fare in modo di dire che il documento e' cambiato
        [ [ self refDoc ] updateChangeCount: NSChangeDone ] ;
    }
}
```

Con questo chiudo il capitolo, senza aver risolto alcuni problemi che mi si sono presentati nel frattempo (che non vi dico, ma che vi lascio scoprire da soli). Ma tanto, adesso cambierà tutto.

Capitolo 035 - Rifondazione

Essendo passato molto tempo dall'ultima volta che ho seriamente programmato con Cocoa, ho deciso di ricominciare daccapo. Chiamo il nuovo progetto, direttamente specificato in XCode, *CDCat2*. Comunque, nessuna paura: molto del codice scritto sarà riutilizzato, qualche volta senza modifica.

Leggo la documentazione...

Primo inserimento: 30 dicembre 2003

Obiettivo

Il punto di partenza della ristrutturazione è l'apertura di due finestre appartenenti allo stesso documento. La prima finestra è ancora la cara vecchia finestra con l'elenco completo di tutti i file del catalogo in una `NSOutlineView`. La seconda finestra vorrebbe essere un contenitore dove cominciare a disegnare la copertina per ogni CD che è stato catalogato. Per il momento, sarà semplicemente un modo differente di vedere il contenuto del catalogo.

Con l'occasione rivedo un po' tutte le classi finora definite, cambiando qualche nome e cercando di renderle più pulite. Ad esempio, la classe che contiene le informazioni di un file sarà semplificata, evitando di trascinarsi dietro un po' di informazioni francamente inutili. Contestualmente, modifico un po' anche l'aspetto delle finestre accessorie.

Poiché sarà un discorso piuttosto lungo, spezzerò questo capitolo in varie parti, continuando in altri due capitoli.

Struttura Dati

La struttura dati che contiene le informazioni di un file è divisa in tre classi: `CatFileInfo` (prende il posto di `LSFileInfo`), `FileStruct` e `VolInfo`. Gli oggetti `CatFileInfo` contengono tutte le informazioni relative ad un file considerato da solo: quindi nome, data di creazione, di modifica, permessi di accesso, dimensione, eccetera. Continuo ad utilizzare le funzioni messe a disposizione da Carbon per meglio caratterizzare le dimensioni del file. Dalla struttura dati ho eliminato un po' di informazioni che ritengo inutili o comunque poco interessanti ai fini della catalogazione; altre informazioni invece non saranno più visualizzabili all'interno delle finestre; rimangono comunque come appoggio per svolgere altre funzioni interessanti. In questo modo la struttura definitiva della classe `CatFileInfo` è la seguente:

```
// nomi delle colonne dei dati
#define         COLID_FILENAME        @"fileName"
#define         COLID_MODDATE        @"modDate"
#define         COLID_CREATDATE        @"creatDate"
#define         COLID_FILESIZE        @"fileSize"
#define         COLID_GROUPNAME        @"ownGroupName"
#define         COLID_OWNERNAME        @"ownerName"
#define         COLID_POSIXPERM        @"filePosixPerm"
```

```
#define        COLID_OSCREATOR        @"creatorCode"
#define        COLID_OSTYPE        @"typeCode"
#define        COLID_ADD2PRINT        @"is2Print"

@interface CatFileInfo : NSObject {
    NSString         *fileName ;
    NSDate             *modDate ;         // NSFileModificationDate
    NSDate             *creatDate ;     // NSFileCreationDate
    unsigned long long    fileSize;     // NSFileSize
    NSString         *ownGroupName ;     // NSFileGroupOwnerAccountName
    NSString         *ownerName ;    // NSFileOwnerAccountName
    unsigned long    filePosixPerm ;     // NSFilePosixPermissions
    OSType             creatorCode ;    // NSFileHFSCreatorCode
    OSType             typeCode ;         // NSFileHFSTypeCode
    BOOL            is2Print ;        // definito dall'utente
    NSImage            * fileIcon ;
    // caratteristiche che non saranno visualizzate
    NSString         *fileFullPath ;
    NSString         *fileType ;         // NSFileType
}

- (id)        initWithPath:        (NSString*) fullPath ;
- (void)    dealloc ;

- (void)    encodeWithCoder:    (NSCoder *)encoder ;
- (id)        initWithCoder:        (NSCoder *)decoder ;

// seguono tutti i metodi accessor
```

Alla fine, sono rimaste dieci caratteristiche visualizzabili (quelle per le quali ho riportato le `#define`) ed altre due non visualizzate ma utili a vari scopi.

Ovviamente, assieme a questa ristrutturazione dell'interfaccia della classe, segue una uguale modifica dei metodi presenti. Le modifiche sono tuttavia abbastanza semplici (si tratta di eliminare le istruzioni che fanno riferimento a variabili d'istanza non più esistenti), per cui non riporto il codice.

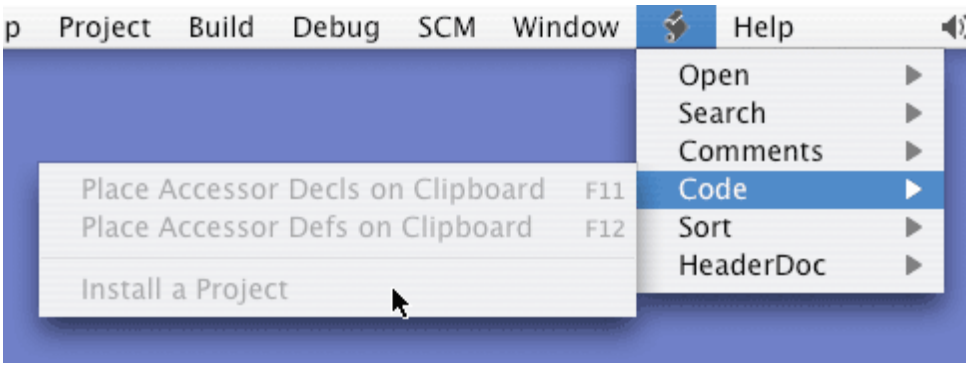
Le altre due classi utilizzate per modellare un file system sono le 'vecchie' `FileStruct` e `VolInfo`. Non hanno subito modifiche importanti. Dalla classe `VolInfo` ho tolto alcune variabili d'istanza poco significative, ed ho aggiunto i due metodi:

```
- (void)encodeWithCoder:(NSCoder *)encoder ;
- (id)initWithCoder:(NSCoder *)decoder ;
```

per effettuare un corretto salvataggio dei dati (in effetti, me ne ero dimenticato a suo tempo...).

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    // salvo le variabili ereditate
    [ super encodeWithCoder: encoder ];
    // vedo quanti file ci sono
      [ encoder encodeValueOfObjCType: @encode(unsigned long long) at: &
volSize];
      [ encoder encodeValueOfObjCType: @encode(unsigned long long) at: &
volFreeSize];
}
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// metodo per il salvataggio/ripristino dell'istanza da file
- (id)initWithCoder:(NSCoder *)decoder
{
    // recupero le variabili ereditate
    [ super initWithCoder: decoder ];
      [ decoder decodeValueOfObjCType: @encode(unsigned long long) at: &
volSize];
      [ decoder decodeValueOfObjCType: @encode(unsigned long long) at: &
volFreeSize];
    return self ;
}
```



Volevo notare una funzionalità presente in XCode, ovvero un menu che raccoglie una serie di script utili per la programmazione. Tra questi script ce ne sono un paio che aiutano la scrittura dei metodi accessor: sono le voci `Place Accessor Decls on Clipboard` e `Place Accessor Defs on Clipboard` del sottomenu `Code`. Il funzionamento è molto semplice: si selezionano le righe che contengono la variabili d'istanza e si esegue il comando da menu. Negli appunti sono presenti tutte le dichiarazioni delle funzioni (nel primo caso) oppure il codice vero e proprio delle funzioni accessor (il secondo caso).

Ho provato ad utilizzare tale funzione, ma mi sono trovato di fronte a diversi problemi. Il primo problema è noto, ed è l'incapacità dello script di riconoscere tipi di variabili un po' più complicate della struttura

```
<tipo con singola parola>    <nome della variabile> ;
```

Ad esempio con `unsigned long` non funziona. Di più, alcuni tipi semplici (uno tra tutti, il tipo `BOOL`) non è riconosciuto come scalare, e subisce quindi un trattamento da oggetto. Questo secondo problema si può risolvere facilmente, modificando lo script ed aggiungendo i tipi mancanti (lo script si trova nel file `/Library/Application Support/Apple/Developer Tools/Scrip ts/10-User Scripts/40-Code/10-accessorsFromIvars.sh`. Ci ho messo un bel po' a trovarlo...).

Tuttavia, il più grosso problema è dato dal codice prodotto; per quanto sia scritto da Apple, ho forti dubbi non tanto sul corretto funzionamento quanto sull'efficienza. Ad esempio, ogni metodo accessor di tipo `setNomeVar` esegue le seguenti operazioni:

```
TipoVar        * nomeVar ;
```

```
- (TipoVar *)nomeVar {
    return [[nomeVar retain] autorelease];
}

- (void)setNomeVar:(TipoVar *)newNomeVar {
    if (nomeVar != newNomeVar) {
        [nomeVar release];
        nomeVar = [newNomeVar copy];
    }
}
```

Non trovo particolarmente felice l'operazione di `copy` qualsiasi sia il tipo di oggetto; in taluni casi (ad esempio, l'intero catalogo!) è tremendamente pesante dal punto di vista delle risorse di calcolo. Per oggetti di dimensioni notevoli, un messaggio di `retain` dovrebbe essere sufficiente.

Particolarmente bizzarro il comportamento nel caso di variabili 'scalari', quali un intero, come si può vedere nel seguente esempio. L'assegnazione immediata del valore, senza prima il confronto col valore precedente, non provoca alcun problema.

```
int             nomeVar1 ;
- (int)nomeVar1 {
    return nomeVar1;
}

- (void)setNomeVar1:(int)newNomeVar1 {
    if (nomeVar1 != newNomeVar1) {
        nomeVar1 = newNomeVar1;
    }
}
```

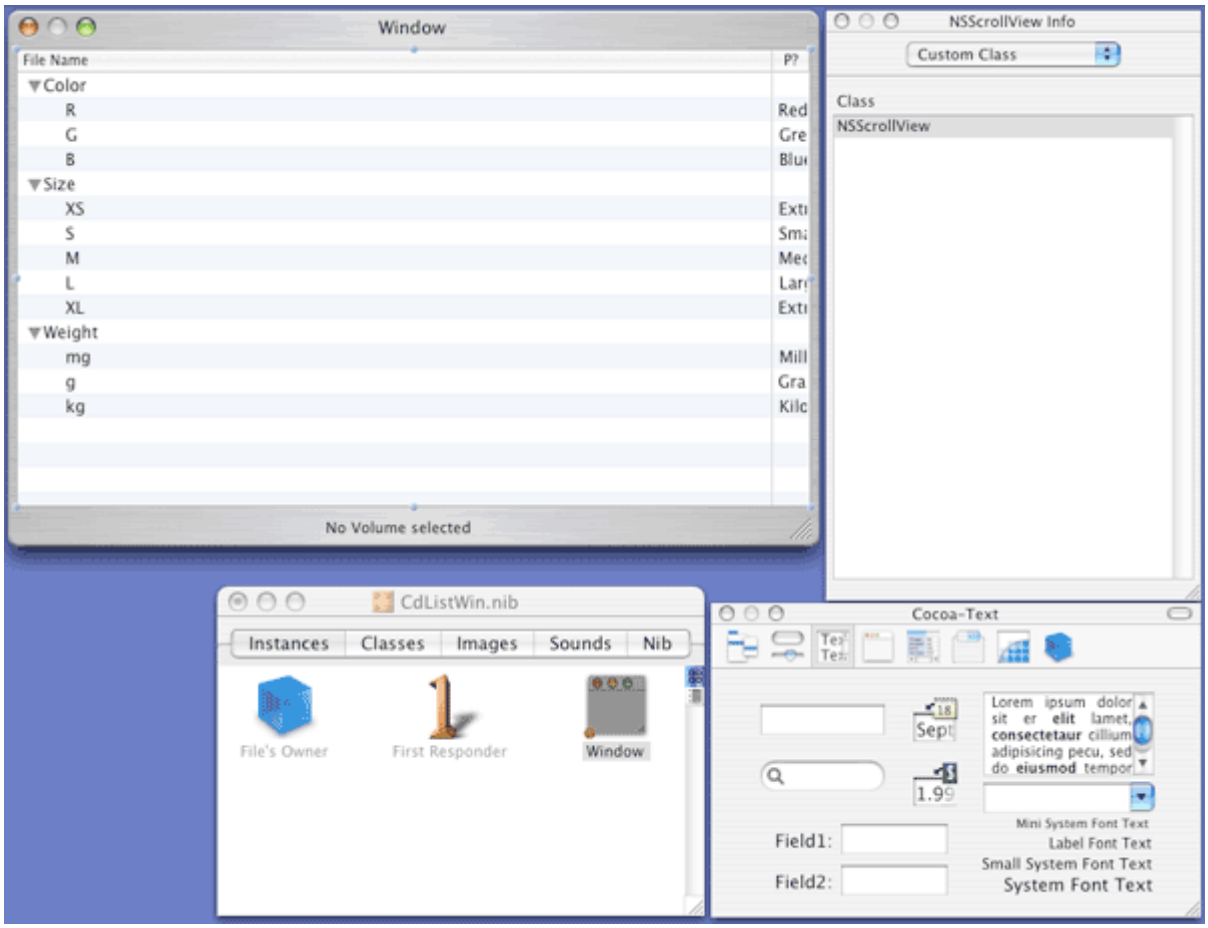
Ad ogni modo, in questa classe (ma anche in tutte le altre) ho riscritto tutti i metodi accessor basandosi sullo schema proposto; ho modificato sistematicamente i metodi `setNomeVar` per le variabili scalari e i metodi `nomeVar` per gli oggetti. Per quanto riguarda i metodi `setNomeVar` per gli oggetti, ho lasciato l'operazione di `copy` per gli oggetti *piccoli* (stringhe, ad esempio), e modificato il messaggio in `retain` per gli oggetti più consistenti.

CONTROLLORI PER FINESTRE

Uno dei motivi principali della ristrutturazione è la possibilità di aprire due finestre che rappresentino lo stesso documento. Con la tecnica sviluppata fino a questo momento, non è possibile; occorre passare ad un altro meccanismo. Ricordo che tre sono le classi preposta alla realizzazione di una applicazione *document-based*. C'è la classe `NSDocumentController`, unica all'interno dell'applicazione, che si occupa di gestire i documenti. Questi sono presenti sotto forma di classi `NSDocument`, una classe per ogni documento. Il documento è visualizzato poi all'interno di una finestra, il cui funzionamento è gestito da un oggetto della classe `NSWindowController`, uno per ogni finestra. Normalmente, quando si usa il principio 'un documento, una finestra', non occorre gestire esplicitamente l'oggetto `NSWindowController`, ed utilizzarne uno di default.

Tuttavia, per come intendo lavorare, devo gestire esplicitamente all'interno di

`NSDocument` la creazione degli oggetti `NSWindowController` (perché a questo punto ce ne saranno almeno due). Bisogna cambiare un po' di cose: la cosa interessante è che in effetti si riutilizza la maggior parte del codice già scritto, ma lo si sposta qui e lì.



Per prima cosa bisogna intervenire sul file NIB, o meglio, sulla caratterizzazione del `File's Owner`. Ricordo che il `File's Owner` è il *proprietario* del file NIB, l'oggetto che estrae e rappresenta a video tutti gli oggetti conservati nel file NIB. Attraverso il `File's Owner` passa tutta la comunicazione tra l'applicazione ed il contenuto del file NIB. Mentre nella versione precedente il `File's Owner` era un oggetto del tipo `NSDocument` (in particolare , `CatalogDoc`), adesso deve diventare un oggetto del tipo `NSWindowController`. Sfrutto le potenzialità di Interface Builder per dichiarare una nuova classe **ListWinCtl**, appunto sottoclasse di `NSWindowController`. Già che ci sono, faccio qualche modifica estetica alla finestra, aggiungendo la texture ed un campo di testo a mo' di barra di stato; ne cambio anche il nome in **CdListWin**. Completo la classe `ListWinCtl` con gli outlet necessari, poi genero i file relativi alla classe. Torno in Xcode per scrivere un po' di codice.

Prima di tutto, occorre caricare il file NIB. Poiché ho creato il nuovo progetto da zero, utilizzando l'apposito template fornito da XCode, mi trovo a dover riempire gli spazi vuoti della classe **CdCatDoc**, il nome con cui ho ribattezzato la nuova sottoclasse di documento. Nella versione precedente, era sufficiente scrivere il metodo `windowNibName` affinché ritornasse come stringa il nome del file NIB. Adesso non è più il caso: occorre utilizzare il metodo seguente.

```
- (void)
makeWindowControllers
{
    // istanza del controllore
    ListWinCtl * ctl =
        [[ ListWinCtl alloc ] initWithWindowNibName: @"CdListWin" ];
    // autrelease, che tanto poi sara' ritenuta
    [ ctl autorelease ] ;
    // aggiungo la fienstra al documento
    [ self addWindowController: ctl ];
}
```

In questo metodo, si costruisce un oggetto della classe `ListWinCtl` inizializzandolo col nome del NIB desiderato. Tale oggetto è stabilito essere `autorelease`, che tanto il messaggio successivo effettua un `retain`, associandolo al documento.

Presumo che ogni documento abbia come variabile d'istanza un vettore che contiene l'elenco di tutti gli oggetti `NSWindowController` associati. Nella procedura di inizializzazione dell'oggetto documento ad un certo punto è inviato il messaggio `makeWindowControllers` che procede alla costruzione di tutte le finestre associate. La realizzazione standard del metodo (quella utilizzata finora) presuppone un unico `NSWindowController` standard, che invia il messaggio `windowNibName` al documento stesso per avere un nome di file NIB da caricare. In questo caso invece il metodo `makeWindowControllers` fa di testa sua e costruisce un `NSWindowController` di tipo opportuno.

Passo adesso a costruire la finestra, e quindi alla classe `ListWinCtl`. Nella versione precedente, la finestra era meglio caratterizzata attraverso il codice del metodo `windowControllerDidLoadNib:`. Questo metodo è specifico di `NSDocument`, e qui non può essere utilizzato (non è vero, ripensandoci lo si potrebbe fare, tuttavia è meglio procedere in altro modo). Si utilizza il metodo `windowDidLoad` tipico degli oggetti `NSWindowController`. Presumo che quando il file NIB sia stato interamente caricato, sia inviato all'oggetto `NSWindowController` il messaggio `windowDidLoad`. Sospetto che la versione di default si limiti ad inviare il messaggio `windowControllerDidLoadNib:` al documento cui il controllore appartiene.

Ad ogni modo, all'interno del metodo ci vanno più o meno quelle istruzioni che prima erano presenti all'interno dell'oggetto `CatalogDoc`.

```
- (void)
windowDidLoad
{
    ImageAndTextCell    * imageAndTextCell = nil;
    NSTableColumn        * tableColumn = nil;
    CatDataSrc            * myDataSrc ;

    [super windowDidLoad ];
    // associo questa finestra alla sorgente dati
    myDataSrc = [ [self document] dataSource] ;
      // inserisco una cella custom nella outlineView; la cella standard
gestisce solo
    // testo, questa nuova testo ed una immagine
    tableColumn = [fullList tableColumnWithIdentifier: COLID_FILENAME];
    // creo l'oggetto autorelease perche' passa in carico alla tableColumn
    imageAndTextCell = [[[ImageAndTextCell alloc] init] autorelease];
```

```
    [tableColumn setDataCell:imageAndTextCell];
    // assegno alla colonna presente l'immagine che indica l'ordinamento
        [ fullList setIndicatorImage: [NSImage imageNamed: @"imgUpS"]
inTableColumn: tableColumn ];
    [ fullList setDelegate: myDataSrc ];
    [ fullList setDataSource: myDataSrc ];
    // salvo nelle prefs le dimensioni delle colonne della outlineView
    [ fullList setAutosaveName: @"FullListViewOptions" ];
    [ fullList setAutosaveTableColumns: YES ];
    // facendo doppio clic, si cambia la direzione di ordinamento
    [ fullList setTarget: self ];
    [ fullList setDoubleAction: @selector( changeSortOrder:) ];
#if 0
    // registro la finestra che accetti drag
    [ [ aController window] registerForDraggedTypes:
        [NSArray arrayWithObject: NSFilenamesPboardType] ];
#endif
    // attacco la toolbar alla finestra
    [self addToolbarToWin: [ self window] ];

    // mi abbono all'evento: pref aggiornate
    [[ NSNotificationCenter defaultCenter] addObserver: self
        selector: @selector( prefsUpdated: )
        name: PREF_UPDATE_NOTIFICATION object: nil ] ;
    // recupero le pref ed adeguo la finestra
    // in realta', faccio finta che siano state aggiornate le prefs
    [[NSNotificationCenter defaultCenter]
        postNotificationName: PREF_UPDATE_NOTIFICATION object:self];
}
```

Qui ci sono alcune osservazioni. In primo luogo, dall'oggetto `ListWinCtl` si risale immediatamente al documento di appartenenza utilizzando il messaggio `document`. Presumo che la variabile d'istanza associata sia predisposta automaticamente dalla procedura di inizializzazione. In questo modo è possibile risalire immediatamente a tutte le altre variabili d'istanza del documento necessarie per il corretto funzionamento della finestra. La finestra in pratica consiste nella outlineView (e nel campo di testo che funziona da barra di stato, di cui per il momento non mi occupo). Per la corretta gestione della outlineView occorre individuare due enti ausiliari: la sorgente dei dati e la classe delegata. Nella versione precedente attribuivo le funzioni di sorgente di dati ad una classe apposita, e le funzioni di classe delegata erano attribuite al documento. Questa volta decido di attribuire entrambe le funzioni ad una unica classe; poiché la funzione principale è di essere sorgente di dati, dico che la classe nel codice individuata come

```
[self document] dataSource
```

svolgerà questi compiti. Rispetto alla versione precedente mancano alcuni pezzi di codice. La creazione e l'inizializzazione della sorgente dei dati avviene all'interno della classe `CdCatDoc`, all'interno del metodo `init` (è giusto qui sotto). È scomparsa la gestione della colonna `COLID_ADD2PRINT` introdotta più di recente, in quanto sarà gestita altrove. Per il momento, non attivo il funzionamento del drag and drop.

```
- (id)
init
{
    self = [super init];
```

```
    if (self)
    {
        CatDataSrc        * dataSrc = [[CatDataSrc alloc] init ] ;
        [ dataSrc autorelease ] ;
        // devo costruire l'oggetto sorgente dei dati
        [ self setDataSource: dataSrc ];
        [ dataSource setRefDoc: self ];
    }
    return self;
}
```

Per il resto, nulla di nuovo.

Capitolo 036 - Altre finestre

Continuo dal capitolo precedente: qui si apre una seconda finestra sullo stesso documento.

Leggo la documentazione...

Primo inserimento: 2 gennaio 2004

PREFERENZE E GESTIONE COLONNE

La classe `UserPrefs` per la gestione delle preferenze non è sostanzialmente cambiata: ho eliminato solamente i valori che si riferiscono a campi non più presenti o non più visualizzati. È invece leggermente cambiata la gestione della visualizzazione delle colonne. Adesso questa operazione è volta all'interno della classe `ListWinCtl`, come per altro dovrebbe essere stato chiaro quando ho indicato `self` come `target` delle notifiche di aggiornamento delle preferenze. Ho aggiustato il metodo `prefsUpdated` per tenere conto della mutata numerosità delle colonne. Ho anche trasferito qui la gestione della colonna `COLID_ADD2PRINT`, cosa che mi permette di eseguire un giochino interessante. Decido infatti che questa colonna sia sempre l'ultima della serie. Per fare questo, ho aggiunto tre righe (ma solo per chiarezza, avrei potuto fare tutto con una unica istruzione...).

```
- (void)
prefsUpdated: ( NSNotification *) notification
{
    int fi, ti ;
    // recupero le preferences e mostro o meno la colonna
       setupColumn( fullList, [[ UserPrefs getPrefValue: keyColModDate]
boolValue], COLID_MODDATE );
      setupColumn( fullList, [[ UserPrefs getPrefValue: keyColCreatDate]
boolValue], COLID_CREATDATE );
       setupColumn( fullList, [[ UserPrefs getPrefValue: keyColFileSize]
boolValue], COLID_FILESIZE );
      setupColumn( fullList, [[ UserPrefs getPrefValue: keyColGroupName]
boolValue], COLID_GROUPNAME );
      setupColumn( fullList, [[ UserPrefs getPrefValue: keyColOwnerName]
boolValue], COLID_OWNERNAME );
      setupColumn( fullList, [[ UserPrefs getPrefValue: keyColPosixPerm]
boolValue], COLID_POSIXPERM );
       setupColumn( fullList, [[ UserPrefs getPrefValue: keyColCreator]
boolValue], COLID_OSCREATOR );
        setupColumn( fullList, [[ UserPrefs getPrefValue: keyColOSType]
boolValue], COLID_OSTYPE );
    // questa colonna e' sempre visualizzata
    setupColumn( fullList, TRUE , COLID_ADD2PRINT );
    // sposto la colonna is2Print in ultima posizione
    fi = [ fullList columnWithIdentifier: COLID_ADD2PRINT ];
    ti = [ fullList numberOfColumns] - 1 ;
    [ fullList moveColumn: fi toColumn: ti ];
    [ fullList reloadData ];
}
```

Sfrutto alcuni metodi presenti all'interno della classe per trovare l'indice corrente della

colonna, individuare quante sono le colonne, e spostare poi la colonna in ultima posizione (gli indici, a quanto pare, partono da Zero, quindi l'ultima colonna è data dal numero di colonne presenti meno uno).

La funzione `setupColumn` non è stata modificata, se non per il fatto che la funzione in precedenza chiamata `attachFormatter` adesso è meglio individuata con `configureColumn`; all'interno di questa funzione infatti procedo alla personalizzazione delle colonne. Nella maggior parte dei casi si tratta solamente di attaccare un formattatore per la corretta visualizzazione del valore; fa eccezione proprio la colonna `COLID_ADD2PRINT` che subisce un trattamento particolare:

```
    // ... altro codice ...
    if ( [ colId isEqual: COLID_ADD2PRINT ] )
    {
        NSButtonCell        * add2print = nil ;
        add2print = [[[ NSButtonCell alloc] init ] autorelease ];
        [ add2print setButtonType: NSSwitchButton ];
        [ add2print setTitle:@""] ;
        [ tc setWidth: 16 ];
        [ tc setResizable: FALSE ];
        [ [ tc headerCell] setStringValue: @"P?" ];
        [ tc setDataCell:add2print];
    }
    // ... altro codice ...
```

Qui, oltre all'impostazione del tipo di cella, la si costringe ad una dimensione fissa di 16 pixel (giusto quelli necessari per visualizzare il pulsante di spunta).

L'ORDINE DEGLI ELEMENTI

Ho modificato largamente il meccanismo preposto alla gestione dell'ordinamento degli elementi all'interno della outlineView. Ho già stabilito all'interno del metodo `windowDidLoad` che con un doppio clic si seleziona un diverso criterio, in base alla colonna prescelta. Questa serie di operazioni è volta dal metodo seguente, che è stato modificato (e spero migliorato):

```
- (void)
changeSortOrder: (id) sender
{
    NSTableColumn    * tableColumn ;
    CatDataSrc        * myDataSrc ;

    // associo questa finestra alla sorgente dati
    myDataSrc = [ [self document] dataSource] ;
    // voglio che sia selezionata una intera colonna
    if ( [ sender selectedColumn ] == -1 ) return ;
    // determino allora quale colonna e' selezionata
      tableColumn = [ [fullList tableColumns] objectAtIndex: [ fullList
clickedColumn ]];
      if ([ [ tableColumn identifier] isEqualToString: COLID_ADD2PRINT])
return ;
    // se la colonna di ordinamento non cambia
    if ([ [ tableColumn identifier] isEqual: [ myDataSrc orderColumn ]])
    {
        // rovescio l'ordine e cambio l'immagine (lo faccio poi)
        [ myDataSrc setOrderDirection: ! [ myDataSrc orderDirection ] ];
```

```
    }
    else
    {
        [ fullList setIndicatorImage: nil
            inTableColumn: [ fullList
                tableColumnWithIdentifier: [ myDataSrc orderColumn ]
            ]
        ];
        // rovescio l'ordine negando il valore corrente
        [ myDataSrc setOrderColumn: [ tableColumn identifier]];
    }
    // in ogni caso imposto l'immagine nello header della colonna
    if ( [ myDataSrc orderDirection ])
              [ fullList setIndicatorImage: [NSImage imageNamed: @"imgUpS"]
inTableColumn: tableColumn ];
     else     [ fullList setIndicatorImage: [NSImage imageNamed: @"imgDwS"]
inTableColumn: tableColumn ];
    // rinfresco la finestra con i nuovi dati
    [ fullList reloadData ];
}
```

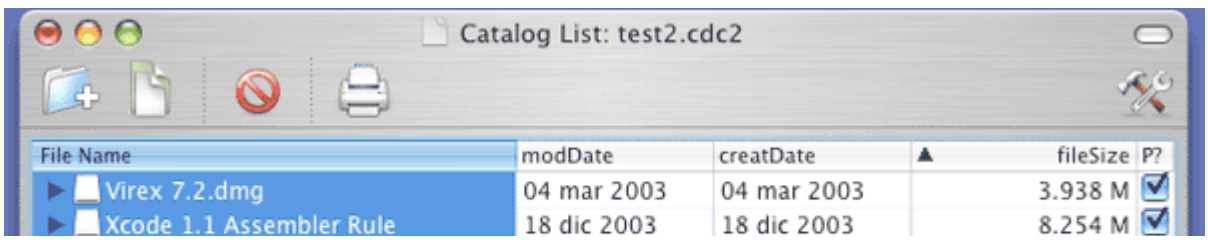
In primo luogo stabilisco che non si possono ordinare gli elementi secondo la colonna `COLID_ADD2PRINT`; in effetti in questo caso il metodo fa nulla e lascia tutto come trovato.

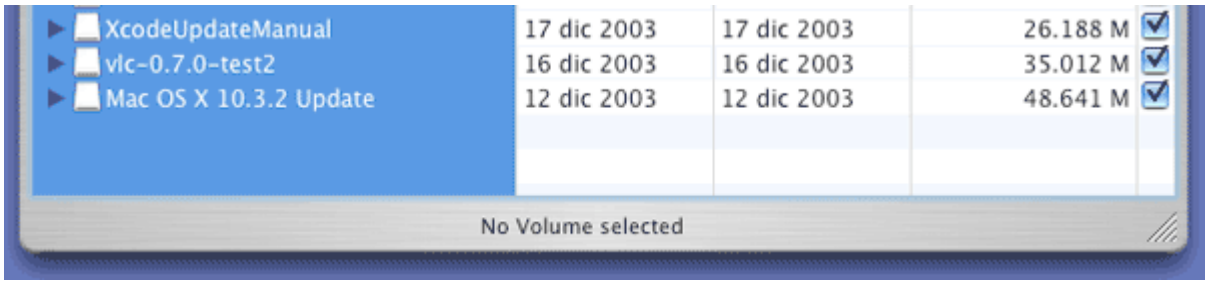
Poi verifico se la colonna selezionata è la stessa che governa l'ordine corrente; in tal caso significa che devo solamente rovesciare l'ordinamento, ma non cambiare la colonna. Altrimenti, significa che devo cambiare colonna (nella versione precedente, un doppio clic generava un cambiamento di colonna ed anche di direzione di ordinamento; adesso cambia solo la colonna, ma la direzione non cambia). Da notare anche il meccanismo con cui tolgo il triangolino dalla colonna in cui si trovava (basta rintracciare la colonna che governa l'ordinamento corrente: lo si fa attraverso il suo identificatore, conservato nella sorgente dei dati).

In ogni caso, imposto la nuova immagine triangolo nella colonna cliccata, ed aggiorno il contenuto della finestra (è questo aggiornamento forzato che provoca materialmente l'ordinamento dei dati all'interno della sorgente).

Contestualmente è stato modificato e semplificato un metodo delegato della outlineView, che svolgeva alcune funzioni che si sovrapponevano (adesso fa parte della classe sorgente di dati; nella versione precedente era invece dentro la classe `CatalogDoc`):

```
- (BOOL)
outlineView:(NSOutlineView *) oView
    shouldSelectTableColumn:(NSTableColumn *)tableColumn
{
    return YES;
}
```





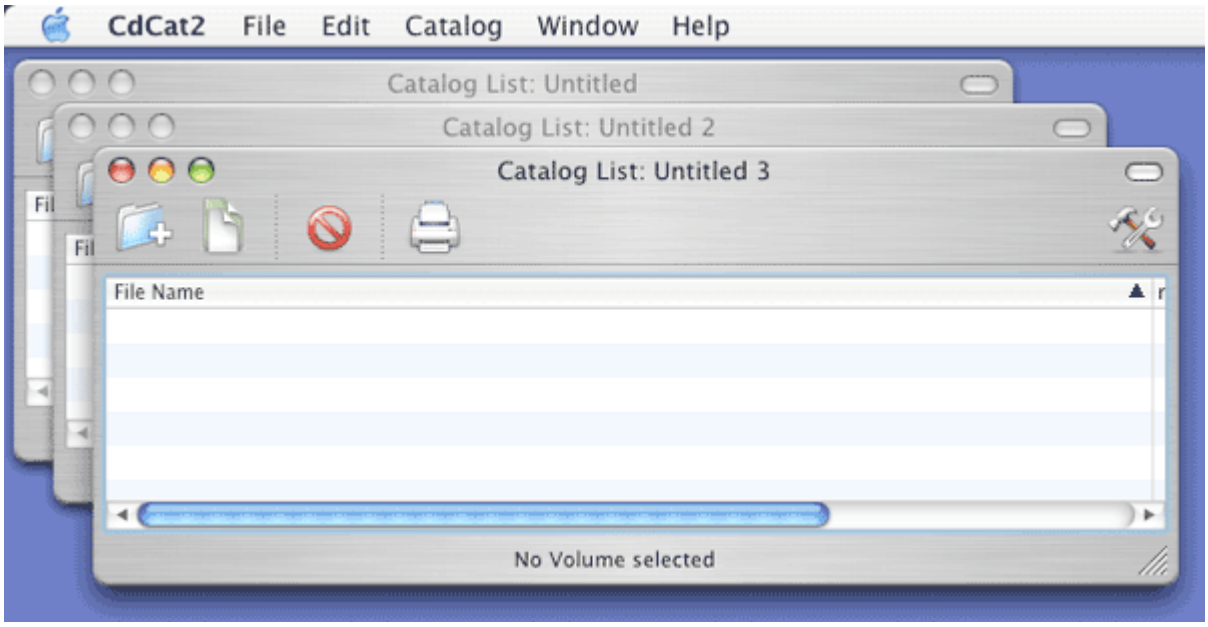
Adesso la selezione di una colonna e l'ordinamento sono concetti slegati tra loro. Facendo doppio clic su di una colonna, ovviamente si seleziona la colonna, ed il contenuto della finestra è ordinato secondo la colonna stessa. Tuttavia, è possibile selezionare altre colonne senza provocare riordinamenti. Facendo doppio clic sulla colonna `COLID_ADD2PRINT` si ottiene la sola selezione della colonna, ma nessun riordinamento.

ULTIMI TOCCHI ALLA FINESTRA

Ci sono ancora alcuni argomenti da toccare per concludere la trattazione di questa finestra.

La prima cosa è un bel tocco di classe per cambiare il nome visualizzato; dal momento che potrebbero esserci più viste dello stesso documento, è bene dare un nome più significativo alla finestra.

```
- (NSString *)
windowTitleForDocumentDisplayName:(NSString *)displayName
{
    return ( [ @"Catalog List: " stringByAppendingString: displayName] );
}
```



Il metodo sopra indicato è invocato ogni volta che deve essere visualizzata una finestra; l'argomento `displayName` è tipicamente la stringa `Untitled` seguita da un numero progressivo. Il metodo di default fa nulla, e restituisce la stringa intonsa. Qui faccio lo

overload del metodo e inserisco davanti al nome una caratterizzazione della finestra. Questo metodo è anche comodo se voglio utilizzare un nome differente, al posto di `Untitled`, per scopi non meglio imprecisati.

C'è poi un metodo che prima mostro poi spiego:

```
- (BOOL)shouldCloseDocument
{
    return ( YES );
}
```

A quanto pare, Cocoa si occupa di gestire per me una serie di faccende noiose. In particolare, gestisce autonomamente lo stato dei documenti, e chiede gentilmente se è il caso di salvare un documento quando chiudo la finestra associata. Normalmente, essendoci una sola finestra per documento, la cosa non dovrebbe ingenerare equivoci: chiudendo la finestra, si chiude il documento associato. Cosa succede quando un documento presenta più di una finestra? Cocoa di default è tanto gentile da gestire per me la cosa: finché ci sono finestre aperte relative ad un documento, non sono inviate richieste di salvataggio. Non appena si chiude l'ultima finestra, chiede, se il caso, se salvare o meno il documento. Questo comportamento di default potrebbe non sempre essere quello gradito. Ad esempio, potrei avere una finestra principale sul documento, e le altre finestre sono solo accessorie. In tal caso, finché si chiudono finestre accessorie, non si sono problema. Tuttavia, potrei volere che alla chiusura della finestra principale si chiudesse tutto. Ecco, questo è il mio caso: dico che quando chiudo la finestra gestita da `ListWinCtl`, anche il documento debba essere chiuso (e che quindi si scateni, se il caso, il processo di salvataggio). In altri termini, quando il controllore chiude la finestra, si invia il messaggio `shouldCloseDocument`; normalmente il metodo corrispondente risponde NO. A meno che non sia l'ultima finestra associata al documento, il documento è lasciato aperto. Se però si fa lo overload di questo metodo, restituendo `YES` (qui per partito preso, ma la risposta potrebbe dipendere da altre questioni), il controllore della finestra invia al documento una richiesta di chiusura. Se questa richiesta procede, si chiude il documento (e quindi, tutte le altre finestre eventualmente aperte).

La toolbar è rimasta immutata. L'unico accorgimento è stato quello di ridirigere verso il documento alcuni messaggi (uno solo, in effetti, quello per il salvataggio del file) inviati dai pulsanti. Uguale sorte per il metodo che permette di aggiungere elementi al catalogo:

```
- (void)
addFileItem: (id)sender
{
    ChooseVolCtrl *tmpCVCtrl = [[ ChooseVolCtrl alloc] init ] ;
    // comincio la procedura: mi metto modale per il documento
    // alla chisura della sheet, sara' invocato il metodo addVolume:...
    [ NSApp beginSheet: [ tmpCVCtrl window ]
        modalForWindow: [ NSApp mainWindow ]
        modalDelegate: [ self document]
        didEndSelector: @selector( addVolume2Cat:returnCode:contextInfo: )
        contextInfo: nil
    ];
}
```

Qui non ci sono differenze se non appunto la delega al documento al termine delle

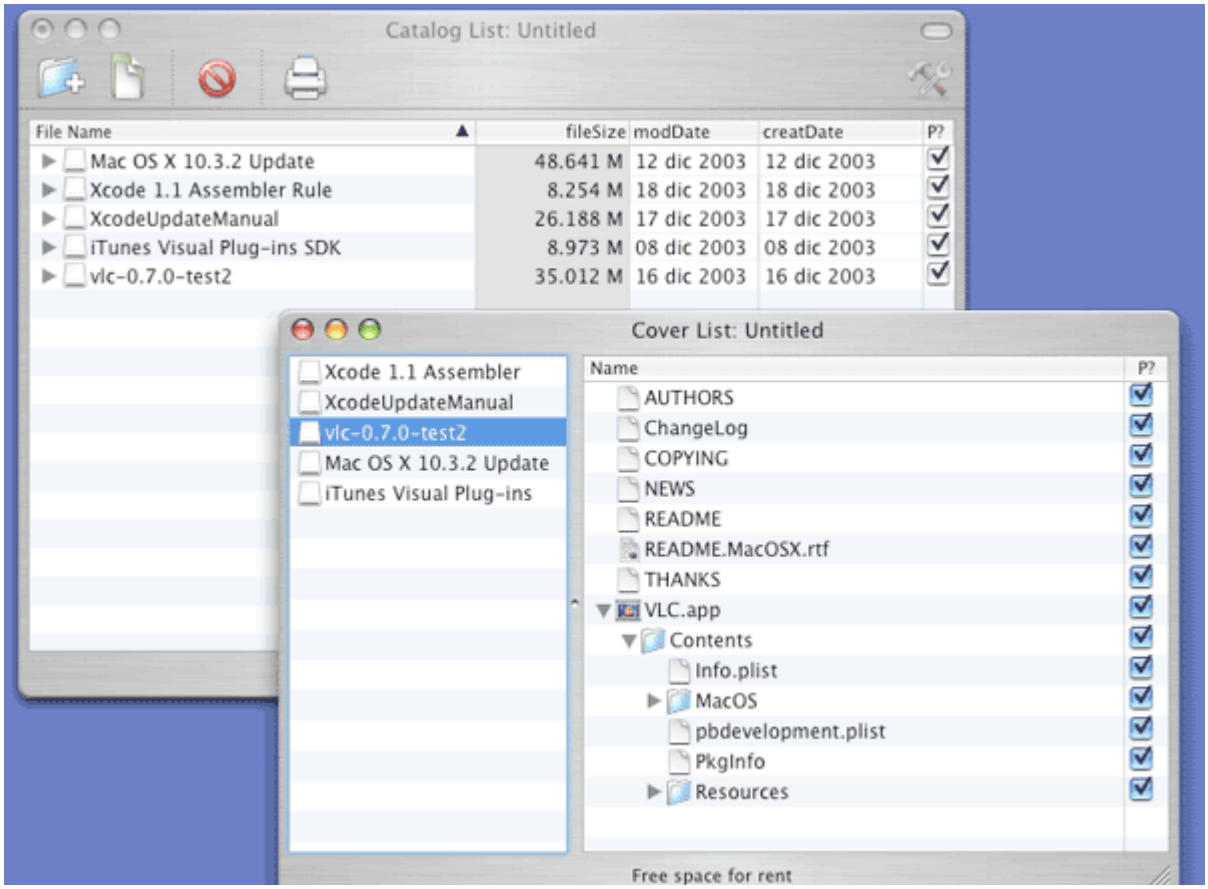
operazioni di selezione del volume attraverso la finestra apposita (che non ha subito modifiche).

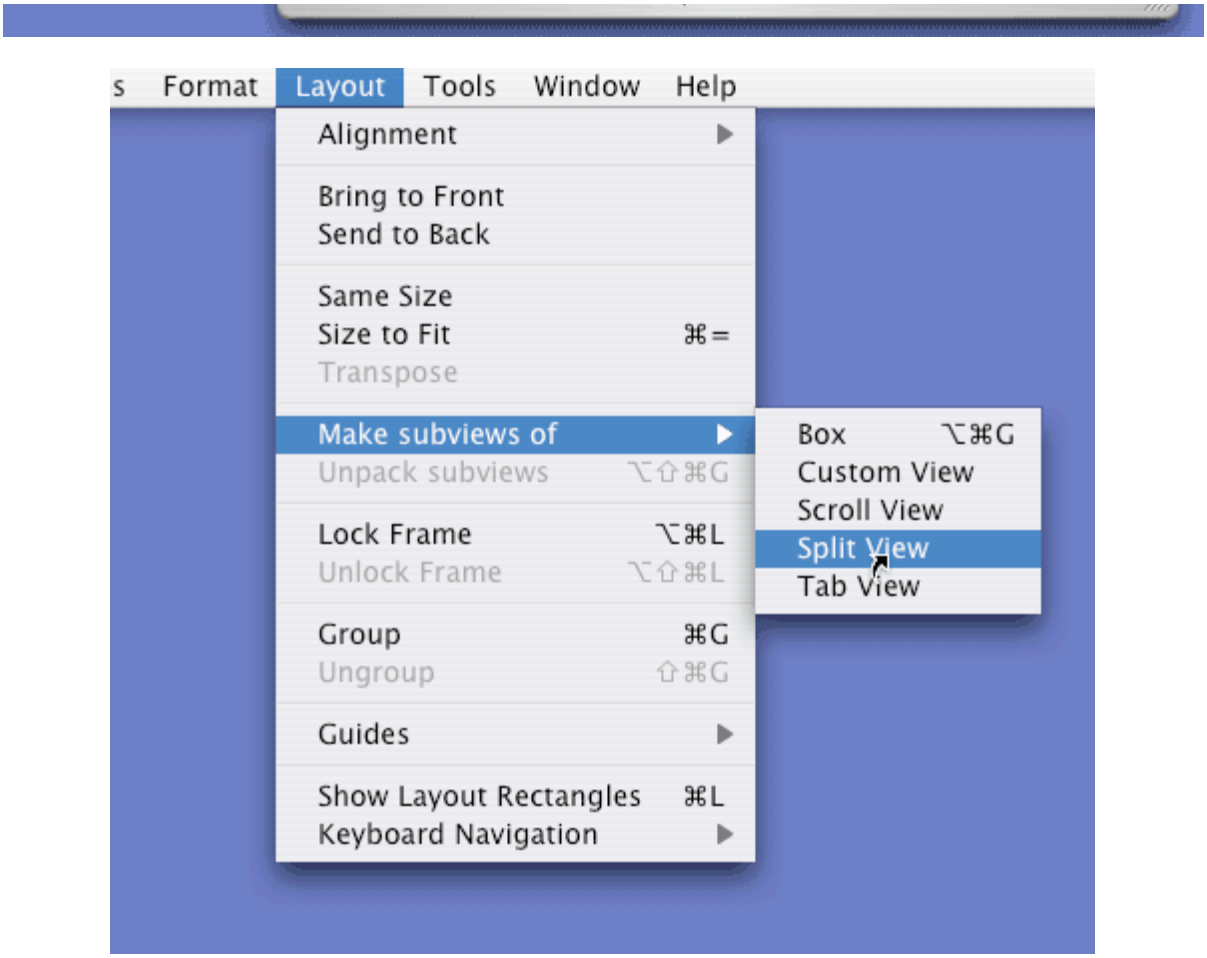
Di diverso avviso invece il metodo invocato per la cancellazione di un elemento dal catalogo.

```
- (void)
delItem: (id)sender
{
    FileStruct * fInfo ;
    // guardo se c'e' una riga selezionata
    int    rowsel = [ fullList selectedRow ];
    if ( rowsel == -1 )
    {
        // no, lascio stare
        NSBeep();    // bip!!
        return ;
    }
    // se arrivo qui, c'e' una riga selezionata
    fInfo = [ fullList itemAtRow: rowsel ] ;
    // adesso voglio eliminare questo elemento da dataSource
    [ [[ self document] dataSource] removeEntry: fInfo ];
}
```

Qui ho preferito raccogliere le informazioni sull'oggetto selezionato, e poi passare la palla alla sorgente dati utilizzando un nuovo, apposito, metodo, per la rimozione di un elemento (che vedrò più avanti).

UNA DIVERSA FINESTRA





È giunto il momento di aprire una nuova finestra sul documento. L'obiettivo finale è che questa finestra sia utilizzata per predisporre la stampa della copertina del CD, secondo i vari formati disponibili (slim, jewel box, eccetera). Per il momento, tuttavia, mi limito a dare una diversa rappresentazione della sorgente dei dati. La rappresentazione individuata assomiglia a quella del Finder secondo Panther, ovvero una vista laterale in cui sono elencati i Volumi, ed una outlineView centrale in cui sono elencati i file del volume selezionato. Si fa prima a vedere una figura che raccontarlo. Costruisco la finestra all'interno di Interface Builder. Per poter avere un comportamento corretto, occorre rendere la `NSTableView` di sinistra e la `NSOutlineView` di destra delle sottoviste di una unica classe `NSSplitView`. Per fare questo, occorre selezionare le due View e selezionare la voce corretta dal menu `Layout` -& `Make subviews of` -& `Split View`. Decido che la outlineView non sarà configurabile, ma mostra solo due colonne: il nome del file e la colonna `COLID_ADD2PRINT` sotto forma di pulsante di spunta. La tableView invece mostrerà i volumi completi di icona.

Sempre all'interno di Interface Builder battezzo la nuova classe `CoverWinCtl` che funziona come controllore della finestra. Assegno questo tipo al `File's Owner`, aggiungo i soliti outlet e faccio i collegamenti standard. Costruisco i file e passo a scrivere un po' di codice.

Fondamentalmente c'è un unico metodo da scrivere, il seguente:

```
- (void) windowDidLoad
{
```

```
    NSTableColumn        * tableColumn = nil;
    ImageAndTextCell    * imageAndTextCell = nil;
    CatDataSrc            * myDataSrc ;
    NSButtonCell        * add2print = nil ;

    [super windowDidLoad ];
    // associo questa finestra alla sorgente dati
    myDataSrc = [ [self document] dataSource] ;
    [ volList setDelegate: myDataSrc ];
    [ volList setDataSource: myDataSrc ];
    // salvo nelle prefs le dimensioni delle colonne della outlineView
    [ volList setAutosaveName: @"VolumeViewOptions" ];
    tableColumn = [volList tableColumnWithIdentifier: @"volumeList"];
    // creo l'oggetto autorelease perche' passa in carico alla tableColumn
    imageAndTextCell = [[[ImageAndTextCell alloc] init] autorelease];
    [imageAndTextCell setEditable: NO];
    [tableColumn setDataCell:imageAndTextCell];

      // inserisco una cella custom nella outlineView; la cella standard
gestisce solo
    // testo, questa nuova testo ed una immagine
    tableColumn = [fileList tableColumnWithIdentifier: COLID_FILENAME];
    // creo l'oggetto autorelease perche' passa in carico alla tableColumn
    imageAndTextCell = [[[ImageAndTextCell alloc] init] autorelease];
    [tableColumn setDataCell:imageAndTextCell];
    // una nuova colonna con la visibilita' nella stampa
    tableColumn = [fileList tableColumnWithIdentifier: COLID_ADD2PRINT ];
    add2print = [[[ NSButtonCell alloc] init ] autorelease ];
    [ add2print setButtonType: NSSwitchButton ];
    [ add2print setTitle:@""] ;
    [ tableColumn setDataCell:add2print];
    [ fileList setAutosaveName: @"FileListOptions" ];
    [ fileList setDelegate: myDataSrc ];
    [ fileList setDataSource: myDataSrc ];

    // mi abbono all'evento: pref aggiornate
    [[ NSNotificationCenter defaultCenter] addObserver: self
        selector: @selector( prefsUpdated: )
        name: PREF_UPDATE_NOTIFICATION object: nil ] ;
    // recupero le pref ed adeguo la finestra
    // in realta', faccio finta che siano state aggiornate le prefs
    [[NSNotificationCenter defaultCenter]
        postNotificationName: PREF_UPDATE_NOTIFICATION object:self];
}
```

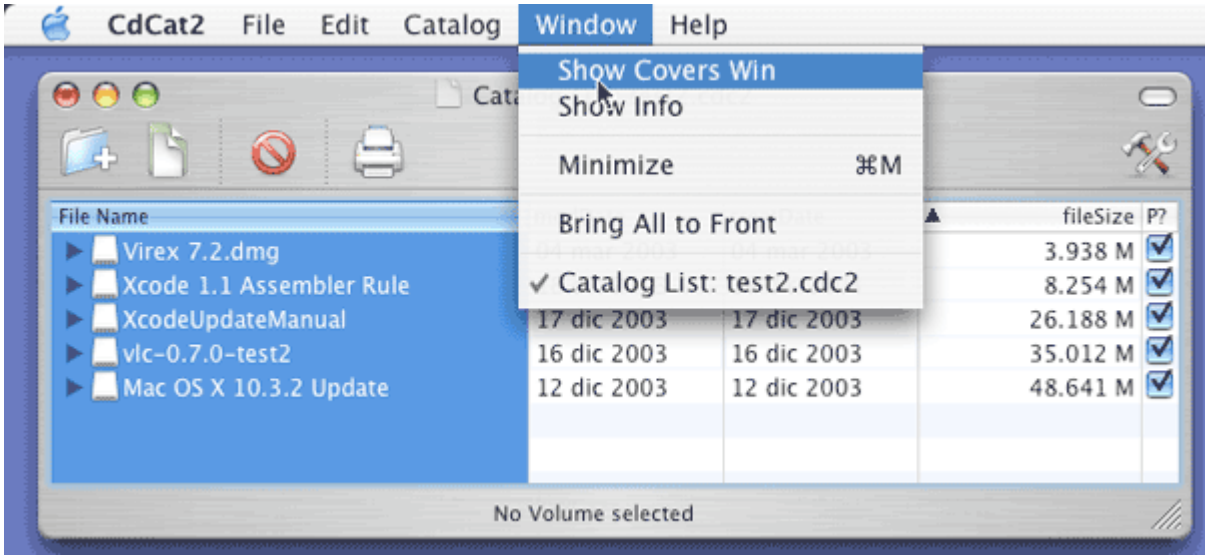
Non ci sono grosse cose da notare. Lo outlet `volList` rappresenta la tableView; poiché deve mostrare i volumi completi di icona, assegno la cella della colonna in modo opportuno. Ugualmente, configuro la `fileList` (outlet che rappresenta la outlineView) con le due colonne col nome del file e la variabile `is2Print`.

Sia `volList` che `fileList` utilizzano la classe sorgente di dati come sorgente di dati e classe delegata. Questo fatto porrà un interessante problema sulla gestione della sorgente dati. Infatti questa classe dovrà fornire i valori da rappresentare all'interno di due diverse outlineView, in due modi differenti. Poiché non avrebbe senso avere due diverse sorgenti che rappresentano sostanzialmente la stessa base di dati, all'interno della classe sorgente dati occorre un meccanismo per distinguere tra le due outlineView.

Ma prima di passare alla soluzione del problema, completo l'esposizione della classe `CoverWinCtl` con due metodi residui:

```
- (NSString *)
windowTitleForDocumentDisplayName:(NSString *)displayName
{
    return ( [ @"Cover List: " stringByAppendingString: displayName] );
}

- (void)
prefsUpdated: ( NSNotification *) notification
{
    [ fileList reloadData ];
    [ fileList sizeLastColumnToFit ] ;
}
```

Il primo metodo impone un prefisso al nome della finestra, come ho già fatto per l'altra finestra. Il secondo metodo forza un rinfresco dei dati della outlineView quando c'è una variazione nelle Preferenze. Qui le preferenze che hanno effetto sono quelle riguardanti la visualizzazione dei *dotFiles* e l'espansione dei *bundle*.



A questo punto, bisogna trovare un meccanismo per visualizzare questa finestra. Una prima soluzione è quella di visualizzare entrambe le finestre alla creazione del documento; basta aggiungere tre righe al metodo `makeWindowControllers` sulla falsariga delle istruzioni già presenti. Invece, ho deciso di utilizzare un comando da menu, aggiunto all'interno del file `MainMenu.nib`. Questa voce di menu invia un messaggio alla classe delegata dell'applicazione per invocare il metodo seguente (nel file `AppDelegate.m`):

```
- (IBAction)showCoversWindow:(id)sender
{
    // piglio il documento davanti e gli dico di mostrare l'altra finestra
     [ [[ NSApp orderedDocuments] objectAtIndex: 0] showCoversWindow ] ;
}
```

Il metodo a sua volta invia un messaggio al documento che si trova di fronte a tutti gli altri.

```
- ( void)
showCoversWindow
{
```

```
    int numOfWC = [ [ self windowControllers ] count ];
    CoverWinCtl * ctl1 ;

    // se ci sono due finestre, devo fare nulla
    if ( numOfWC == 2 ) return ;
    // se arrivo qui, c'e' una sola finestra
    // per come ho predisposto le cose, puo' essere solo quella della
    // completa lista di file, e quindi manca proprio la finestra
    // delle covers
    ctl1 = [[ CoverWinCtl alloc ] initWithWindowNibName: @"CoversWin" ];
    [ ctl1 autorelease ] ;
    [ self addWindowController: ctl1 ];
    [[ ctl1 window ] makeKeyAndOrderFront: self ];
}
```

Il metodo conta quanti sono gli oggetti `NSWindowsController` associati al documenti. Per come ho costruito la cosa, ce ne possono essere uno oppure due. Se ce ne sono due, sono aperte entrambe le finestre desiderate, e quindi faccio nulla. Se è presente una sola finestra, allora passo a costruirne una nuova, utilizzando la classe `CoverWinCtl`. Da notare l'ultima istruzione del metodo, che serve a portare davanti a tutte le altre finestra la finestra appena costruita.

Val la pena di commentare la frase '*per come ho costruito la cosa*'. Alla creazione di un documento nuovo, è aperta automaticamente una finestra governata dalla classe `ListWinCtl`. Questa rimane l'unica finestra per tutta la vita del documento, a meno che non sia invocato il metodo `showCoversWindow`. In questo caso è costruita una nuova finestra, governata dalla classe `CoverWinCtl`. Se l'utente chiude la finestra `CoverWinCtl`, finestra e controllore sono distrutti, e nulla altro accade: il documento e la finestra `ListWinCtl` rimangono aperti. Se invece l'utente chiude la finestra `ListWinCtl`, sono chiusi il documento e tutte le finestre relative (è questo l'effetto del metodo `shouldCloseDocument` sopra discusso), compresa, se presente, la finestra `CoverWinCtl`. Effetto collaterale di questa costruzione è lo sheet di salvataggio file, che è sempre attaccato alla finestra `ListWinCtl`, anche se il comando di salvataggio è inviato al documento con la finestra `CoverWinCtl` di fronte a tutti.

Capitolo 037 - Chiare e fresche sorgenti

Continuo dai capitoli precedenti: qui si parla della sorgente dei dati.

Leggo la documentazione...

Primo inserimento: 5 gennaio 2004

LA SORGENTE DATI

La classe sorgente di dati è il cuore di tutte le rappresentazioni dei dati all'interno delle finestre. Ho definito una nuova classe `CatDataSrc`, erede della precedente `LSDataSource`. Le variabili d'istanza non sono molto cambiate:

```
@interface CatDataSrc : NSObject {
    // elenco dei volumi
    NSMutableArray    * startPoint ;
    // criterio di ordine
    NSString          * orderColumn ;
    // direzione
    BOOL              orderDirection ;
    // il documento cui fa riferimento
    CdCatDoc          * refDoc ;
    // volume correntemente selezionato
    int                 currentVol ;
}
```

C'è in più una variabile intera che tiene conto del volume correntemente selezionato nella finestra `CoverWinCtl`; il suo utilizzo è spiegato successivamente.

I metodi per questa classe sono numerosi, e divisibili in vari gruppi. Il primo gruppo contiene i metodi per la gestione della struttura dati:

```
- (id)         init ;
- (void)     dealloc ;
- (void)     addFileEntry: (id) newEntry ;
- (void)     removeEntry: (FileStruct *) newEntry ;
```

I primi tre non sono molto diversi dalla realizzazione precedente; l'ultimo metodo è stato introdotto per semplificare al resto dell'applicazione il processo di eliminazione di un elemento. A parte le ultime due istruzioni (che saranno chiare più avanti), serve sostanzialmente ad isolare la funzione `deleteItemFromHierarchy` che, essendo chiamata ricorsivamente, conviene non sia pensata come metodo.

```
- (void )
removeEntry: (FileStruct *) fInfo
{
    //considero ricorsivamente gli start points e vado
    deleteItemFromHierarchy( startPoint, fInfo );
    // ovviamente, il documento e' stato sporcato
```

```
    [ refDoc updateChangeCount: NSChangeDone ] ;
    // rinfresco il contenuto delle finestre
    [ refDoc refreshWinsWithListSts: nil coverSts: nil ];
}
```

Il secondo gruppo di metodi gestisce la tableView della finestra `CoverWinCtl`:

```
// per la visualizzazione della lista dei volumi
- (int)        numberOfRowsInTableView:(NSTableView *)tv;
-     (id)                                    tableView:(NSTableView     *)tv
objectValueForTableColumn:(NSTableColumn *)tc row:(int)row;
-       (BOOL)                                tableView:(NSTableView       *)tv
shouldEditTableColumn:(NSTableColumn *)tc row:(int)row ;
// metodi delegati
- (void)    tableViewSelectionDidChange:(NSNotification *)aNotification
```

I primi tre metodi sono quelli chiamati in accordo alla gestione della sorgente di dati; il quarto metodo invece fa parte dei metodi che possono essere realizzati dalla classe delegata. Pur non essendo elegante metterli tutti assieme, ho preferito accorparli qui per averli sottomano tutti assieme, ben consapevole della loro differenza.

Il terzo gruppo di metodi gestisce le due outlineView, una della finestra `ListWinCtl` e l'altra della finestra `CoverWinCtl`.

```
// per la visualizzazione della lista dei file
-    (int)                outlineView:    (NSOutlineView    *)    outlineView
numberOfChildrenOfItem: (id) item;
- (BOOL)outlineView: (NSOutlineView *) outlineView isItemExpandable: (id)
item ;
- (id)      outlineView: (NSOutlineView *) outlineView child: (int) index
ofItem: (id) item ;
-    (id)                outlineView:    (NSOutlineView    *)    outlineView
objectValueForTableColumn: (NSTableColumn *) tableColumn byItem: (id) item;
-       (void)outlineView:        (NSOutlineView       *)        outlineView
setObjectValue:(id)object    forTableColumn:(NSTableColumn    *)tableColumn
byItem:(id)item ;
// metodi delegati
-         (BOOL)outlineView:          (NSOutlineView         *)          oView
shouldSelectTableColumn:(NSTableColumn *)tableColumn;
-   (BOOL)outlineView:   (NSOutlineView   *)   oView   shouldEditTableColumn:
(NSTableColumn *) tableColumn item: (id) item ;
-  (void)outlineView:  (NSOutlineView  *)  oView  willDisplayCell:(id)aCell
forTableColumn:(NSTableColumn *)aTableColumn item:(id)item ;
- (void)outlineViewSelectionDidChange:(NSNotification *)notification
```

Anche qui, i primi cinque metodi sono propri di una sorgente dati, mentre gli altri quattro sono della classe delegata.

LA LISTA DEI VOLUMI

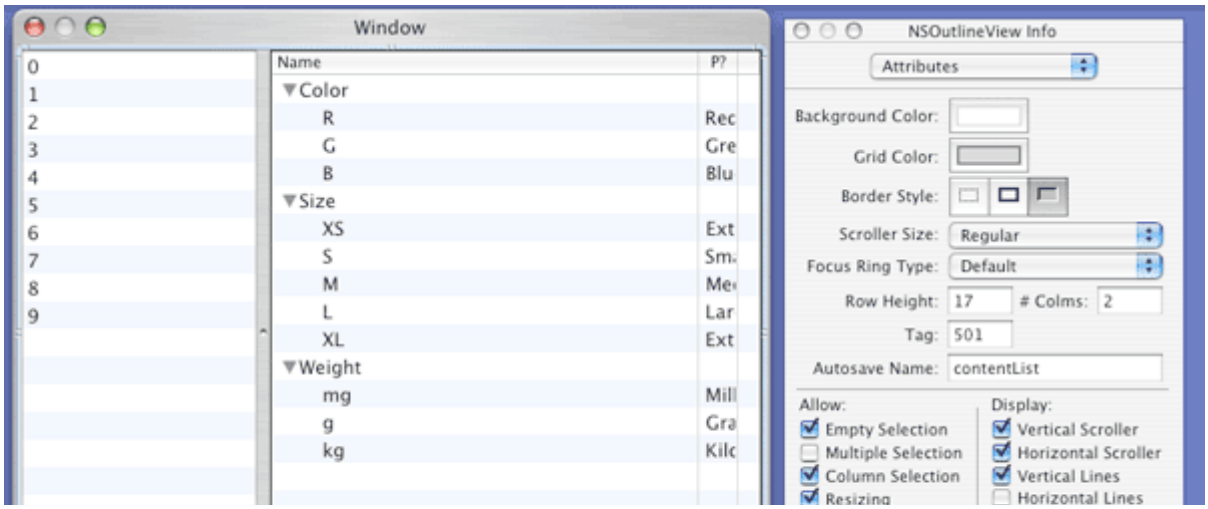
La gestione della tableView della finestra `CoverWinCtl` è molto semplice e lineare. Ciò che deve essere visualizzato è in pratica il vettore `startPoint`. L'unico metodo a presentare qualcosa di interessante è il quarto (delegato).

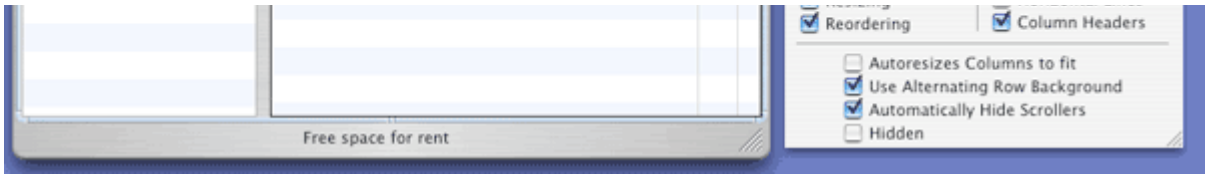
```
- (int)        numberOfRowsInTableView:(NSTableView *)tv
```

```
{
    // numero di volumi
    return ( [ startPoint count]);
}

-    (id)                                    tableView:(NSTableView     *)tv
objectValueForTableColumn:(NSTableColumn *)tc row:(int)row
{
    // c'e' sempre e solo una sola colonna
    // predispongo l'immagine da mostrare
     [ [tc dataCell] setImage: [[startPoint objectAtIndex: row] fileIcon ]
];
    // restituisco il volume i-esimo
    return ( [ [startPoint objectAtIndex: row ] fileName ]);
}

-       (BOOL)                                tableView:(NSTableView       *)tv
shouldEditTableColumn:(NSTableColumn *)tc row:(int)row
{
    return ( NO );
}

- (void)
tableViewSelectionDidChange:(NSNotification *)aNotification
{
    NSTableView        * ov = (NSTableView*)[ aNotification object ] ;
    // e' cambiata la selezione
    currentVol = [ ov selectedRow ] ;
    // rinfresco le finestra
    [ refDoc refreshWinsWithListSts: nil coverSts: @"aloha" ];
}
```

Il metodo `tableViewSelectionDidChange:` è invocato tramite una notifica alla classe delegata. La notifica avviene quando cambia la selezione all'interno della tableView. Anche se la notifica è possibile solo a partire da una sola tableView, ho voluto esplicitamente ricavare l'oggetto della notifica (la `NSTableView` la cui selezione è mutata) come esercizio. Da questo oggetto ricavo l'indice della riga selezionata (che è -1 nel caso in cui non ci siano righe selezionate), e lo assegno alla variabile d'istanza `currentVol`. Questa variabile sarà usata per visualizzare correttamente la outlineView della finestra `CoverWinCtl`.

DUE LISTE UNA SORGENTE





Il problema di utilizzare una unica sorgente dati per visualizzare elementi all'interno di due distinte outlineView (distinte anche per come è organizzato il contenuto) sta nel riconoscere univocamente la outlineView che invoca i metodi. Il meccanismo è molto semplice: attribuisco alle due outlineView un codice di tag differente. Occorre quindi usare Interface Builder per assegnare due valori di tag differenti alle due outlineView. Poi, all'interno dei metodi, la prima istruzione eseguita discrimina proprio questo codice.

Se il codice di tag è caratteristico della outlineView della finestra `ListWinCtl`, si utilizza essenzialmente il codice della versione precedente. Se invece occorre fornire i dati della outlineView della finestra `CoverWinCtl`, ci sono due situazioni di cui tenere conto.

Innanzitutto, occorre verificare se c'è un elemento selezionato (un volume) nella tableView. È questo il compito della variabile `currentVol`, predisposta come visto in precedenza. Se non ci sono volumi selezionati, la outlineView rimane vuota. Se c'è un elemento selezionato, allora bisogna saltare il primo livello (l'elenco dei volumi) e passare al primo livello del file selezionato. Ho adattato questa tecnica cercando di unificare la trattazione il più possibile, evitando troppi distinguo. Ne è venuto fuori una cosa di cui non sono pienamente soddisfatto, ma che per il momento tengo così.

Vado per ordine.

```
- (int)
outlineView:            (NSOutlineView *)outlineView
    numberOfChildrenOfItem:    (id)item
{
    VolInfo * localItem ;
    // se sto visualizzando la outlineView della CoverCtlWin
    if ( [ outlineView tag] == TAG_CDCATDOC_FILELIST )
    {
        // se non c'e' alcuna selezione, devo mostrare nulla
        if ( currentVol == -1 )
            return ( 0 ) ;
        // se arrivo qui, c'e' un elemento selezionato
        // se l'item e' nil, stiamo parlando della radice
        if (item == nil)
            // salto di un livello proponendo il volume come radice
            localItem = [ startPoint objectAtIndex: currentVol ];
        // altrimenti e' un elemento standard
        else localItem = item ;
        // da qui si prosegue normalmente
    }
    else if ( [ outlineView tag] == TAG_CDCATDOC_FULLLIST )
    {
        // se l'item e' nil, stiamo parlando della radice
        if (item == nil)
             // dico quindi che ci sono tanti elementi quanti presenti nel
vettore
            return( [ [self startPoint] count ]);
        // imposto l'elemento radice come quello corrente
```

```
        localItem = item ;
    }
    // da qui in poi, la trattazione e' la stessa
    // se arrivo qui, mi si chiede quali figli ha un file
    // tratto subito i file normali, che non hanno figli
    if ( [localItem numOfFiles] == 0 )
        return ( 0 ) ;
    // se arrivo qui, ho una directory
    // se devo mostrare anche i dotfiles, conto tutto
    if ( [[ UserPrefs getPrefValue: keyShowDotFiles] boolValue] )
        return ( [localItem numOfFiles] ) ;
    // se arrivo qui, devo eliminare dal computo i dotFiles
    // ed alora, mi tocca esaminare tutti i file e contare quelli
    // che mi interessano
    return ( countNormalFiles( localItem ) ) ;
}
```

Per capire quanti *figli* ha un elemento, distinguo subito le due outlineView. Se sto procurando i dati per la finestra `CoverWinCtl`, mi chiedo subito se c'è un elemento selezionato. Se così non è, devo visualizzare nulla. Se c'è un elemento selezionato, allora dovrò visualizzare l'elemento `item`. Se però `item` è `nil`, significa che mi sto chiedendo quanti elementi ci sono al livello più alto della outlineView. La tecnica è di saltare un livello, impostando il valore della variabile `localItem` con `item` stesso se non è `nil`, oppure con il volume correntemente selezionato. L'uso della variabile `localItem` procura una modifica anche nella parte relativa alla outline della finestra `ListWinCtl`: imposto il valore corrente di `item` se questo non è `nil`. Da qui in poi, le istruzioni proseguono come nella versione precedente.

Un po' più semplice il metodo in cui ci si chiede se l'elemento è espandile (se ha *figli*).

```
- (BOOL)
outlineView:            (NSOutlineView *)outlineView
    isItemExpandable:    (id)item
{
    // se sto visualizzando la outlineView della CoverCtlWin
    if ( [ outlineView tag] == TAG_CDCATDOC_FILELIST )
    {
        // se non c'e' alcuna selezione, devo mostrare nulla
        if ( currentVol == -1 )
            return ( NO ) ;
        // se arrivo qui, c'e' un elemento selezionato
        if (item == nil)
        {
            VolInfo * curVol ;
            // mi si chiede se si puo' espandere.. dipende da
            // quanti elementi ci sono all'interno del volume
            // (che succede se non ci sono elementi?)
            curVol = [ startPoint objectAtIndex: currentVol ];
            return ( ([curVol numOfFiles] == 0) ? NO : YES );
        }
        // altrimenti, e' un elemento standard
    }
    else if ( [ outlineView tag] == TAG_CDCATDOC_FULLLIST )
    {
        if (item == nil)
            return ( YES ) ;
    }
    // se non ha figli, e' un file, non si espande
    if ( [item numOfFiles] == 0 )
```

```
        return ( NO ) ;
    // se arrivo qui, l'item ha figli
    // se devo espandere anche i bundle, espando tutto
    if ( [[ UserPrefs getPrefValue:keyExpandBundle] boolValue] )
        return ( YES ) ;
    // se arrivo qui, non devo espandere i bundle
    // espando allora i non bundle
    return (! checkIfBundleDirectory([ item fileName ]) ) ;
}
```

La risposta è certamente `NO` in assenza di volumi selezionati nella finestra `CoverWinCtl`. Nel caso di elemento `nil`, la risposta dipende dal numero di elemento presenti all'interno del volume (e sarà quasi certamente `YES`). In tutti gli altri casi, l'elemento non subisce un trattamento differente dal caso standard.

Sulla stessa falsariga il metodo per rintracciare il figlio i-esimo di un elemento.

```
- (id)
outlineView:        (NSOutlineView *) outlineView
    child:        (int) index
    ofItem:        (id) item
{
    VolInfo * localItem ;
    // se sto visualizzando la outlineView della CoverCtlWin
    if ( [ outlineView tag] == TAG_CDCATDOC_FILELIST )
    {
        // se non c'e' alcuna selezione, devo dire nulla
        if ( currentVol == -1 )
            return ( nil ) ;
        // c'e' un elemento selezionato
        if (item == nil)
                localItem = [ startPoint objectAtIndex: currentVol ];
        else    localItem = item ;
    }
    else if ( [ outlineView tag] == TAG_CDCATDOC_FULLLIST )
    {
        if (item == nil)
        {
            NSArray            * newArray ;
            NSMutableArray * tmpArray ;
            // ordino gli elementi secondo criterio
            if ( [ self orderDirection ] )
                                        newArray  =  [  [self  startPoint]
sortedArrayUsingFunction: ascSortFunc
                        context: [ self orderColumn]] ;
                           else         newArray  =  [  [self  startPoint]
sortedArrayUsingFunction: desSortFunc
                        context: [ self orderColumn]] ;
            // trasformo il risultato in un NSMutableArray
                tmpArray = [ NSMutableArray arrayWithCapacity: [ newArray
count]];
            [ tmpArray addObjectsFromArray: newArray ];
            return( [ tmpArray objectAtIndex: index ]);
        }
        else localItem = item ;
    }
    // negli altri casi, recupero la lista
       [ localItem sortFileList: [ self orderColumn] direction: [ self
orderDirection ] ];
    // se visualizzo anche i dotFiles, non c'e' problema
    if ( [[ UserPrefs getPrefValue:keyShowDotFiles] boolValue] )
```

```
        return( [localItem getFileAtIndex:index]);
    // altrimenti, e' un bel pasticcio, devo saltare i dotfiles
    return ( getNormalFile ( localItem, index ) );
}
```

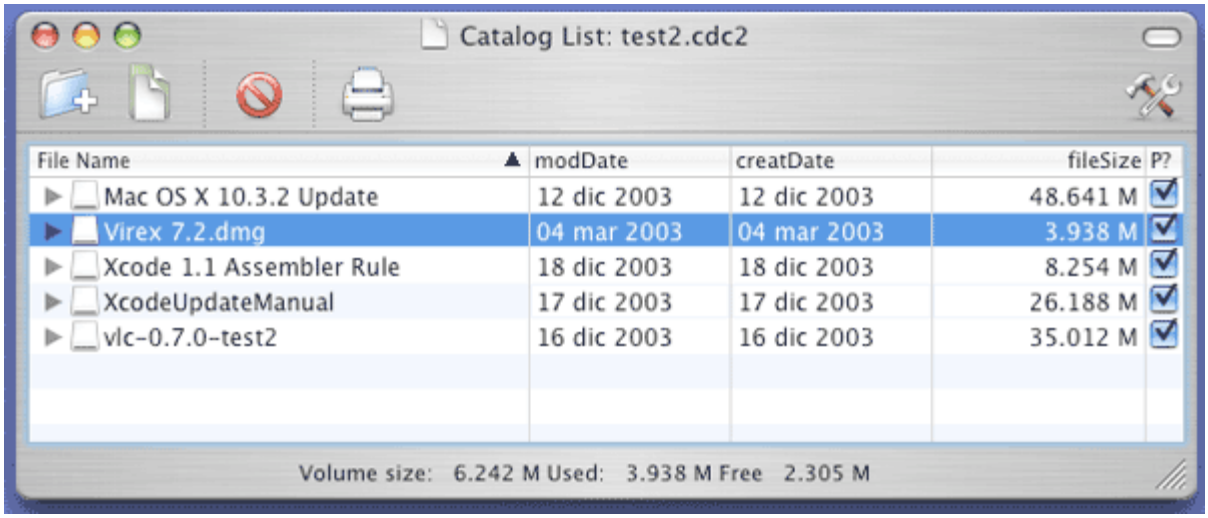
C'è qualcosa di diverso dalla soluzione precedente quando, nella outlineView della finestra `CoverWinCtl`, ci si chiede il figlio i-esimo dell'elemento `nil`, cioè il livello superiore. In questo caso l'elemento `nil` è sostituito dal volume ed il metodo procede come per tutti gli elementi standard.

C'è da notare una cosa: poiché la sorgente di dati è unica, l'ordinamento imposto dalla finestra `ListWinCtl` si riflette anche sulla finestra `CoverWinCtl`. C'è una significativa eccezione, che è la lista dei Volumi; la finestra `CoverWinCtl` presenta (nella tableView) i volumi nell'ordine in cui sono stati inseriti all'interno del documento. In effetti, mentre l'elenco dei file è riordinato in loco dal metodo `sortFileList:direction:`, il vettore `startPoint` non è mai rimpiazzato da una sua versione ordinata. In questo modo è possibile, per la visualizzazione nella finestra `CoverWinCtl`, rintracciare il volume utilizzando semplicemente il metodo `objectAtIndex:`.

Gli altri metodi (e funzioni) della sorgente dati non hanno subito modifiche particolari. è invece interessante valutare il metodo attivato ad ogni cambio di selezione nella outlineView.

```
- (void)
outlineViewSelectionDidChange:(NSNotification *)notification
{
    FileStruct        * currsel ;
    NSString        * statusStr ;
    NSOutlineView * ov ;
    // se sto visualizzando la outlineView della ListWinCtl
    ov = (NSOutlineView*)[ notification object ] ;
    if ( [ ov tag] == TAG_CDCATDOC_FULLLIST )
    {
        // e' cambiata la selezione
        int cs = [ ov selectedRow ] ;
        if ( cs == -1 )
        {
            // imposto il testo della barra di stato
                   statusStr  =  [  NSString  stringWithString:  @@"No  Volume
selected@"];
            [ refDoc refreshWinsWithListSts: statusStr coverSts: nil ];
            return ;
        }
        // se arrivo qui, c'e' qualcosa di selezionato
        currsel = [ ov itemAtRow: cs ];
        // se l'elemento selezionato e' uno dei volumi
        if ([ startPoint indexOfObject: currsel] != NSNotFound)
        {
            // imposto la stringa con info utili
            unsigned long long d1, d2, d3 ;
            d1 = [ currsel volSize];
            d2 = [ currsel volFreeSize] ;
            d3 = [ currsel fileSize] ;
               statusStr = [ NSString stringWithFormat: @"Volume size: %@
Used: %@ Free %@",
                     fileSizeFromLongLong( d1), fileSizeFromLongLong( d3),
fileSizeFromLongLong(d2) ];
        }
```

```
        else    statusStr = [ NSString stringWithFormat: @"Full path: %@",
[ currsel fileFullPath]];
        // rinfresco le finestre associate alla dataSource
        [ refDoc refreshWinsWithListSts: statusStr coverSts: nil ];
    }
}
```



Qui la cosa è interessante (per il momento) solo nel caso della finestra `ListWinCtl`. Ad ogni modifica di selezione, rintraccio la riga, e dalla riga l'elemento; in base all'elemento aggiorno adeguatamente la barra di stato. Se l'elemento è un volume, ne dico dimensione e spazio impegnato; se è un normale file, ne riporto il nome (giusto perché non avevo altre idee).

La funzione `fileSizeFromLongLong` ricorda un po' un metodo della classe dei formattatori:

```
NSString *
fileSizeFromLongLong(unsigned long long fSize )
{
    unsigned long long    fSizeK ;
    float             fsizeM, fsizeG ;

    // se il file e' piccolo, mostro byte
    if ( fSize < 1024 )
    {
        long    tmp = fSize ;
        return ( [ NSString stringWithFormat: @@" %4d b@", tmp] );
    }
    // la dimensione e' in byte, divido per 1024 ed arrotondo, ottengo K
    fSizeK = (long) (( fSize / 1024.0 ) + 0.5 );
    // se il file e' medio, mostro K
    if ( fSizeK < 1024 )
    {
        long    tmp = fSizeK ;
        return ( [ NSString stringWithFormat: @@" %4d K@", tmp] );
    }
    // se inferiore al giga, restituisco mega con tre decimali
    fsizeM = ( fSizeK / 1024.0 ) ;
    if ( fsizeM < 1024 )
        return ( [ NSString stringWithFormat: @@" %6.3f M@", fsizeM] );
    // negli altri casi, ritorno Giga, con tre cifre decimali
    fsizeG = ( fsizeM / 1024.0 ) ;
```

```
    return ( [ NSString stringWithFormat: @@" %6.3f G@", fsizeG] );
}
```

Per chiudere la questione, bisogna parlare del metodo `refreshWinsWithListSts:` che è comparso qui e lì nel codice finora visto. Ebbene, c'è il problema di sincronizzare l'aspetto delle due finestre perché rispecchino entrambe la stessa situazione della sorgente dati. In particolare questo significa che l'aggiornamento della visualizzazione deve avvenire più o meno assieme, in modo da non avere situazioni difformi tra le due finestre. Per fare questo mi sono inventato un metodo del documento, appunto quello citato, che forza l'aggiornamento.

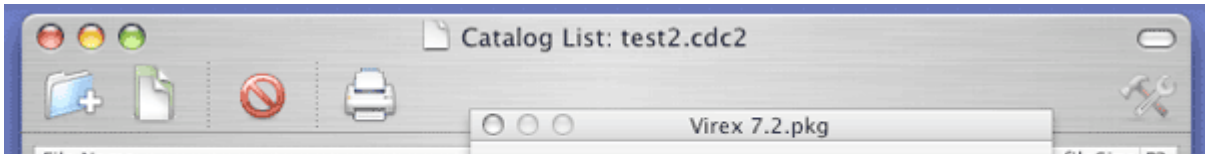
```
- (void)
refreshWinsWithListSts: (NSString*) sts1Bar coverSts: (NSString*) sts2Bar
{
    // recupero tutti i controllori di finestre del documento
       NSEnumerator          *  enumerator  =  [[  self  windowControllers  ]
objectEnumerator];
    NSWindowController * winCtl ;

    winCtl = [enumerator nextObject] ;
    // finche' ci sono controllori
    while ( winCtl )
    {
        // dico di aggiornare il contenuto della finestra
        [ winCtl refreshWindow ] ;
        // ed anche la barra di stato
        [ winCtl setStsStrings: sts1Bar andSts: sts2Bar] ;
        // passo al controllore successivo
        winCtl = [enumerator nextObject] ;
    }
}
```

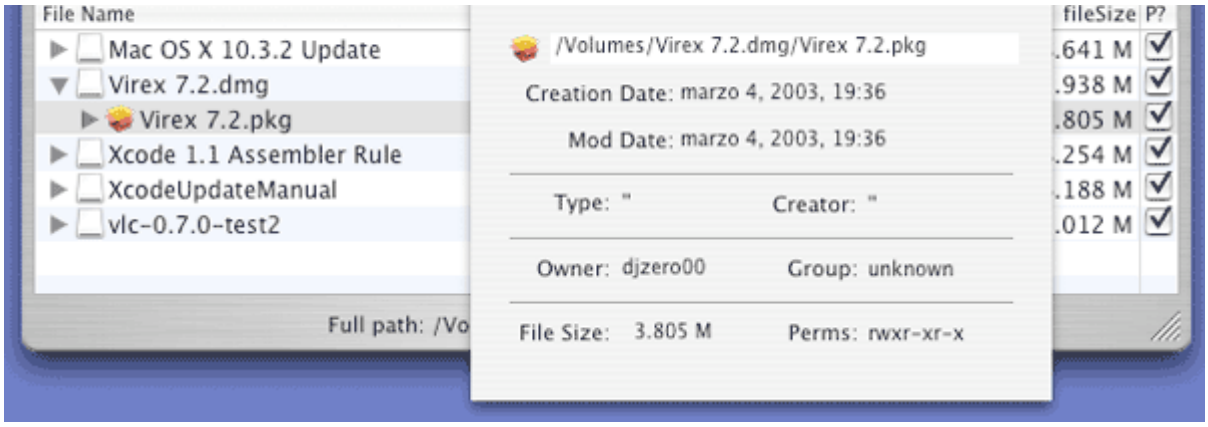
In pratica, piglia l'elenco degli oggetti `NSWindowController` e a ciascuno invia gli stessi messaggi. Ogni controllore di finestra realizza in maniera opportuna, ciascuno per la propria competenza, i metodi corrispondenti; in questo modo, sfruttando il polimorfismo, ogni finestra si aggiorna da sé. Ad esempio, per la finestra `CoverWinCtl`, il metodo sarà così realizzato:

```
- (void)
refreshWindow
{
    [ volList reloadData ];
    [ fileList reloadData ];
}
```

Il metodo `setStsStrings:andSts:` vuole invece provare ad aggiornare separatamente le due barre di stato delle due finestre, ma per il momento non è un gran vedere.

PANNELLO INFORMAZIONI





Per concludere, occorre notare un paio di modifiche nella gestione del pannello delle informazioni. A parte la ristrutturazione della finestra con l'eliminazione dei dati non più visualizzati, sono da notare i tre metodi attivati dalle notifiche di cambiamento selezione, o meglio, il solo metodo seguente:

```
- (void )
mainWindowChanged: ( NSNotification *) notification
{
    NSWindowController *winCtl ;
    winCtl = [[ notification object] windowController] ;
    [ self updateInfo: [ winCtl getMainView ] ];
}
```

Il metodo `updateInfo:`, per funzionare correttamente, ha bisogno di conoscere la outlineView di riferimento. Questa, ovviamente, cambia, quando cambio la finestra; tuttavia, la nuova finestra può essere `ListWinCtl`, oppure `CoverWinCtl`.

Nel dubbio, lascio fare al controllore della finestra sfruttando il polimorfismo: il metodo `getMainView` non fa altro che restituire, finestra per finestra, la outlineView corretta. Ad esempio, per la `ListWinCtl`:

```
- (NSOutlineView*) getMainView
{
    return ( fullList );
}
```

Capitolo 038 - Ho una vista

Questo capitolo, piuttosto discorsivo, introduce l'argomento viste, ovvero la classe `NSView` e le sue sottoclassi, punto di partenza per disegnare qualcosa sullo schermo.

Leggo la documentazione...

Primo inserimento: 8 gennaio 2004

Quattro Classi Basilari

Una applicazione è, nella sua essenza, costruita da tre classi di oggetti: `NSApplication`, `NSWindow` e `NSView`. Questi classi sono tutte figlie della classe `NSResponder` (a sua volta derivata da `NSObject`). Queste quattro classi sono in grado di fornire le funzioni di base necessarie al funzionamento di una applicazione, alla rappresentazione a schermo di elementi di interfaccia e per l'interazione dell'utente.

La classe `NSResponder` è in grado di gestire gli eventi (clic del mouse, caratteri immessi da tastiera, eccetera). Fornisce i metodi di base per la gestione degli eventi ed il meccanismo con cui gli eventi sono distribuiti ai vari oggetti costituenti l'applicazione.

Ogni applicazione possiede uno ed uno solo oggetto della classe `NSApplication`: è quello costruito dalla funzione `NSApplicationMain` chiamata all'interno del file `main.m`. L'oggetto supervisiona e coordina il lavoro complessivo dell'applicazione. Distribuisce ogni evento alla finestra alla quale pertiene, gestisce le finestre, i documenti, eccetera.

Le attività di una applicazione si svolgono normalmente all'interno di una finestra, oggetti di classe `NSWindow`. Un oggetto `NSWindow` gestisce una finestra sullo schermo, o meglio il contenuto ed il comportamento della finestra: discegna il contenuto e risponde alle azioni di spostamento, chiusura, ridimensionamento ed altre manipolazioni. Scopo principale di una finestra è di mostrare l'interfaccia utente (o parte di essa) al suo interno.

Arrivo finalmente agli oggetti `NSView`. Ogni oggetto che si può vedere all'interno di una finestra è una istanza di una sua sottoclasse (raramente un oggetto `NSView` è utilizzato così come si trova). Ogni `NSView` è rappresentato all'interno di una regione rettangolare, ed è responsabile di come sia visualizzata e di come interagisce con l'utente. Ad esempio un pulsante (`NSButton`) è (alla fin fine) una sottoclasse di `NSView`, in grado di rappresentarsi secondo l'iconografia classica di un pulsante, e di interagire con il mouse (o la tastiera) nella modalità cui siamo abituati.

Tutti gli oggetti `NSView` sono organizzati all'interno di una finestra secondo una gerarchia ad albero. Ogni view è dunque racchiusa all'interno di una view di livello superiore (*superview*) e possiede al suo interno zero, una o più view (*subview*). Ad esempio, nella finestra `ListWinCtl` la view principale (la *content view* della finestra) possiede come subview la barra di stato, e la `NSOutlineView`. A sua volta, la `NSOutlineView` possiede come subview le barre di scorrimento, la view d'angolo, le

colonne, eccetera.

Ogni view all'interno della gerarchia è rappresentata all'interno della sua superview. Dato il rettangolo della superview, ogni sua subview può essere rappresentata solo all'interno di questo rettangolo. Associata ad ogni view ci sono due rettangoli: il primo rettangolo, *frame*, determina la posizione della view all'interno della sua subview; il secondo rettangolo, *bounds*, determina la geometria proprio della view. In altre parole, il frame è utilizzato dalla superview per riservare lo spazio alla view, mentre la view rappresenta se stessa all'interno del rettangolo bounds.

Le view rappresentano se stesse come risposta indiretta al messaggio `display` (o simili), generalmente gestito automaticamente dall'ambiente operativo. Il messaggio porta all'esecuzione del metodo `drawRect:` per la view e, in successione, per tutte le subview contenute. All'interno di questo metodo devono trovarsi tutte le istruzioni per il disegno a video del contenuto della view.

COORDINATE

Per il posizionamento corretto di finestre e view, occorre un sistema di coordinate, che permette di disporre nei posti giusti i rettangoli frame. Cocoa utilizza diversi sistemi di coordinate: per lo schermo, per le finestre, per le viste. In realtà, seguono la stessa logica.

Si recupera il concetto di coordinate cartesiane. Occorre specificare una origine, i due assi cartesiani, e le unità di misura. Nel caso delle coordinate per lo schermo, l'origine è il punto in basso a sinistra dello schermo. L'asse delle ascisse (orizzontale) va verso destra, l'asse delle ordinate (verticale) va verso l'alto. L'unità di misura è il pixel. Tuttavia, essendo le coordinate espresse come numeri floating point, si possono utilizzare anche dimensioni frazionarie, lasciando al motore di visualizzazione gli arrotondamenti per la visualizzazione sullo schermo. Queste coordinate sono utilizzate per posizionare le finestre sullo schermo.

Le coordinate per le finestre sono molto simili: l'origine è il punto in basso a sinistra della finestra, gli assi cartesiani vanno verso destra e verso l'alto. Queste coordinate rimangono attaccate alla finestra, dovunque questa finestra sia spostata. In altre parole, le coordinate degli oggetti (`NSView`) all'interno di una finestra rimangono sempre le stesse, anche se la finestra è spostata e le posizioni assolute di questi oggetti nello schermo è cambiata.

Le coordinate per le view funzionano come per quelle delle finestre: hanno origine nel punto in basso a sinistra della view, e gli assi viaggiano normalmente verso destra e verso l'alto. Anche qui, le coordinate degli oggetti contenuti all'interno di una view rimangono fissi, anche se la view è spostata per qualsiasi motivo.

In definitiva, ogni view si porta appresso un proprio sistema di riferimento locale. Questo fatto ha una importante conseguenza sulle operazioni di disegno: le istruzioni di disegno rimangono sempre le stesse, trascurando ogni mutamento nella posizione della view o della catena di superview.

Ora che ho introdotto il concetto di coordinate, è più facile spiegare la differenza tra i rettangoli `frame` e `bounds`. Il rettangolo `frame` determina la posizione della view

all'interno della superview. Rappresenta quindi le coordinate del rettangolo della view espresse nel sistema di riferimento della superview. Il rettangolo `bounds` invece rappresenta la porzione potenzialmente visibile della view espressa nelle coordinate della view stessa (potenzialmente visibile, perché lo spazio potrebbe estendersi fuori della superview o essere coperto da altre subview).

La Scroll View

Una view che è spesso utilizzata è la `NSScrollView`. Una scrollview permette di visualizzare view piuttosto grandi all'interno di finestre più piccole. Ogni volta che in una finestra vedete delle barre di scorrimento, abbiamo a che fare con una scrollView (beh, quasi sempre). Una scrollView è una view che gestisce una collezione di subview. Ci sono due `NSScroller`, due viste nate per gestire una barra di scorrimento; è possibile prevedere anche uno o due righelli `NSRulerView` (o sottoclassi), utili per dimensionare il contenuto della view; tipicamente c'è una `NSClipView`, che gestisce quale porzione del documento è visualizzata (raramente si ha a che fare con questa view direttamente); finalmente, c'è la *document view*, ovvero la view effettiva che contiene ciò che si intende visualizzare. L'idea di base è molto semplice: La document view funziona come se non sapesse di essere visualizzata solo in parte (o meglio, potrebbe funzionare: spesso ragioni di efficienza richiedono che la document view sia consapevole della zona rappresentata); la clipView limita al rappresentazione all'area visibile, e la scrollView gestisce le eventuali barre di scorrimento ed i righelli.

Ho chiamato la view interna genericamente 'document view' perché questa view può essere di tipo qualsiasi. Ad esempio, sto scrivendo questo testo con un semplice editor di testi. La finestra del documento è una unica `NSScrollView`, all'interno della quale ci sono barre di scorrimento (una sola, verticale) ed un righello (uno solo, orizzontale). Il testo vero e proprio si trova all'interno di una `NSTextView`. Se l'applicazione fosse invece di disegno, la finestra sarebbe ancora una scrollView, ma il disegno vero e proprio si troverebbe in una view specifica per disegnare (che non c'è, quindi, bisogna che il programmatore se la costruisca).

Disegnare in una view

La visualizzazione del contenuto di una view avviene attraverso il metodo `drawRect:`. All'interno di questo metodo trovano posto tutte le istruzioni per il disegno, ovvero i comandi per Quartz. Infatti, per disegnare in una finestra non si devono pilotare direttamente i pixel di uno schermo, ma si scrivono una serie di comandi di disegno, specifici di Quartz; è poi compito dell'ambiente operativo trasformare questi comandi in effettive rappresentazioni a schermo. Ma questo non è vero; siamo in un ambiente OOP, quindi, si ha a che fare con una serie di classi, di oggetti e di messaggi per il disegno.

Prima di cominciare, occorre conoscere come rappresentare i dati. Ho già parlato del sistema di coordinate cartesiane; è quindi piuttosto ovvio che un punto sia un dato di tipo `NSPoint` (attenzione, non è un oggetto), costituito da due coordinate

```
typedef struct _NSPoint {
    float    x ;
```

```
    float y ;
}    NSPoint ;
```

Molti costrutti si basano poi sul concetto di dimensione, il dato `NSSize`:

```
typedef struct _NSSize {
    float    width ;
    float    height ;
}    NSSize ;
```

Ad esempio, un rettangolo è compiutamente specificato da un punto e dalla dimensione:

```
typedef struct _NSRect {
    NSPoint    origin ;
    NSSize    size ;
}    NSRect ;
```

Esistono poi funzioni (o macro; non sono metodi) di convenienza per costruire punti, dimensioni e rettangoli: `NSMakePoint`, `NSMakeSize`, `NSMakeRect`.

Ci sono fondamentalmente tre tipi di elementi che possono essere disegnati: elementi grafici quali linee, rettangoli, cerchi, eccetera; caratteri di testo; immagini complesse. Ogni elemento ha le sue istruzioni caratteristiche.

Per disegnare elementi grafici quali linee, rettangoli, archi, eccetera, si ricorre all'oggetto `NSBezierPath`. Per ogni elemento grafico ci costruisce un oggetto di questo tipo con metodi di convenienza; ad esempio, una volta definito un quadrato di cento pixel di lato, lo si può disegnare 'pieno' a partire dal punto (10,10) con l'istruzione

```
[ NSBezierPath fillRect: NSMakeRect( 10, 10, 100, 100) ];
```

Esiste poi una lunga serie di metodi per costruire percorsi a forma di linea, di arco, per costruire percorsi complessi, eccetera, che vedrò man mano che servono.

Per disegnare testo all'interno di una view, occorre in primo luogo avere un oggetto di tipo `NSString`. Poi occorre specificare carattere, dimensione e tutti gli altri attributi che si desiderano, immagazzinandoli tipicamente all'interno di un oggetto `NSMutableDictionary`. A questo punto si aprono due strade equivalenti; la prima è di costruire un oggetto `NSMutableAttributedString` a partire dalla stringa e dagli attributi, ed utilizzare questo oggetto come destinatario del messaggio `drawAtPoint:`. Alternativamente (e più semplicemente, mi pare), si usa direttamente la stringa ed un messaggio diverso:

```
[    miaStringa    drawAtPoint:    NSMakePoint(    10,10)    withAttributes:
miaStringaAttributi ];
```

Infine, per disegnare immagini sullo schermo, si parte da un oggetto di classe `NSImage`. Il processo utilizzato si chiama *compositing* e consiste nel sovrapporre in modo determinato due immagini: quella già presente sulla view e quella nuova che si intende disegnare. La determinazione del modo avviene secondo diverse modalità

(distruggendo la parte coperta, mischiando in modo opportuno, mantenendo la trasparenza, eccetera). Piuttosto che dilungarmi oltre, ritardo l'esame della cosa quando mi servirà.

PERCHÉ?

Perché ho scritto tutte queste cose? Finora, tutte le view utilizzate erano già prefabbricate: `NSButton`, `NSOutlineView`, `NSTableView`, eccetera, tutte queste view erano già complete e definite; si trattava solo di metterle assieme e di collegare tra loro i posti rimasti liberi. Adesso voglio avventurarmi nella costruzione di view particolari, non predefinite.

Capitolo 039 - Primi esperimenti con le viste

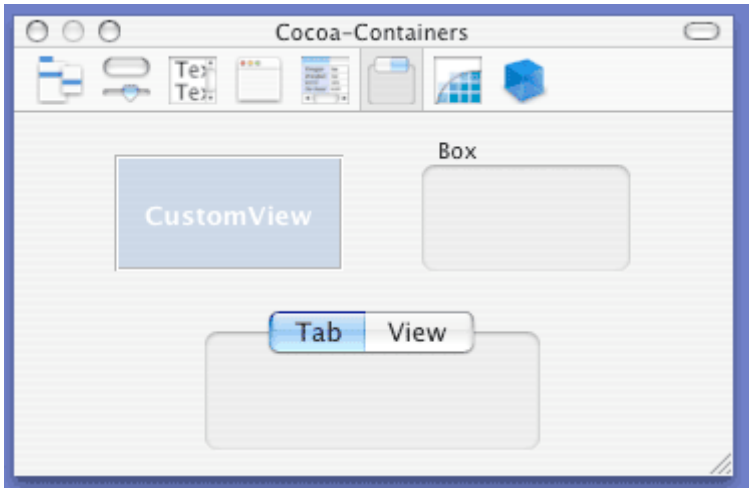
Comincio a fare alcuni esperimenti con le viste, ed in particolare con una vista proprietaria per progettare copertine di CD.

Leggo la documentazione...

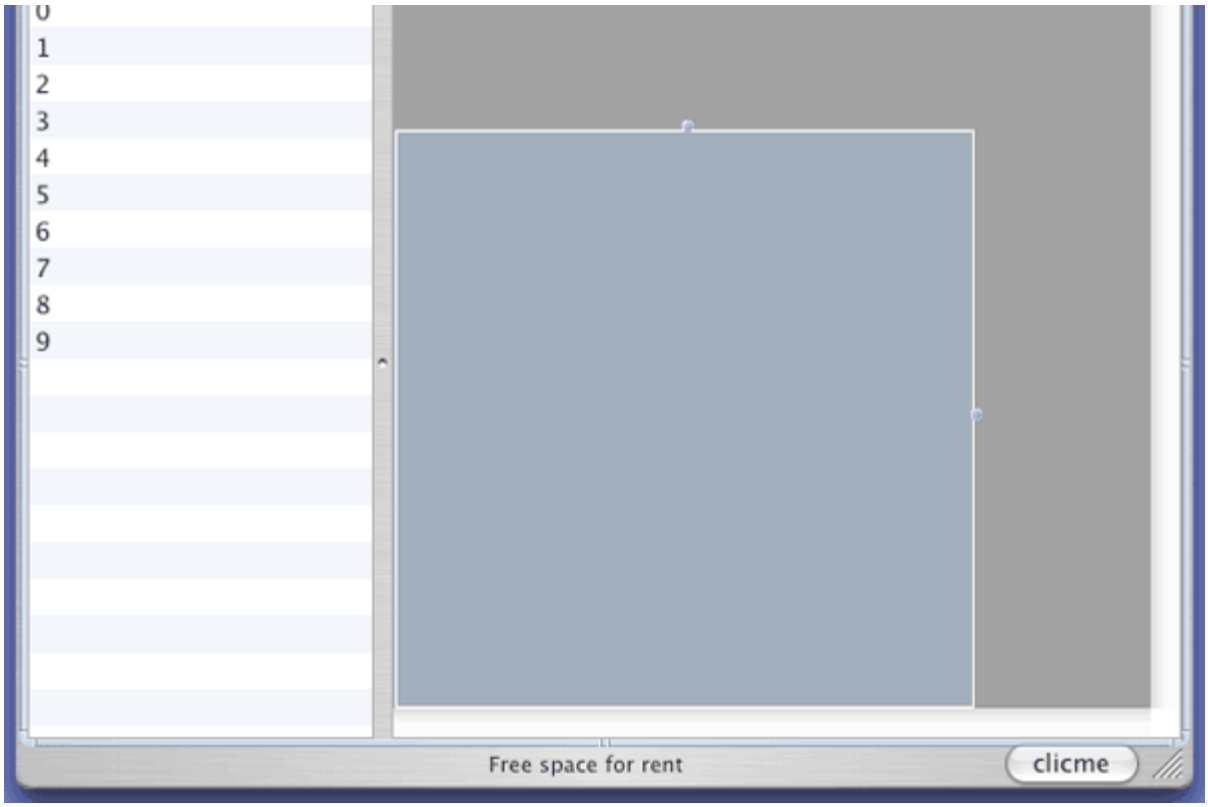
Primo inserimento: 9 gennaio 2004

UNA VISTA PARTICOLARE

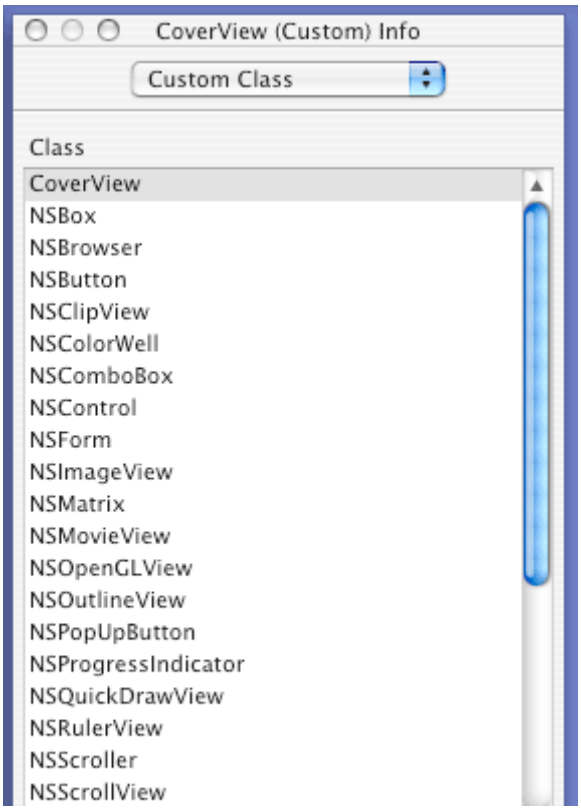
Se si deve visualizzare qualcosa che non appartiene all'ampio catalogo delle viste predefinite di Cocoa, occorre costruire una vista proprietaria (*custom view*). È questo il caso di molte applicazioni interessanti, ad esempio di disegno, vettoriale o meno; oppure applicazioni dove si manipolano elementi all'interno di una finestra, eccetera. Una custom view non è altro che una sottoclasse di `NSView` in cui sono riscritte le parti di visualizzazione ed eventualmente di interazione con l'utente. Una custom view può essere molto semplice, oppure estremamente complicata (si pensi alla finestra di disegno di Photoshop, cosa non deve saper gestire...). In ogni caso, un bravo (e bello) programmatore Cocoa deve essere in grado di costruire le proprie viste; è quello che comincio a fare.

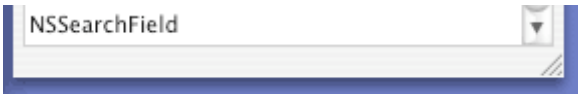


La cosa più semplice per cominciare è di utilizzare Interface Builder e piazzare dove necessario un oggetto `CustomView` (lo si trova predisposto nella palette degli oggetti). Nel mio caso, piglio il NIB `CoversWin.nib` relativo alla finestra `CoverWinCtl` e sostituisco la outlineView con una customView. In realtà il passaggio è duplice. Poiché all'interno della vista vorrò farci stare il disegno di una copertina di un CD, mi occorre un certo spazio su schermo che non è detto la finestra mi renda completamente disponibile. È giocoforza rendere la nuova customView una sottovista di una `NSScrollView`.

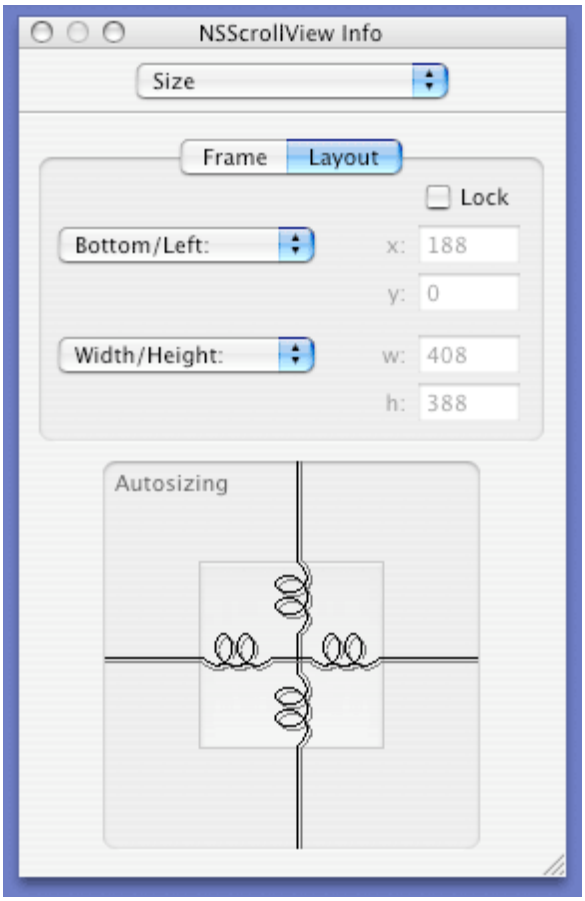


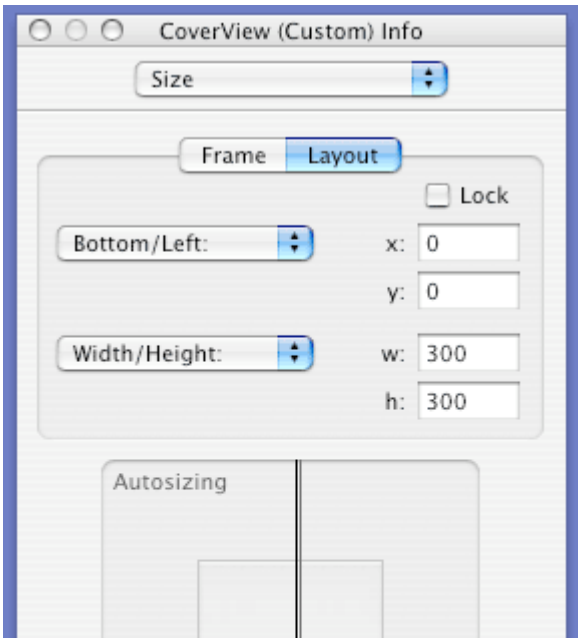
Ricordando la gerarchia delle viste discussa nel capitolo precedente, mi trovo ad avere una finestra `CoverWinCtl` che contiene due sottoviste: la barra di stato ed una `NSClipView`. La `NSClipView`, tra le altre, possiede due sottoviste: la tableView con l'elenco dei volumi e la `NSScrollView`. Questa contiene la `NSClipView`, eventuali barre di scorrimento laterali e righelli; la `NSClipView`, finalmente, contiene la customView.

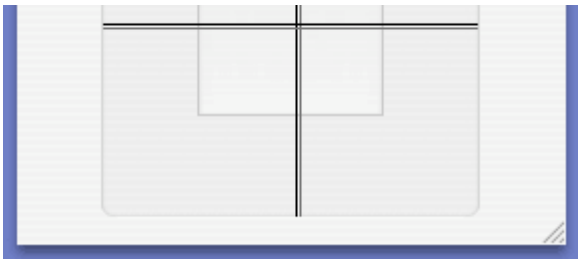




Tuttavia, non posso lasciare la customView in questo stato. È bene che dichiari una nuova sottoclasse di `NSView`, che chiamerò `CoverView`; forzo la customView ad essere un oggetto di questa classe. Si tratta di una operazione tipica per ogni customView: devo per ognuna dichiarare la classe di appartenenza; essendo appunto custom, molte volte sarà dichiarata al momento.







C'è una cosa cui deve essere posta la massima attenzione, pena la perdita di molto tempo alla ricerca dei motivi di comportamenti bizzarri: il dimensionamento della vista e la disposizione automatica sullo schermo. Ora, quando modifico le dimensioni della finestra, mi piacerebbe che la barra di stato rimanga della stessa dimensione verticale (ma orizzontalmente continua a coprire la maggior parte dello spazio disponibile), mentre la `NSSplitView` si adatti a coprire la maggior parte della superficie disponibile. All'interno della splitView la tableView e la scrollView dovrebbero spartirsi la spazio proporzionalmente. Il corretto funzionamento di tutto ciò è garantito dalla presenza delle molle e delle barre nelle caratteristiche di dimensionamento dell'oggetto relativo. All'inizio avevo pensato che anche la `CoverView` all'interno della scrollView dovesse avere le stesse caratteristiche. È evidente a tutti (a posteriori, ovviamente) che ciò non è vero: la `CoverView` si trova all'interno di una scrollView proprio perché la sua (della `CoverView`) dimensione non vuole dipendere dallo spazio che la scrollView mette a disposizione (oppure, con altre parole: perché l'accoppiata `NSScrollview`/`NSClipView` mette a disposizione un'apertura di dimensione limitata attraverso la quale vedere la più grande view interna). In definitiva, è fondamentale che le caratteristiche di dimensionamento della `CoverView` siano sempre fisse, ovvero siano sempre barre e mai molle, pena il mancato funzionamento della scrollView. Le dimensioni effettive della `CoverView`, sebbene impostabili da interfaccia, preferisco attribuirle da programma, alla creazione della vista stessa (dipenderanno dal tipo di copertina che si vuole disegnare).

Già che sono in Interface Builder, ho aggiunto un pulsante all'interfaccia giusto per avere un meccanismo per inviare messaggi in giro ai vari oggetti.

Completo le operazioni aggiungendo un po' di collegamenti fra i vari elementi presenti e costruendo i file per la classe `CoverView`.

LA COVERVIEW

Devo adesso scrivere i metodi per realizzare la `CoverView`. I metodi essenziali sono due:

```
- (id)initWithFrame:(NSRect)frameRect ;
- (void)drawRect:(NSRect)rect ;
```

Il primo è utilizzato per mettere a posto le cose la prima volta che è visualizzata la view:

```
- (id)initWithFrame:(NSRect)frameRect
{
    if ((self = [super initWithFrame:frameRect]) != nil)
    {
```

```
        // inizializzazioni particolari
    }
    return self;
}
```

Come si può vedere, per il momento è piuttosto scarna. è importante tuttavia la chiamata alla classe `super` tutte le inizializzazioni standard. L'argomento del metodo è il rettangolo `frame`, ovvero la posizione di questa vista all'interno della sua superview, espresso nelle coordinate della superview.

Il secondo metodo è invece il metodo fondamentale per la visualizzazione della `CoverView`. È bene racchiudere qui dentro ogni istruzione di disegno, dal momento che questo metodo è automaticamente chiamato da Cocoa quando la vista deve essere ridisegnata in seguito a qualche accadimento (ridimensionamento della finestra, spostamento delle barre di scorrimento, eccetera). L'argomento è il rettangolo che deve essere ridisegnato a video. Può quindi essere utilizzato per evitare di ridisegnare parti della vista che nessuno vedrà.

Ma prima di esaminare il codice di questo metodo, operazione cui è dedicato il resto del capitolo, occorre fare una premessa ed un passo indietro.

CONTENUTO DELLA COVERVIEW

Non ho ancora chiarito cosa intendo disegnare all'interno della `CoverView`. Per il momento mi limito a fare qualche esperimento. L'idea è comunque di avere un ambiente di progettazione grafica delle copertine di un CD; in questo ambiente una delle caratteristiche principali sarà quella di rappresentare l'elenco dei file presenti sotto forma della outlineView (limitatamente ai file visualizzati e con la caratteristica `is2Print` a vero).

In questi momenti iniziali, proverò tuttavia a disegnare il retro di una copertina di un CD e sperimentare un po' con il disegno di vari elementi grafici all'interno della view. Mi sono munito di righello, ed ho misurato una copertina, ottenendo alcune dimensioni che ho raccolto in una serie di `#define` (che evito di riportare qui, ma sono presenti nel file `CoverView.h`). Tutte le mie dimensioni sono in centimetri; poiché, a quanto pare, le coordinate di Quartz sono pixel, prima di ogni istruzione trasformo le unità centimetri in unità pixel utilizzando una macro.

```
#define         XFORM_FACTOR        28.346
#define         CM2PX( val )        ( val * XFORM_FACTOR)
```

In tutto ciò presuppongo che ci siano 72 pixel per pollice; con qualche altro fattore di conversione, si arriva a 28.346 pixel per centimetro. Evito di inserire dimensioni sotto forma numerica direttamente nel codice, ma uso piuttosto `#define` e macro proprio perché non sono sicuro dei valori e delle unità di conversione; se dovessero risultare sbagliate, basta correggere i numeri in pochi posti e tutto funzionerà a meraviglia.

Dato che uno degli elementi principali sarà una outlineView, ho provveduto ad aggiungere la variabile d'istanza `catView` alla `CoverView` di classe `NSOutlineView`. Per il momento decido che la outlineView sarà sempre presente, e creata all'inizio dei

tempi. Il problema della outlineView è che ha bisogno di molto supporto da altri oggetti dell'applicazione; può quindi essere costruita solo dopo che gli oggetti di cui necessita sono stati costruiti. Poiché la `CoverView` è costruita prima di altri, il posto più adatto per la sua costruzione è all'interno del metodo `windowDidLoad` di `CoverWinCtl`.

Tutto ciò giustifica la revisione di questo metodo come segue:

```
- (void) windowDidLoad
{
    NSTableColumn        * tableColumn1 = nil;
    NSTableColumn        * tableColumn2 = nil;
    ImageAndTextCell    * imageAndTextCell1 = nil;
    ImageAndTextCell    * imageAndTextCell2 = nil;
    CatDataSrc            * myDataSrc ;
    float                x1, x2, y1, y2 ;
    NSRect                outRect ;
    NSOutlineView        * catView = nil ;

    [super windowDidLoad ];
    // associo questa finestra alla sorgente dati
    myDataSrc = [ [self document] dataSource] ;
    [ volList setDelegate: myDataSrc ];
    [ volList setDataSource: myDataSrc ];
    // salvo nelle prefs le dimensioni delle colonne della outlineView
    [ volList setAutosaveName: @"VolumeViewOptions" ];
    tableColumn1 = [volList tableColumnWithIdentifier: @"volumeList"];
    // creo l'oggetto autorelease perche' passa in carico alla tableColumn
    imageAndTextCell1 = [[[ImageAndTextCell alloc] init] autorelease];
    [imageAndTextCell1 setEditable: NO];
    [tableColumn1 setDataCell:imageAndTextCell1];

    // metto a posto la dimensione della view della copertina
                    [cvrView     setFrameSize:NSMakeSize(CM2PX(VIEW_DIM_HOR),
CM2PX(VIEW_DIM_VEL))];

    x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST + CUTSIGN_DIM ;
    y1 = STARTPOINT_Y + CUTSIGN_DIM + CUTSIGN_DIST + CUTSIGN_DIM ;
    x2 = BACK_OUTER_HOR_DIM - 2 * CUTSIGN_DIM ;
    y2 = BACK_OUTER_VER_DIM - 2 * CUTSIGN_DIM ;
    outRect = NSMakeRect(CM2PX(x1), CM2PX(y1), CM2PX(x2), CM2PX(y2)) ;
    catView = [[ NSOutlineView alloc] initWithFrame: outRect ] ;
    [ catView autorelease ];
    // creo la NSTableColumn con l'identificatore
          tableColumn2   =   [   [NSTableColumn   alloc]   initWithIdentifier:
COLID_FILENAME ];
    imageAndTextCell2 = [[[ImageAndTextCell alloc] init] autorelease];
    [ tableColumn2 setDataCell:imageAndTextCell2];
    [ tableColumn2 setResizable: TRUE ];
    [ tableColumn2 setWidth: CM2PX(y2)] ;
    // imposto il titolo della colonna
       [  [  tableColumn2  headerCell  ]  setStringValue:  NSLocalizedString(
COLID_FILENAME, COLID_FILENAME ) ];
    [ catView addTableColumn: tableColumn2 ];
    [ catView setTag: TAG_CDCATDOC_FILELIST ];
    [ catView setDataSource: myDataSrc ];
    [ catView setDelegate: myDataSrc ];
    [ catView setUsesAlternatingRowBackgroundColors: TRUE ];
    [ catView setDrawsGrid: TRUE ];
    [ catView setBackgroundColor: [ NSColor redColor] ];

    [ cvrView setWinCtl: self ];
    [ cvrView setCatView: catView ];
```

```
    [ cvrView addSubview: catView ];
}
```

All'inizio ci sono le note istruzioni per la configurazione della tableColumn con la lista dei volumi. La prima istruzione degna di interesse è quella che dimensiona la `CoverView` (cui `cvrView` punta: è un outlet che ho aggiunto proprio allo scopo). Successivamente c'è una lunga serie di istruzioni necessarie alla costruzione della outlineView. Poiché non c'è più Interface Builder a darmi una mano, devo specificare tutto: quanto grande è, quali colonne e come sono fatte, quale sia il tag della outlineView (indispensabile per la visualizzazione), ovviamente la classe dataSource e delegata.

Al termine, oltre ad aggiustare un paio di collegamenti, c'è la fondamentale istruzione che dice di aggiungere la outlineView come una sottovista della `CoverView`. Solo in questo modo, infatti, la outlineView riceve i messaggi di ridisegno (che poi scatenano tutte le richieste alla classe dataSource).

DISEGNARE IN UNA VIEW

Eccomi arrivato al metodo principale di questo capitolo, ovvero come disegnare all'interno di una view. Il metodo è piuttosto lungo ed in alcune sue parti noioso; ne riporto qui alcuni spezzoni interessanti:

```
- (void)drawRect:(NSRect)rect
{
    float x1, x2, y1, y2 ;
    NSFont * locFont ;
    NSMutableDictionary *strAttr ;
    // sfondo bianco
    [[ NSColor whiteColor] set ];
    NSRectFill( rect );
    // linee di taglio in rosso
    [[ NSColor redColor] set ];
    // due linee di taglio inferiori verticali
    x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST ;
    y1 = STARTPOINT_Y ;
    x2 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST ;
    y2 = STARTPOINT_Y + CUTSIGN_DIM ;
    [ NSBezierPath strokeLineFromPoint: NSMakePoint( CM2PX(x1), CM2PX(y1) )
toPoint: NSMakePoint( CM2PX(x2), CM2PX(y2) ) ];
```

La prima cosa da fare è riempire lo sfondo della finestra. Sono già fermo: occorre specificare il colore.

Un colore è un oggetto della classe `NSColor`. Esistono moltissimi metodi per definire un colore, e `whiteColor` è uno di questi: quindi l'istruzione

```
[ NSColor whiteColor]
```

produce un oggetto della classe `NSColor`, che rappresenta il colore bianco. Inviando ad un colore il messaggio `set` si costringe ogni successivo comando di disegno ad utilizzare il colore indicato.

Per riempire un rettangolo con un dato colore, ho scoperto esistere due diverse

modalità; la prima è quella di utilizzare un oggetto `NSBezierPath` con l'istruzione

```
[ NSBezierPath fillRect: rect ];
```

La seconda possibilità, utilizzata nell'esempio, è di utilizzare la funzione di libreria dello Application Kit `NSRectFill`. Ignoro quale sia la differenza tra le due diverse modalità; l'effetto è comunque lo stesso.

Comincio poi una lunga serie di istruzioni in cui disegno una serie di linee (le chiamo *di taglio* in quanto dovrebbero aiutare il ritaglio della copertina una volta stampata) di colore rosso. Il metodo utilizzato è quello per disegnare un linea per due punti `strokeLineFromPoint:toPoint:`.

Al termine, disegno in nero un rettangolo e due linee verticali che individuano le due aree laterali riservate generalmente al titolo del CD.

Comincio poi una serie di blocchi di istruzioni per mio divertimento.

```
    x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST + BACK_OUTER_HOR_DIM / 2
;
    y1 = STARTPOINT_Y + CUTSIGN_DIST ;
    locFont = [ NSFont fontWithName: @"Arial" size: 16 ] ;
    strAttr = [ NSMutableDictionary dictionary] ;
    [ strAttr setObject: locFont forKey: NSFontAttributeName ];
              [   strAttr   setObject:   [NSColor   greenColor]   forKey:
NSForegroundColorAttributeName ];
       [ @"back cover" drawAtPoint: NSMakePoint( CM2PX(x1), CM2PX(y1) )
withAttributes: strAttr ];
```

In questo primo blocco provo a scrivere una stringa di testo; per fare questo, produco un oggetto di classe `NSFont` in cui sono racchiuse le informazioni di carattere (tipo e dimensione). Questa informazione, assieme ad altre caratteristiche del testo (qui, ad esempio, il colore, ma ce ne sono tantissime) sono inserite in un dizionario di attributi del testo. Il tutto contribuisce all'istruzione finale del blocco, in cui la stringa 'back cover' sarà scritta a partire da un certo punto.

In effetti per molti scopi il metodo `drawAtPoint:withAttributes:` è sufficiente. Non sono tuttavia riuscito a scoprire un metodo per ruotare il testo di novanta gradi, in modo da inserirlo all'interno dell'area laterale del titolo. Cercando nella documentazione, ho trovato un esempio che permette di scrivere testo lungo un percorso qualsiasi (per Bacco!) con non troppo sforzo. Non mi voglio addentrare nei dettagli (che non mi interessano), ma solo capire a grandi linee cosa succede.

Ci sono tre classi che partecipano a quest'unico scopo: `NSTextStorage`, `NSLayoutManager` e `NSTextContainer`. La prima classe fornisce uno strumento per conservare del testo, con tutti gli attributi che si vogliono per ogni singolo elemento. Potrei dichiararla una classe 'modello' (nel senso del paradigma MVC) di un elemento di testo. La classe `NSTextContainer` funziona come una classe 'vista' (sempre per MVC), definendo una regione di spazio dove il testo deve essere rappresentato. Chi controlla la rappresentazione del testo è infine la classe `NSLayoutManager`, che coordina la disposizione e la visualizzazione dei singoli elementi di testo.

Per far funzionare tutto questo accrocchio, occorre in primo luogo costruire l'ambiente di lavoro. Ciò avviene (nella fattispecie, potrebbe non essere il posto migliore) all'interno del metodo `initWithFrame:`

```
- (id)initWithFrame:(NSRect)frameRect
{
    if ((self = [super initWithFrame:frameRect]) != nil)
    {
          textStorage = [[NSTextStorage alloc] initWithString:@"This is the
place for the cd title"];
        layoutManager = [[NSLayoutManager alloc] init];
        textContainer = [[NSTextContainer alloc] init];
        [layoutManager addTextContainer:textContainer];
        [textContainer release];
        [layoutManager setUsesScreenFonts:NO];
        [textStorage addLayoutManager:layoutManager];
        [layoutManager release];
    }
    return self;
}
```

Ho definito tre variabili d'istanza (`textStorage`, `layoutManager` e `textContainer`) per semplificarmi la vita. Per prima cosa costruisco questi tre oggetti, poi costruisco i collegamenti tra le variabili d'istanza (il codice è copiato di peso dall'esempio). Di seguito, quando devo visualizzare la stringa, procedo come segue (sono tornato all'interno del metodo `drawRect:`):

```
    NSGraphicsContext *context = [NSGraphicsContext currentContext];
    NSAffineTransform *transform = [NSAffineTransform transform];
        NSRange glyphRange = [layoutManager glyphRangeForTextContainer:
textContainer ];
    x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST + BACK_LABEL_HOR_DIM;
    y1 = STARTPOINT_Y + CUTSIGN_DIM + CUTSIGN_DIST ;
    [ transform translateXBy:CM2PX(x1) yBy:CM2PX(y1) ];
    x2 = 90 ;
    [ transform rotateByDegrees: x2];
    [context saveGraphicsState];
    [transform concat];
                [    layoutManager    drawGlyphsForGlyphRange:glyphRange
atPoint:NSMakePoint(5, 0)];
    [context restoreGraphicsState];
```

In questa sezione di codice temo ci siano concetti che potrebbero impegnare giorni per essere spiegati compiutamente; anche qui, mi darò una giustificazione più o meno funzionale, giusto per capire come fare. Parto dall'oggetto `NSGraphicsContent`. Ogni istruzione di disegno avviene all'interno di un contesto grafico, un oggetto che raccoglie tutte le caratteristiche di disegno (spessore delle linee, colore di background, sistema di coordinate, cose del genere, penso). Il motivo della presenza di questo oggetto è di poter bloccare per un momento il sistema di riferimento, per poi ripristinarlo al termine delle operazioni di disegno (la coppia `saveGraphicsState` e `restoreGraphicsState`). Infatti, per poter eseguire con comodo il disegno della stringa in senso verticale, effettuo un cambio nel sistema di riferimento. È questo lo scopo dell'oggetto della classe `NSAffineTransform`: la variabile `transform` permette di modificare il sistema di riferimento (qui, se non avete mai fatto un po' di geometria non ne venite fuori facilmente). Prima sposto l'origine sul punto base del rettangolo che delimita l'area

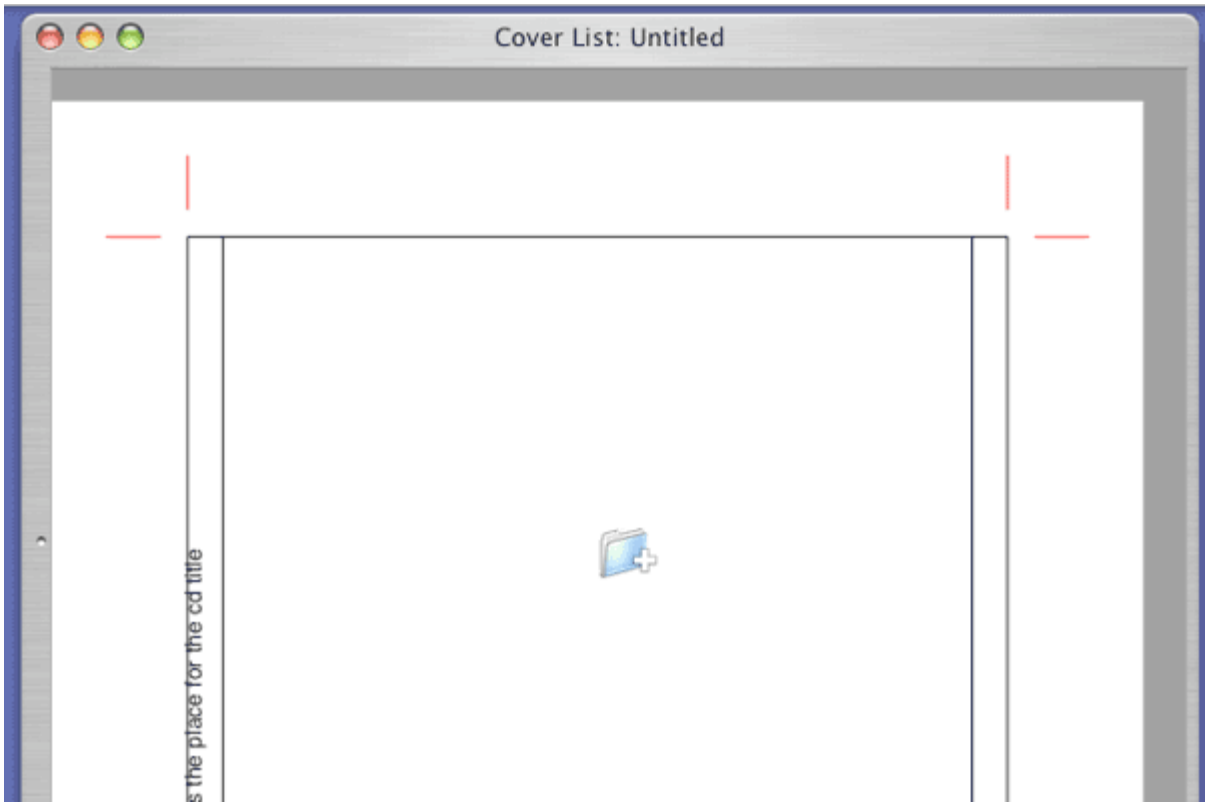
riservata al titolo, poi ruoto il tutto di novanta gradi. Il messaggio `concat` permette di applicare queste trasformazioni in catena a quelle standard (che sono già state predisposte dall'ambiente operativo in quanto sto disegnando all'interno di una view). Finalmente posso dare il comando di disegno del testo con il metodo `drawGlyphsForGlyphRange:atPoint:`, che dice di disegnare gli elementi del testo (i *glifi*) nel range specificato a partire dal punto indicato (che ho spostato di 5 pixel giusto per non stare troppo vicino al rettangolo delimitante). Il range indicato (un range individua una serie di elementi all'interno di una lista ordinata) è ottenuto tramite il metodo `glyphRangeForTextContainer:` il quale, a quanto pare, effettua anche il 'rendering' interno dei caratteri in base alla regione specificata (che, non avendo io specificato, sarà di default una linea infinita).

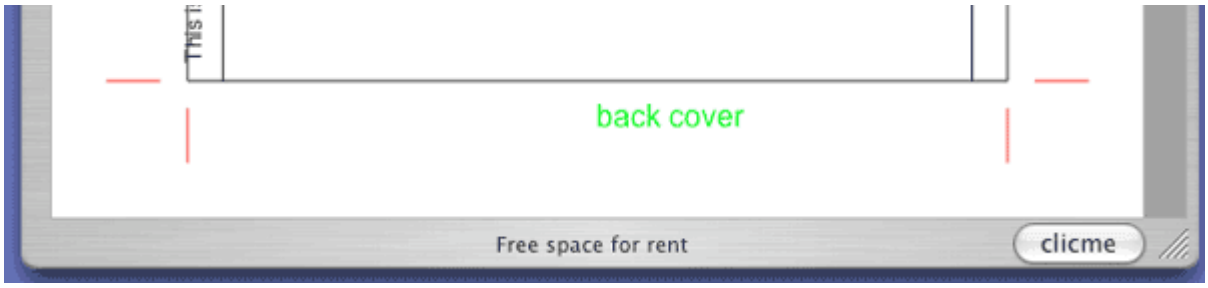
C'è un'ultima cosa che voglio fare prima di concludere: disegnare un'immagine predefinita all'interno della vista.

```
    NSImage * locImg ;
    locImg = [NSImage imageNamed: @"Add"] ;
    x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST + BACK_OUTER_HOR_DIM / 2
;
    y1 = STARTPOINT_Y + CUTSIGN_DIST+ CUTSIGN_DIST + BACK_OUTER_VER_DIM / 2
;
    [ locImg compositeToPoint:NSMakePoint( CM2PX(x1), CM2PX(y1) )
            operation: NSCompositeSourceOver ];
```

Qui riciclo una delle immagini della toolbar (giusto perché era sotto mano). Costruisco un oggetto della classe `NSImage` e poi, banalmente, uso un metodo apposito. Per il momento, nulla di più facile.

A TESTA IN SÙ

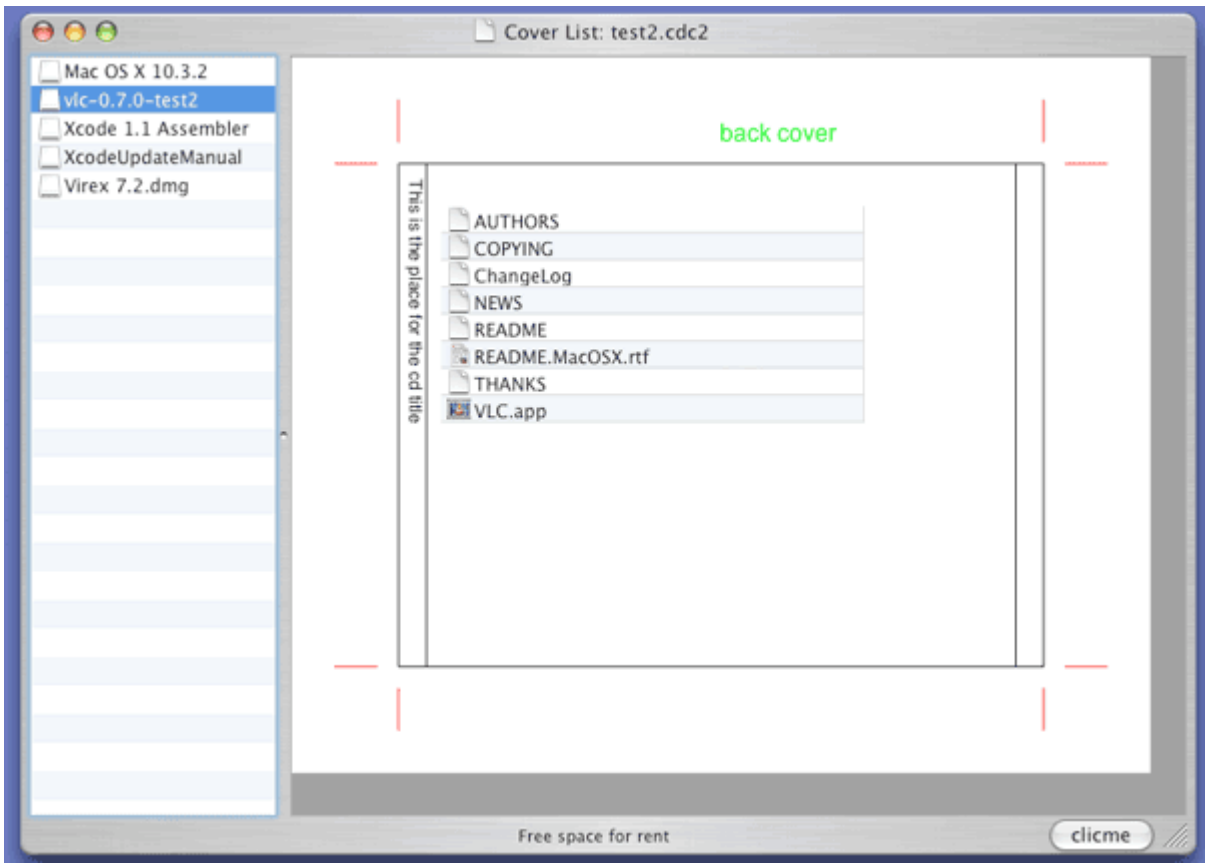




Il sistema di riferimento su cui si basa l'intero sistema di coordinate di Cocoa è particolarmente originale. Molti altri ambienti operativi utilizzano una convenzione differente (anche Mac OS 9): l'origine del sistema di riferimento è in alto a sinistra, l'asse delle ascisse viaggia verso destra e l'asse delle ordinate verso il basso. Benché Cocoa utilizzi un sistema di riferimento più matematico e naturale, in realtà è piuttosto innaturale per l'uso invalso in informatica. Ciò è particolarmente evidente quando la scrollview diventa più grande della `CoverView`; lo sfondo della scrollView non coperta dalla `CoverView` (che ho appositamente mantenuto di un colore grigio piuttosto diverso dal bianco utilizzato come sfondo per la `CoverView`) cresce in alto e a destra. In effetti, sono abituato a vederlo crescere in basse e destra.

Senza dover diventare matto a cambiare ogni singola coordinata, c'è un meccanismo molto semplice per rovesciare tutte le coordinate verticali. Basta scrivere il metodo seguente:

```
- (BOOL)isFlipped {
    return YES;
}
```



La realizzazione di default di questo metodo restituisce `NO`. Dicendo invece `YES` si costringe Quartz a rovesciare l'asse delle ordinate, spostando appunto l'origine del sistema di riferimento dal basso-a-sinistra in alto-a-sinistra.

Capitolo 040 - Polimorfismo classico

Comincio ad organizzare la modalità di disegno all'interno delle view.

C'è un esempio di Apple nei Dev Tools, si chiama Sketch, da cui trarre consigli.

Primo inserimento: 15 gennaio 2004

UN CLASSICO MODELLO OOP

Uno degli esempi più abusati di quando si intende spiegare la programmazione object oriented è di rappresentare un elemento grafico come un oggetto. Poiché ogni oggetto è autocontenuto e possiede nel suo interno tutte le informazioni e le capacità necessarie, per disegnare una collezione di oggetti potrei mandare a tutti lo stesso messaggio.

Bene, è proprio questo il meccanismo che mi accingo a realizzare per strutturare meglio il disegno di elementi grafici all'interno della coverView. Finora infatti il disegno avveniva brutalmente all'interno del metodo `drawRect:` con la pedissequa sequenza delle operazioni.

In realtà, l'idea della view è che l'utente disegni la copertina come meglio gli aggrada, magari partendo da uno schema predefinito (le linee guida di taglio, cose del genere), aggiungendo o togliendo elementi dalla finestra. Per fare questo, bisogna che il contenuto della view sia disegnato non staticamente, ma in base ad una collezione di elementi grafici che sono stati definiti dall'utente (e magari in parte definiti per la natura della copertina). Vedo che sto cercando di spiegarmi con un mare agitato di parole; vi invito allora a pensare ad una applicazione di disegno vettoriale (Adobe Illustrator, ad esempio, ma anche l'esempio citato Sketch va benissimo). Qui l'utente disegna sulla finestra linee, rettangoli, inserisce testi, eccetera. Alla fine la mia finestra coverView vorrà, nel suo piccolo, fornire le stesse funzioni (ovviamente, in forma semplificata). In più dovrà fornire una serie di *template*, ovvero schemi precostituiti come il contorno da ritagliare del retro della copertina (sul quale finora mi sono esercitato) come guida per la composizione.

Come esercizio per questo capitolo voglio ripetere il disegno del capitolo precedente; questa volta, tuttavia, le linee, i rettangoli, i testi e le immagini non sono disegnate brutalmente dall'interno del metodo `drawRect:`, ma derivano da una operazione propriamente OOP su una struttura dati, dove è descritto, in maniera astratta, il contenuto della view.

UN MODELLO PER LA GRAFICA

Il primo passo è costruire una classe modello per un elemento grafico, una sorta di classe astratta dalla quale far discendere poi tutti gli elementi grafici.

Pigliando ispirazione da Sketch (ma rifuggo dall'ipotesi di plagio: ho già sfruttato tale

costruzione molti anni fa in altri ambienti, e poi si tratta comunque di una tecnica oserei dire scolastica), dichiaro una classe siffatta:

```
@interface CCE_BasicForm : NSObject {
    // rettangolo che contiene l'oggetto
    NSRect        elemLimit ;
    // spessore della linea di contorno
    float        cceLineWidth ;
    // colore dell'eventuale interno e del contorno
    NSColor        * cceFillColor ;
    NSColor        * cceLineColor ;
    // se devo colorare l'interno o meno
    BOOL        cceIsFilled ;
    // se l'elemento e' selezionato o meno
    BOOL        cceIsSelected ;
}
// metodi che ogni sottoclasse deve realizzare
// inizializzazione, in qualche forma
- (id) init ;
// deallocazione
- (void) dealloc ;
// disegno dell'elemento
- (void) drawElement: (NSRect) inRect ;
...
@end
```

In questa classe raccolgo un po' di variabili d'istanza che serviranno più o meno sempre in tutti gli elementi grafici da disegnare; per tutti aggiungo i metodi accessor, così mi tolgo il pensiero e li rendo disponibili a tutte le sottoclassi.

La parte importante della classe sono però i tre metodi lì dichiarati, che ogni sottoclasse deve in qualche forma realizzare. In particolare, il primo non dovrà essere sempre così semplice (sarò chiaro più tardi), il secondo viene da sé, mentre il terzo è il cuore del meccanismo. Se infatti faccio in modo che tutti gli elementi che a qualche titolo si possono disegnare all'interno della coverView rispondono al messaggio `drawElement`, ecco che per ri-disegnare il contenuto della view, è sufficiente inviare questo messaggio a tutti gli elementi.

Dal punto di vista concettuale, ho già finito. Adesso aggiungo alla coverView una variabile d'istanza della classe `NSMutableArray` (un vettore di dimensione variabile); ogni volta che voglio rappresentare un elemento grafico nella coverView costruisco un oggetto delle classe `CCE_BasicElem` che risponda al messaggio `drawElement`, e poi aggiungo quest'elemento al vettore.

Il metodo `drawRect:` della coverView diventa di una semplicità disarmante (`elemArray` è il vettore `NSMutableArray`).

```
- (void) drawRect: (NSRect) aRect
{
    int        i , numElem ;
    CCE_BasicForm * elem ;
    // conto quanti elementi sono presenti
    numElem = [ elemArray count ];
    // spazzolo tutit gli elementi del vettore
    for ( i = 0 ; i < numElem ; i ++ )
    {
```

```
        // recupero l'elemento i-esimo
        elem = [ elemArray objectAtIndex: i ];
        // gli dico di disegnarsi
        [ elem drawElement: aRect ];
    }
}
```

Come si può vedere, tutto molto bello ed elegante.

Detto questo, tutto il percorso è adesso in discesa.

Pigliamo ad esempio come fare a disegnare una linea. Non si può fare a meno di richiedere due punti, il punto di partenza e quello di arrivo; il metodo di inizializzazione presenta quindi due argomenti:

```
- (id) initStartPt: (NSPoint) stPt toPt: (NSPoint) endPt ;
```

Poi, per evitare ogni volta di costruire un oggetto `NSBezierPath` (come fatto nel capitolo precedente, nel quale per ogni segmento si costruiva un oggetto del genere, per poi buttarlo via subito), mi conservo il segmento come variabile d'istanza. In conclusione, la dichiarazione della classe è la seguente:

```
@interface CCE_Line : CCE_BasicForm {
    // punto di inizio
    NSPoint            startPoint ;
    // punto di terminazione
    NSPoint            endPoint ;
    // tengo il percorso per meglio disegnare
    NSBezierPath    * theLine ;
}
// i soliti metodi comuni
- (id) initStartPt: (NSPoint) stPt toPt: (NSPoint) endPt ;
- (void) dealloc ;
- (void) drawElement: (NSRect) inRect ;
...
@end
```

La realizzazione è molto semplice:

```
-(id) initStartPt: (NSPoint) stPt toPt: (NSPoint) endPt
{
    self = [ super init ];
    if ( self )
    {
        // per mia praticita', costruisco un percorso
        NSBezierPath *path = [NSBezierPath bezierPath];
        // assegno i punti di partenza ed arrivo
        startPoint = stPt ;
        endPoint = endPt ;
        // costruisco il percorso
        [path moveToPoint: startPoint ];
        [path lineToPoint: endPoint ];
        // lo assegno
        [ self setTheLine: path ];
        // metto a posto il rettangolo che lo contiene
        [self setElemLimit: [theLine bounds] ];
    }
    return ( self );
```

```
}
```

Mi tengo da parte i due punti, poi costruisco ed assegno il segmento. Per il momento, mi tengo anche i punti estremi ed il rettangolo che racchiude il segmento (anche se in effetti non servono) in quanto ho il sospetto mi potranno servire poi (ad esempio, per capire se un clic all'interno della coverView avviene proprio sopra in segmento; una informazione del genere è fondamentale per poter selezionare e manipolare il segmento...).

Perfino banale il metodo che realizza il disegno vero e proprio:

```
- (void) drawElement: (NSRect) inRect
{
    // imposto il colore
    [ cceLineColor set ];
    // imposto lo spessore della linea
    [ theLine setLineWidth: [ self cceLineWidth ]];
    // disengo la linea
    [ theLine stroke ];
}
```

Si tratta di impostare colore e spessore, e di procedere al disegno.

Arrivato a questo punto, non voglio tediarvi con la spiegazione degli altri elementi grafici che ho definito, ovvero le classi `CCE_Rect`, `CCE_Text` e `CCE_Image`, le quali, nell'ordine, servono a rappresentare un rettangolo, un testo variamente orientato, ed una immagine. In pratica ho estratto il codice del vecchio metodo `drawRect:` e l'ho spezzato, elemento per elemento per elemento, inglobando ogni pezzo all'interno dell'oggetto di appartenenza.

VETTORI, GRUPPI E TEMPLATE

C'è però un particolare elemento grafico che mi preme descrivere, in quanto non è un elemento grafico, ma è particolarmente utile ad altri scopi.

Accade spesso che un certo insieme di elementi grafici semplici concorrano a realizzare un elemento più complicato. Oppure questi elementi fanno tutti parte di una entità logica comune. Se avete usato Illustrator o un CAD qualsiasi, sto parlando del concetto di *Gruppo*. Un gruppo è un insieme di oggetti che dovrebbe essere manipolato assieme in quanto è un tutt'uno logico. Concetti parenti del Gruppo sono quello di *layer* (suddivisione degli elementi secondo un piano di appartenenza) e quello di *template* (o blocco moduli, o file modello, chiamatelo come preferite:insieme di elementi predefiniti, generalmente non modificabile, a partire dal quale costruire rappresentazioni più complesse). In altre parole, spesso conviene organizzare tutti gli elementi grafici non tanto sotto forma di vettore (in cui le uniche relazioni tra gli elementi sono chi viene prima e chi viene dopo) ma sotto una forma gerarchica più complicata (un albero, tipicamente).

Per tenere conto in qualche modo di queste esigenze, ho definito una sottoclasse `CCE_ElemGroup`; questa classe in effetti è un vettore di elementi grafici. Tuttavia, come sua caratteristica peculiare, è in grado di rispondere ai (per ora) tre caratteristici metodi di un elemento grafico: `init`, `dealloc` e `drawElement`. Ma per rendersi conto

della semplicità e della potenza di questo oggetto, occorre guardare la sua dichiarazione e la sua realizzazione.

```
@interface CCE_ElemGroup : CCE_BasicForm {
    // il vettore degli elementi raggruppati
    NSMutableArray        * elemArray ;
}
// i soliti metodi comuni
- (id) init ;
- (void) dealloc ;
- (void) drawElement: (NSRect) inRect ;

// per aggiungere un elemento al gruppo
- (void) addElem: (CCE_BasicForm *) elem ;
...
@end
```

Fin qui, nulla di speciale; l'unica variabile d'istanza è un `NSMutableArray`; è quindi giocoforza aggiungere anche metodi per manipolare tale vettore. Al momento c'è solo il metodo per aggiungere un elemento, ma mi figuro che poi ce ne saranno altri (per togliere, ordinare, eccetera).

Il metodo di inizializzazione non è particolarmente brillante:

```
-(id) init
{
    self = [ super init ] ;
    if ( self )
        [ self setElemArray: [ NSMutableArray array ]];
    return self ;
}
```

Ma passiamo al metodo di disegno:

```
- (void) drawElement: (NSRect) aRect
{
    int        i , numElem ;
    CCE_BasicForm * elem ;

    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem drawElement: aRect ];
    }
}
```

ricorda qualcosa, vero? è esattamente lo stesso codice del metodo `drawRect:` della coverView.

Bene, seguite questa visione: tutti gli elementi all'interno della coverView sono raccolti dentro vettori, o meglio, ancora, dentro `CCE_ElemGroup`. L'insieme di tutti gli elementi è un `CCE_ElemGroup`; il quale vettore a sua volta è composto da elementi grafici ed altri `CCE_ElemGroup`; e questi `CCE_ElemGroup`, a cascata, da altri elementi ed altri `CCE_ElemGroup`.

Quando, fra qualche (chissà quanti) capitoli, sarà possibile selezionare un po' di elementi e poi, da comando di menu, farne un gruppo, si pigliano questi elementi e li si riuniscono all'interno di un `CCE_ElemGroup`, che li sostituisce all'interno del `CCE_ElemGroup` che li elencava in precedenza. E poiché il gruppo è un elemento grafico, sarà in grado di rispondere a tutti i messaggi comuni ed eseguire le funzionalità comuni a tutti gli elementi. Ad esempio, un clic dell'utente su un elemento del gruppo sarà in prima battuta intercettato dall'elemento `CCE_ElemGroup` che definisce il gruppo: l'utente quindi seleziona il gruppo (comportamento corretto) e non l'elemento (che sarebbe scorretto). Non sono sicuro di aver reso appieno la potenza e l'eleganza di questo costrutto informatico. Tutto quello che posso fare è completare questo capitolo con le modifiche della coverView.

DISEGNARE PER ELEMENTI

La classe coverView elimina tutte le precedenti variabili d'istanza (che servivano come appoggio per disegni vari, ed utilizza (a parte gli outlet, che ovviamente rimangono) una singola variabile d'istanza, che guarda un po', è la seguente:

```
    CCE_ElemGroup        * theElements ;
```

Questa variabile contiene tutti gli elementi da rappresentare all'interno della coverView. Il metodo `drawRect:` è quindi diventato molto ma molto semplice:

```
- (void)drawRect:(NSRect)rect
{
    // sfondo bianco
    [[ NSColor whiteColor] set ];
    NSRectFill( rect );
    // poi dico di disegnarsi a tutti gli elementi
    [ [self theElements] drawElement: rect ];
}
```

Per il momento, per disegnare qualcosa all'interno, aggiungo elementi alla variabile `theElements` all'interno del metodo `initWithFrame` della coverView. In realtà, chi aggiungerà elementi alla variabile sarà l'utente, disegnandoli a schemi come in ogni programma di disegno vettoriale che si rispetti. Anche questa volta, accenno solamente a parti del codice del metodo, lungo e noioso:

```
- (id)initWithFrame:(NSRect)frameRect
{
    float x1, x2, y1, y2 ;

    if ((self = [super initWithFrame:frameRect]) != nil)
    {
        // costruisco il vettore degli elementi
        [ self setTheElements: [[CCE_ElemGroup alloc] init ] ];
        // aggiungo un po' di roba
        if ( 1 )
        { // qui disegno tutto lo sfondo della copertina
            CCE_Line        * tmpLine ;
            CCE_Rect        * tmpRect ;
            CCE_ElemGroup * backCover ;
            // raccolto tutti gli elementi in un gruppo
```

```
            backCover = [ [ [ CCE_ElemGroup alloc ] init ] autorelease ];
            // due linee di taglio inferiori verticali
            x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST ;
            y1 = STARTPOINT_Y ;
            x2 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST ;
            y2 = STARTPOINT_Y + CUTSIGN_DIM ;
                 tmpLine = [ [ CCE_Line alloc] initStartPt: NSMakePoint(
CM2PX(x1), CM2PX(y1) )
                                                    toPt: NSMakePoint( CM2PX(x2),
CM2PX(y2) ) ] ;
            [ tmpLine setCceLineColor: [ NSColor redColor] ];
            [ tmpLine setCceLineWidth: 0 ];
            [ backCover addElem: tmpLine ];
```

Il metodo inizia inizializzando l'elemento `theElements`, vuoto.

Con la prima sezione di istruzioni intendo definire un gruppo; questo gruppo è costruito da tutti gli elementi che formato la rappresentazione di base dello schema della copertina (le linee di taglio e il rettangolo che la contiene, completa delle due bande laterali col titolo del CD).

Le istruzioni, di volta in volta, predispongono i punti di partenza ed arrivo dei segmenti, il colore della linea, il suo spessore, eccetera; aggiungo poi l'elemento grafico così costruito non alla lista degli elementi `theElements`, ma all'elemento `backCover` che li raggruppa. Solo alla fine, col codice seguente, completo l'operazione.

```
                     x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST +
BACK_OUTER_HOR_DIM - BACK_LABEL_HOR_DIM ;
            y1 = STARTPOINT_Y + CUTSIGN_DIM + CUTSIGN_DIST ;
                     x2 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST +
BACK_OUTER_HOR_DIM - BACK_LABEL_HOR_DIM ;
                     y2 = STARTPOINT_Y + CUTSIGN_DIM + CUTSIGN_DIST +
BACK_OUTER_VER_DIM ;
                tmpLine = [ [ CCE_Line alloc] initStartPt: NSMakePoint(
CM2PX(x1), CM2PX(y1) )
                                                    toPt: NSMakePoint( CM2PX(x2),
CM2PX(y2) ) ] ;
            [ tmpLine setCceLineColor: [ NSColor blackColor] ];
            [ tmpLine setCceLineWidth: 0 ];
            [ backCover addElem: tmpLine ];
            // aggiungo l'intero gruppo...
            [ [self theElements] addElem: backCover ];
        }
```

Qui il gruppo è completato con l'ultima linea; poi il gruppo completo è aggiunto, finalmente, all'elenco degli elementi grafici.

La storia prosegue con le stringhe di testo (una verticale ed una orizzontale) e si conclude con la solita immagine.

```
        if ( 1 )
        { // aggiungo il disegno dell'immagine
            NSImage        * locImg ;
            CCE_Image * tmpImg ;
            // recupero l'immagine
            locImg = [NSImage imageNamed: @"Add"] ;
            // calcolo il punto
                     x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST +
```

```
BACK_OUTER_HOR_DIM / 2 ;
                        y1  =  STARTPOINT_Y  +  CUTSIGN_DIST+  CUTSIGN_DIST  +
BACK_OUTER_VER_DIM / 2 ;
            tmpImg = [[ CCE_Image alloc] initWithImage: locImg
                          atPoint: NSMakePoint( CM2PX(x1), CM2PX(y1) ) ];
            // aggiungo l'elemento al vettore
            [ [self theElements] addElem: tmpImg ];
        }
```

Non succede nulla di stano; si costruisce l'immagine e la si aggiunge al vettore degli elementi. In definitiva, gli elementi di cui il disegno è composto sono quattro: due stringhe di testo, una immagine e un gruppo. Il gruppo a sua volta è costruito da una diecina di elementi grafici (vari segmenti ed un rettangolo). Se non vi ho ancora convinto della bontà di questo costrutto, vi lascio immaginare lo scenario seguente. Un oggetto `CCE_ElemGroup` è un oggetto come tanti altri; diciamo che si tratta del gruppo visto sopra che traccia lo schema della copertina. Potrei salvarlo su file coi meccanismi classici (`encodeWithCoder:...`). Come si trova su file, posso recuperalo così com'è, e ridisegnarlo tale e quale su altre coverView.

Ho appena descritto un meccanismo per costruire, importare ed esportare template.

Capitolo 041 - Costruisco Elementi

Prima di avventurarmi nel disegno interattivo di elementi grafici della coverView, faccio un po' di esercizio costruendo una finestra che dipinge elementi di vario tipo.

Nulla di nuovo, uso documentazione Apple e piglio ispirazione da Sketch.

Primo inserimento: 4 febbraio 2004

Alcune Piccole Modifiche

Per prima cosa, ripulisco un po' il codice e modifico alcune classi. Nella inizializzazione della coverView, elimino il gruppo di elementi, alcune scritte e l'immagine. Anzi, elimino del tutto la classe `CCE_Image`, che tanto era imperfetta e verosimilmente la riprenderò più avanti, con più completezza.

Poi passo a trattare meglio la classe base: aggiungo i metodi accessor per la variabile `cceIsFilled`, la variabile `cceIsStroked` (intendendo con essa indicare se si devono tracciare i bordi di figure oppure no) e i relativi metodi accessor.

Appare ovvia poi l'imposizione a `TRUE` di questa variabile per tutti gli oggetti della classe `CCE_Line`. Avendo aggiunto la variabile `cceIsStroked`, ci sono delle modifiche sul metodo di disegno di `CCE_Rect`:

```
- (void) drawElement: (NSRect) inRect
{
    if ( cceIsStroked )
    {
        // imposto il colore del contorno
        [ cceLineColor set ];
        // imposto lo spesosre del contorno
        [ theRect setLineWidth: [ self cceLineWidth ]];
        // disegno il contorno
        [ theRect stroke ];
    }
    // se il rettangolo e' pieno
    if ( cceIsFilled )
    {
        // imposto il colore del pieno
        [ cceFillColor set ];
        // riempi tutto
        [ theRect fill ];
    }
}
```

Non c'è molto da dire qui: da notare che non faccio alcun controllo che almeno una tra le variabili d'istanza `cceIsStroked` e `cceIsFilled` sia a `TRUE`; fossero entrambi `FALSE`, il rettangolo in pratica non è disegnato...

A questo punto, aggiungo un nuovo elemento grafico, che viene fuori facile una volta

messo a punto il rettangolo: l'ellisse (di cui il cerchio è un caso particolare). Dico infatti che per determinare un'ellisse basta specificare il rettangolo in cui è inscritta.

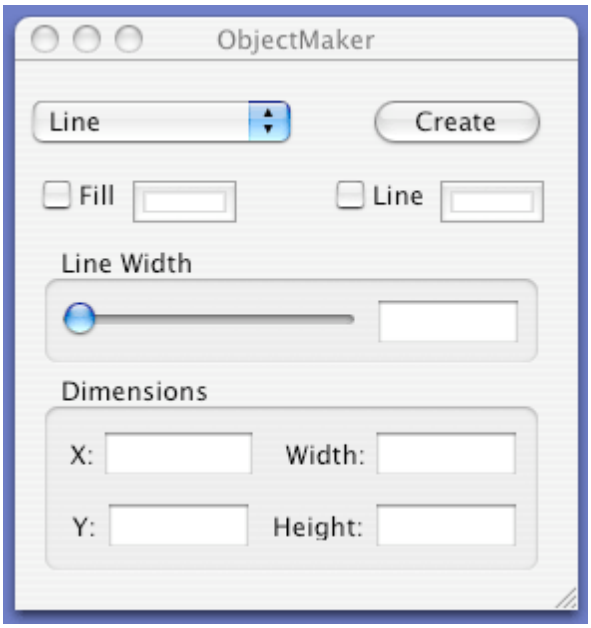
```
- (id) initWithRect: (NSRect) aRect ;
{
    self = [ super init ];
    if ( self )
    {
        // per mia convenienza conservo il percorso
         NSBezierPath *path = [NSBezierPath bezierPathWithOvalInRect: aRect
];
        [ self setTheCircle: path ];
        // metto a posto il rettangolo che lo contiene
        [self setElemLimit: aRect ];
    }
    return ( self );
}
```

Continua a non esserci molto da dire.

Il Fabbricatore d'Elementi

Il metodo più naturale per disegnare elementi all'interno della coverView è ovviamente quello di utilizzare il mouse e tracciare l'elemento che interessa, come ogni buona interfaccia grafica deve fare. Tuttavia, mi pare che il problema non sia particolarmente semplice. Comincio allora col preparare una finestra ausiliaria che consenta di aggiungere elementi grafici alla coverView stabilendo per questi elementi i parametri di inizializzazione richiesti.

Allo scopo prelevo di peso una finestra dall'applicazione Sketch negli esempi di Apple, la copio all'interno della mia applicazione, ne modifico l'aspetto e soprattutto la funzione.



In primo luogo, ho un menu pop-up al quale attribuisco tre voci: `Line`, `Rect`, `Circle`. Con questo controllo stabilisco il tipo da elemento da inserire. Ci sono poi due pulsanti

di spunta e due `NSColorWell` per attribuire il colore del contorno e del riempimento. I pulsanti di spunta stabiliscono col loro stato se il contorno o il riempimento va tracciato, mentre i due `NSColorWell` stabiliscono il colore.

Il controllo `NSColorWell` è particolarmente ricco. In maniera assolutamente trasparente per il programmatore consente all'utente si scegliere un colore con uno dei metodi standard del sistema operativo. Per il programmatore, è possibile reagire nel momento della selezione del colore, oppure lasciar perdere e recuperare il colore da utilizzare all'ultimo momento utile. È proprio questa la strategia che sarà da me utilizzata.

C'è poi un doppio controllo per scegliere lo spessore del contorno: uno *slider* cui è associato un campo testo. Il gioco dei collegamenti fra questi due controlli fa sì che l'impostazione dello spessore sia possibile sia da slider che da campo di testo.

Infine, ci sono quattro campi di testo all'interno dei quali scrivere quattro coordinate, o meglio due coppie di valori. Senza stare troppo a sottilizzare (questa finestra sparirà o cambierà sostanza spero molto presto), nel caso di linea, si determinano i due punti, iniziale e finale, della linea. Nel caso di rettangolo o ellisse, il punto di partenza e le due dimensioni lunghezza e larghezza.

Ogni singolo controllo dell'interfaccia va collegato all'interno della classe che controlla l'aspetto della finestra, in quanto ad un certo punto sarà necessario recuperare tutti i valori per stabilire come tracciare l'elemento grafico. Completa poi l'interfaccia un bel pulsante di `Create` per costruire direttamente l'elemento grafico. Qui ci va un

collegamento del tipo target/action, in modo da scatenare la costruzione dell'elemento grafico. Dicevo appunto della classe cui appartiene il controllore della finestra. Dichiaro la nuova classe `ObjMakerWinCtl`, che risulta così stabilita:

```
@interface ObjMakerWinCtl : NSWindowController {
    IBOutlet NSButton        * fillCheckbox;
    IBOutlet NSColorWell     * fillColorWell;
    IBOutlet NSButton        * lineCheckbox;
    IBOutlet NSColorWell     * lineColorWell;
    IBOutlet NSSlider        * lineWidthSlider;
    IBOutlet NSTextField     * lineWidthTextField;
    IBOutlet NSTextField     * xTextField;
    IBOutlet NSTextField     * yTextField;
    IBOutlet NSTextField     * widthTextField;
    IBOutlet NSTextField     * heightTextField;
    IBOutlet NSPopUpButton    * classSelection;
    // puntatore alla view della finestra dove si disegna
    CoverView                * theDrawView ;
    IBOutlet NSButton        * createButton;
}

+ (id)sharedObjMakerWinCtl;

- (IBAction)createObject:(id)sender;
...
@end
```

Per richiamare questa finestra, opero nel solito modo: aggiungo nel file `MainMenu.nib` una nuova voce di menu, appunto per mostrare questa finestra. Va da sé che aggiungo nella classe `AppDelegate` un nuovo metodo

```
- (IBAction)showObjectMaker:(id)sender;
```

che serve a costruire la solita istanza, condivisa a tutta l'applicazione, della finestra fabbricatrice di elementi `ObjMakerWinCtl`.

C'è poi da inizializzare il tutto con il solito metodo:

```
- (void)windowDidLoad
{
    [super windowDidLoad];
    [fillCheckbox setState:NSOffState];
    [fillCheckbox setEnabled:YES];
    [fillColorWell setColor:[NSColor whiteColor]];
    [fillColorWell setEnabled:YES];
    [lineCheckbox setState:NSOnState];
    [lineCheckbox setEnabled:YES];
    [lineColorWell setColor:[NSColor blackColor]];
    [lineColorWell setEnabled:YES];
    [lineWidthSlider setFloatValue:0.0];
    [lineWidthSlider setEnabled:YES];
    [lineWidthTextField setFloatValue:0.0];
    [lineWidthTextField setEnabled:YES];
    [xTextField setStringValue:@""];
    [xTextField setEnabled:YES];
    [yTextField setStringValue:@""];
    [yTextField setEnabled:YES];
    [widthTextField setStringValue:@""];
```

```
    [widthTextField setEnabled:YES];
    [heightTextField setStringValue:@""];
    [heightTextField setEnabled:YES];
    ...
}
```

Qui mancano le ultime tre istruzioni, che meritano un paragrafo a parte

COLPIRE IL GIUSTO TARGET

Ora, abbiamo il solito problema di una finestra condivisa a tutta l'applicazione che però opera direttamente su una delle tante finestre disponibili. Qui la cosa è vieppiù complicata dal fatto che la finestra davanti a tutti, in cui dovrebbe operare il fabbricatore, può essere corretta (una `CoverWinCtl`) o sbagliata (una `ListWinCtl`).

Per gestire questa situazione, utilizzo la variabile d'istanza `theDrawView`, che contiene la coverView correntemente attiva; userò la convenzione d'impostarla a `nil` ogni volta che la finestra di fronte a tutte le altre non contiene la coverView. Per gestire dunque il valore di questa variabile, ricorro ancora una volta (lo avevo già fatto con la finestra `InfoWinCtrl`) alle notifiche `NSWindowDidBecomeMainNotification` e `NSWindowDidResignMainNotification` inviate dall'ambiente operativo quando cambia la finestra di fronte a tutte le altre.

Per sfruttarle correttamente, occorre aggiungere per due volte (una per ciascuna notifica) la finestra `ObjMakerWinCtl` come osservatore. Bisgna aggiungere le due seguenti istruzioni al metodo `windowDidLoad`:

```
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowChanged:)
        name:NSWindowDidBecomeMainNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowResigned:)
        name:NSWindowDidResignMainNotification object:nil];
```

In questo modo, quando la finestra di fronte cambia, sono chiamati i metodi indicati dalla espressione `@selector(.)`. Questi due metodi sono molto semplici:

```
- (void)
mainWindowChanged: (NSNotification *)notification
{
    [self setMainWindow:[notification object]];
}

- (void)
mainWindowResigned: (NSNotification *)notification
{
    [self setMainWindow:nil];
}
```

ma solo perché è tutto compreso nel metodo `setMainWindow`, che ha come parametro la finestra che è diventata quella davanti a tutti (e quindi, se non ce ne sono, si passa nil).

```
- (void)
setMainWindow: (NSWindow *) mainWindow
{
    NSWindowController *controller = [mainWindow windowController];
    // se c'e' una finestra ed e' del tipo giusto
    if (controller && [controller isKindOfClass:[CoverWinCtl class]])
    {
        theDrawView = [(CoverWinCtl *)controller cvrView];
        [ createButton setEnabled:YES];
    }
    else
    {
        theDrawView = nil;
        [ createButton setEnabled:NO];
    }
    // qui dovrei aggiornare la finestra, se il caso
    // [self updateMe];
}
```

La prima istruzione eseguita cerca di recuperare il controllore della finestra a partire dalla finestra passata come parametro. Non ci fosse la finestra (cioé, il parametro `mainWindow` è `nil`), nessun problema: il messaggio `windowController` (anzi, qualsiasi messaggio) inviato ad un oggetto `nil` restituisce `nil`. Andiamo avanti: l'istruzione `if` successiva verifica se c'è il controllore della finestra, e se c'è questo controllore, se è del tipo corretto (cioè, se è una `CoverWinCtl`). Qui si sfrutta una caratteristica del linguaggio C e della logica simbolica: se in una espressione di due termini combinati secondo un `AND`, se il primo è falso, allora per certo anche l'intera espressione è falsa, e il secondo termine dell'espressione non è neppure valutato. Nel mio caso, se la finestra non c'è, si cade subito nel ramo `else` dell'istruzione `if`; ugualmente si cade lì se la finestra c'è, ma non è del tipo giusto. In questi casi, si impone a `nil` il valore della variabile `theDrawView`, e si disabilita il pulsante `Create` (evitando qualsiasi tentazione).

L'unico caso in cui si ha qualcosa di utile è proprio quando la finestra c'è ed è del tipo corretto. Allora si recupera in qualche modo la coverView e la si assegna alla variabile d'istanza; contestualmente si abilita il pulsante `Create` nel caso fosse stato disabilitato in precedenza. La presenza di un cast esplicito (l'esplicita attribuzione a controller di un tipo `CoverWinCtl*`) serve solo a tranquillizzare il compilatore.

Tuttavia, c'è ancora un problema, all'apertura della finestra `ObjMakerWinCtl`. Questa non è correttamente inizializzata (ad esempio, il pulsante `Create` è abilitato) se alla prima apertura la finestra di fronte a tutte le altre è una `ListWinCtl`. Si rimedia prontamente al problema aggiungendo come ultima istruzione del metodo `windowDidLoad` una esplicita chiamata a `setMainWindow` con argomento la finestra davanti a tutte le altre a livello di applicazione:

```
    [self setMainWindow:[NSApp mainWindow]];
```

Ora che ci penso, questa cosa mi ricorda qualcosa... Ma prima, è meglio completare il discorso aperto sul disegno.

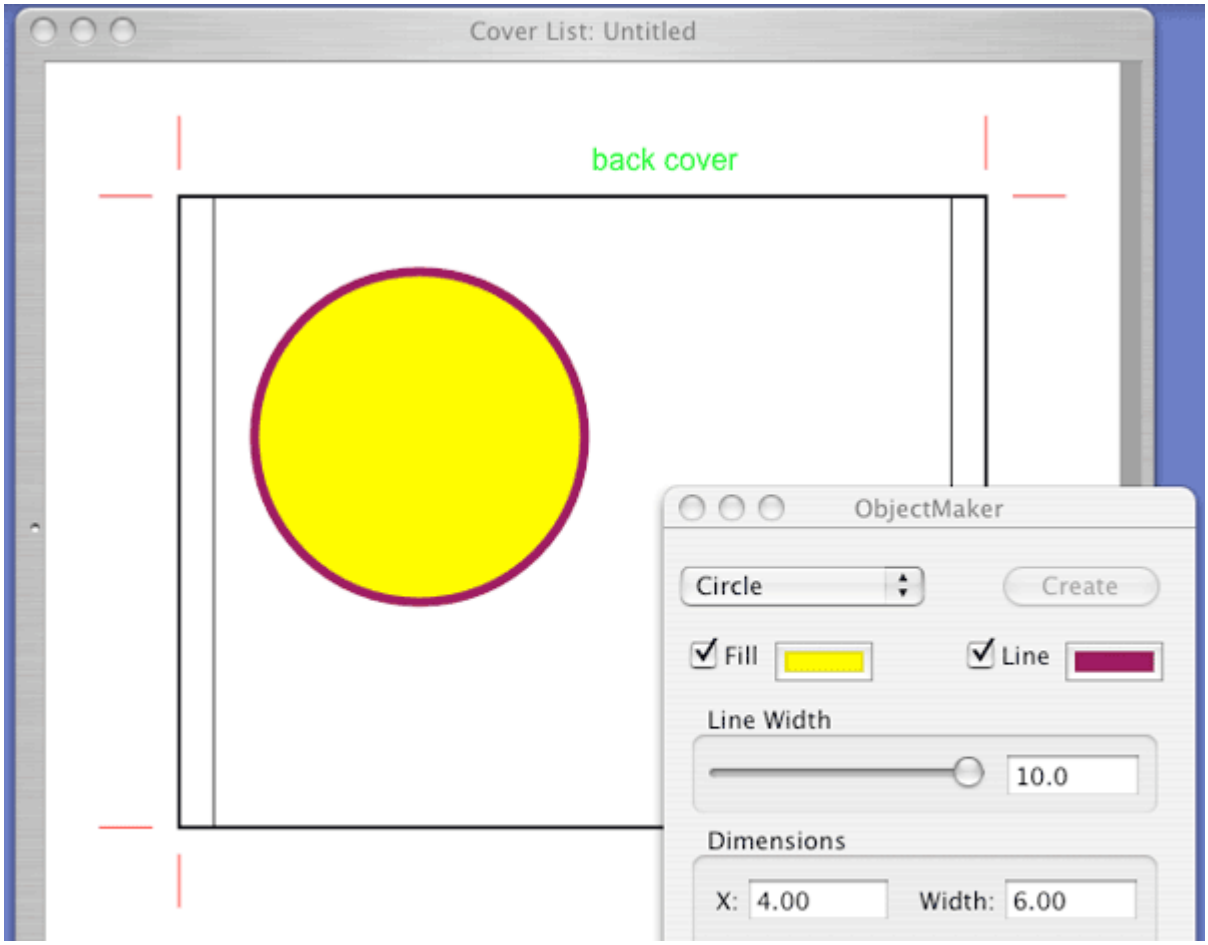
DISEGNARE COL FABBRICATORE

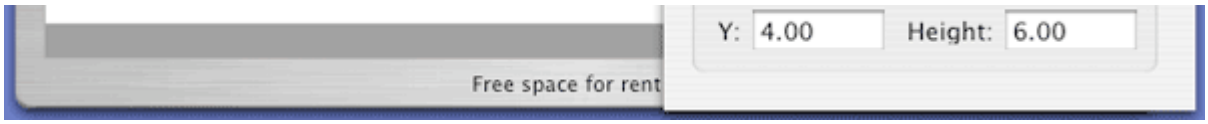
A questo punto, non rimane altro che mostrare il lungo e noioso metodo per disegnare dietro esplicito comando proveniente dal fabbricatore, ovvero il metodo invocato col meccanismo target/action:

```
- (IBAction)createObject:(id)sender
{

    if ( theDrawView == nil )
    {
        NSBeep( );
        return ;
    }
    // se arrivo qui, ho un posto dove disegnare
    switch ( [[ classSelection selectedItem] tag ] ) {
    case 10 : // disegno una linea
        {
            float            x1, y1, x2, y2 ;
            CCE_Line        * tmpLine ;

            // recupero le informazioni di colore
            x1 = [ xTextField floatValue ] ;
            y1 = [ yTextField floatValue ] ;
            x2 = [ widthTextField floatValue ] ;
            y2 = [ heightTextField floatValue ] ;
                 tmpLine = [ [ CCE_Line alloc] initStartPt: NSMakePoint(
CM2PX(x1), CM2PX(y1) )
                           toPt: NSMakePoint( CM2PX(x2), CM2PX(y2) ) ] ;
            [ tmpLine setCceLineColor: [ lineColorWell color] ];
            [ tmpLine setCceLineWidth: [ lineWidthTextField floatValue ] ];
            // recupero la vista
            [ [theDrawView theElements] addElem: tmpLine ];
                  [theDrawView setNeedsDisplayInRect: [theDrawView bounds]
];
        }
        break ;
    case 11 : // disegno un rettangolo
        {
            float            x1, y1, x2, y2 ;
            CCE_Rect        * tmpRect ;

            // recupero le informazioni di colore
            x1 = [ xTextField floatValue ] ;
            y1 = [ yTextField floatValue ] ;
            x2 = [ widthTextField floatValue ] ;
            y2 = [ heightTextField floatValue ] ;
            tmpRect = [ [ CCE_Rect alloc] initWithRect:
                 NSMakeRect( CM2PX(x1), CM2PX(y1), CM2PX(x2), CM2PX(y2) ) ]
;
            [ tmpRect setCceLineColor: [ lineColorWell color] ];
            [ tmpRect setCceFillColor: [ fillColorWell color] ];
            [ tmpRect setCceIsFilled: [fillCheckbox state ] ];
            [ tmpRect setCceIsStroked: [lineCheckbox state ] ];
            [ tmpRect setCceLineWidth: [ lineWidthTextField floatValue ] ];

            [ [theDrawView theElements] addElem: tmpRect ];
                  [theDrawView setNeedsDisplayInRect: [theDrawView bounds]
];
        }
        break ;
    case 12 : // disegno un cerchio
        {
            float            x1, y1, x2, y2 ;
```

```
            CCE_Circle        * tmpRect ;

            // recupero le informazioni di colore
            x1 = [ xTextField floatValue ] ;
            y1 = [ yTextField floatValue ] ;
            x2 = [ widthTextField floatValue ] ;
            y2 = [ heightTextField floatValue ] ;
            tmpRect = [ [ CCE_Circle alloc] initWithRect:
                 NSMakeRect( CM2PX(x1), CM2PX(y1), CM2PX(x2), CM2PX(y2) ) ]
;
            [ tmpRect setCceLineColor: [ lineColorWell color] ];
            [ tmpRect setCceFillColor: [ fillColorWell color] ];
            [ tmpRect setCceIsFilled: [fillCheckbox state ] ];
            [ tmpRect setCceIsStroked: [lineCheckbox state ] ];
            [ tmpRect setCceLineWidth: [ lineWidthTextField floatValue ] ];

            [ [theDrawView theElements] addElem: tmpRect ];
                  [theDrawView setNeedsDisplayInRect: [theDrawView bounds]
];
        }
        break ;
    default :
        break ;
    }
}
```

Il metodo non merita molti commenti; per ogni tipo di oggetto, si recuperano le informazioni dalla finestra con gli appositi metodi, e si costruisce l'elemento con i dati recuperati; poi si aggiunge l'elemento alla lista degli elementi, e si finisce col forzare il ridisegno della finestra. Nulla di più banale.





UN ERRORE CORRETTO

Dicevo prima che l'esplicito richiamo del metodo `setMainWindow:` mi ricordava qualcosa; in effetti, ho controllato: ho lo stesso problema con la finestra `InfoWinCtl`, per visualizzare le informazioni di un file selezionato nella outlineView della finestra `ListWinCtl`. Se infatti apro la finestra `InfoWinCtl`, questa non è sempre correttamente aggiornata. Manca anche qui una esplicita prima attributzione dello stato della stessa. Il problema si risolve nello stesso modo: aggiungendo in fondo al metodo `windowDidLoad` qualche istruzione. Qui non ho una semplice variabile d'istanza da aggiornare, ma una intera finestra da riempire con dati: le operazioni sono già raccolte all'interno del metodo `updateInfo:`, che andrà correttamente invocato:

```
        // pero' devo mettere a posto la prima volta
        // recupero la finestra di fronte a tutti
        NSWindowController *winCtl ;
        winCtl = [[NSApp mainWindow] windowController] ;

        if (winCtl && [winCtl isKindOfClass:[ListWinCtl class]])
            [ self updateInfo: [ (ListWinCtl *)winCtl getMainView ] ];
```

La logica di funzionamento è esattamente quella vista prima. C'è poi anche la modifica corrispondente nel metodo

```
- (void )
mainWindowChanged: ( NSNotification *) notification
{
    NSWindowController *winCtl ;
    winCtl = [[ notification object] windowController] ;
    if (winCtl && [winCtl isKindOfClass:[ListWinCtl class]])
            [ self updateInfo: [ (ListWinCtl *)winCtl getMainView ] ];
    else    [self updateInfo: nil];
}
```

in modo che la finestra torni, ad esempio, vuota quando davanti a tutte è posta una finestra `CoverWinCtl`.

ULTERIORI CORREZIONI

La fase di compilazione produce un elevato numero di warning. Come è noto, sono molto infastidito da questi messaggi, che aborro quasi più degli errori. Ecco quindi che ho cercato di eliminarne un po', dove riuscivo. Molti avvertimenti derivano dal fatto che invio dei messaggi ad un oggetto non riconosciuto come corretto destinatario dal compilatore. Ora, è ben vero che il *late binding* passa sopra allegramente a queste cose, purché alla resa dei conti il messaggio sia recepito dall'oggetto (come effettivamente qui accade). Trovo tuttavia stupido costringere l'applicazione a queste segnalzioni quando è invece ben chiara la situazione. L'uso di un cast esplicito (come già discusso qualche riga fa) elimina molti problemi.

Altre volte, invece, il polimorfismo è in realtà solo un metodo per il programmatore per essere svogliato e non pensare. In questo caso, conviene invece migliorare il codice. È quello che penso di aver fatto modificando il meccanismo con cui il documento tiene conto delle diverse finestre che a lui fanno riferimento. Piuttosto che lasciare i controllori `ListWinCtl` e `CoverWinCtl` anominimi dentro il vettore dei `windowControllers`, li ho esplicitati in due variabili d'istanza:

```
    // le due finestre associate al documento
    ListWinCtl        * listWin ;
    CoverWinCtl        * coverWin ;
```

che sono esplicitamente gestite. La prima variabile non è particolarmente eccitante; è inizializzata al suo valore all'apertura del documento, e tale rimane. La seconda variabile invece è `nil` se la finestra non è ancora aperta, ha un vero valore se la finestra è presente.

Queste variabili hanno introdotto alcune modifiche in alcuni metodi:

```
- (void)
refreshWinsWithListSts: (NSString*) sts1Bar coverSts: (NSString*) sts2Bar
{
    // al massimo ci sono due finestre: questa c'e' sicuro
    [ listWin refreshWindow ] ;
    [ listWin setStsStrings: sts1Bar andSts: sts2Bar] ;
    // questa invece vediamo
    if ( coverWin != nil )
    {
        [ coverWin refreshWindow ] ;
        [ coverWin setStsStrings: sts1Bar andSts: sts2Bar] ;
    }
}
```

Questo metodo è diventato molto più chiaro e semplice: `listWin` è sicuramente presente, ed è direttamente aggiornata. Se la variabile `coverWin` non è `nil`, allora esiste anche la finestra delle copertine, che può essere aggiornata esplicitamente. Ancora semplice è poi il metodo seguente:

```
- ( void)
showCoversWindow
{
    // se la finestra delle copertine non c'e'
    if ( coverWin == nil )
    {
        // la costruisco
        CoverWinCtl * ctl ;
        ctl = [[ CoverWinCtl alloc ] initWithWindowNibName: @"CoversWin" ];
        [ ctl autorelease ] ;
        [ self addWindowController: ctl ];
        [[ ctl window ] makeKeyAndOrderFront: self ];
        // attribuisco il valore della variabile
        [ self setCoverWin: ctl ];
    }
}
```

in cui l'unico commento che mi sento di fare è piuttosto banale: se la finestra non c'è, la si costruisce.

In realtà, le cose sono più complicate; parto da qui:

```
- (void)
makeWindowControllers
{
    // istanza del controllore
    listWin = [[ ListWinCtl alloc ] initWithWindowNibName: @"CdListWin" ];
    // autrelease, che tanto poi sara' ritenuta
    [ listWin autorelease ] ;
    // aggiungo la fienstra al documento
    [ self addWindowController: listWin ];
    if ( 1 )
    {
        CoverWinCtl * ctl ;
        ctl = [[ CoverWinCtl alloc ] initWithWindowNibName: @"CoversWin" ];
        [ ctl autorelease ] ;
        [ self addWindowController: ctl ];
        [[ ctl window ] makeKeyAndOrderFront: self ];
        [ self setCoverWin: ctl ];
    }
}
```

La variabile `listWin` è esplicitamente assegnata, anche se, comandata `autorelease`, rimane viva solo perché assegnata al vettore `windowControllers`. Per mia comodità (mi stufavo ogni volta all'apertura dell'applicazione a selezionare l'apposita voce di menu) apro immediatamente anche la finestra della copertina.

Da notare il fatto che assegno anche la variabile d'istanza con l'apposito accessor. Valuto il valore della famigerata variabile *'contavita'* che dovrebbe tenere conto di quante volte un'oggetto è impiegato: la variabile `ctl` è autorelease, quindi, è come se contavita fosse ancora Zero. Aggiungendola al vettore `windowController`, contavita diventa 1. Attribuendola alla variabile d'istanza `coverWin`, contavita vale 2. Alla chiusura della finestra, l'ambiente operativo si preoccuperà di toglierla dal vettore `windowController`; contavita vale ancora 1. Per toglierla del tutto, a `coverWin` deve essere attribuito il valore `nil` tramite un metodo accessor in modo che il valore di contavita torni a Zero: devo quindi intercettare la chiusura della finestra, in modo da fare un altro release di `coverWin`. Per fare questo, mi avvalgo di un messaggio inviato al controllore alla chiusura della finestra da parte dell'utente, corrispondente al metodo seguente (è un metodo della classe delegata della `NSWindow`. Nel NIB tale collegamento mi pare sia eseguito di default da *Interface Builder*, altrimenti, bisogna eseguire esplicitamente la connessione):

```
- (BOOL)windowShouldClose:(id)sender
{
    [ [ self document ] setCoverWin: nil ] ;
    return ( YES );
}
```

Il metodo permette di fermare il processo di chiusura restituendo `NO` (perché ad esempio si vuole terminare qualche operazione in corso). Nel mio caso, è solo un utile momento per impostare a `nil` la variabile d'istanza.

In effetti, all'inizio, mi era sembrato più promettente un altro metodo delegato:

```
- (void)windowWillClose:(NSNotification *)aNotification;
```

Tuttavia, la cosa non funzionava. Eseguendo l'applicazione col debugger, ho scoperto che prima è chiamato `windowShouldClose` e poi `windowWillClose` (e fin qui è abbastanza ovvio). In effetti, tra la prima e la seconda chiamata, cambia un dettaglio essenziale: al momento dell'esecuzione del secondo metodo, la classe controllore ha perso ogni collegamento con il documento (la variabile d'istanza è `nil`), e quindi l'istruzione

```
    [ [ self document ] setCoverWin: nil ] ;
```

non può produrre alcun risultato sensato.

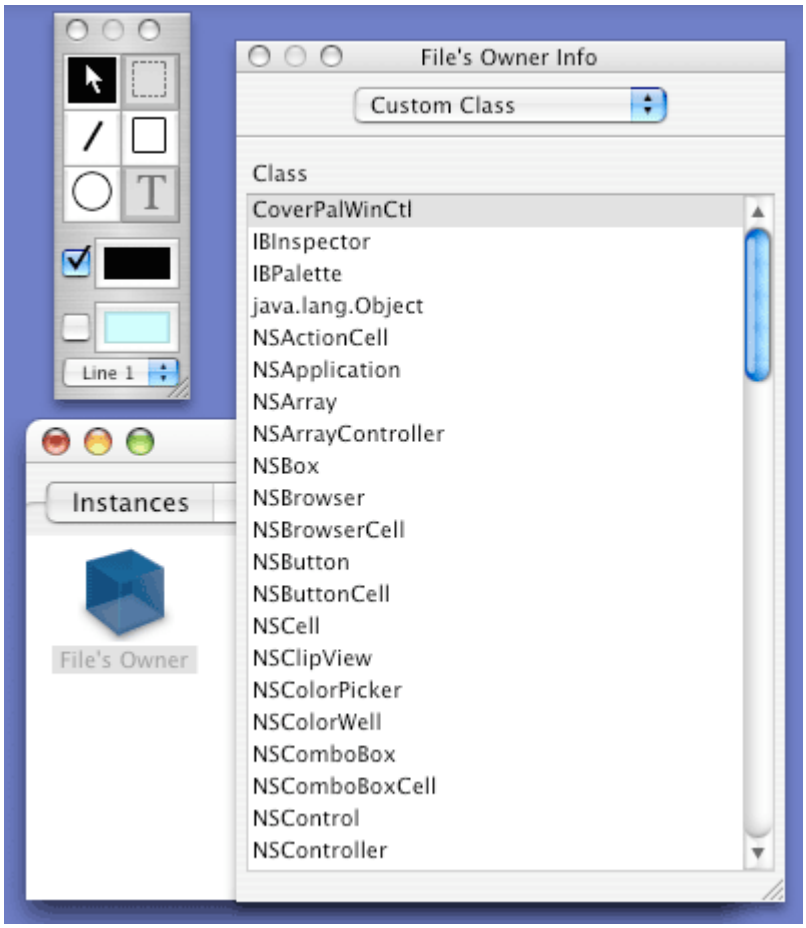
Capitolo 042 - Eppur si disegna

In questo capitolo si comincia a fare sul serio: comincio a tracciare qualche linea all'interno della coverView, usando direttamente il mouse.

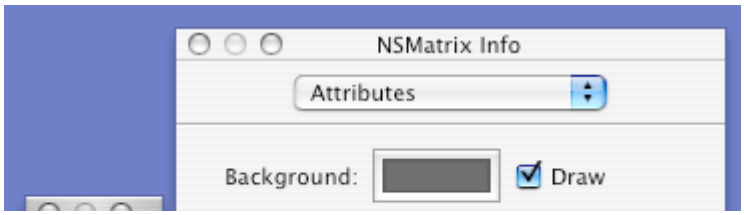
Piglio come al solito spunto dall'esempio Sketch fornito da Apple.

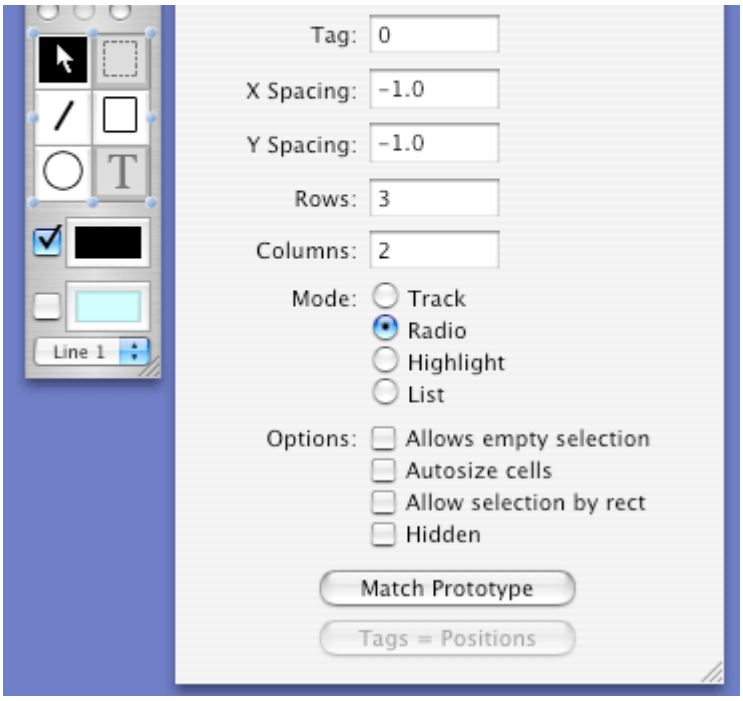
Primo inserimento: 17 febbraio 2004

LA PALETTE DEGLI STRUMENTI

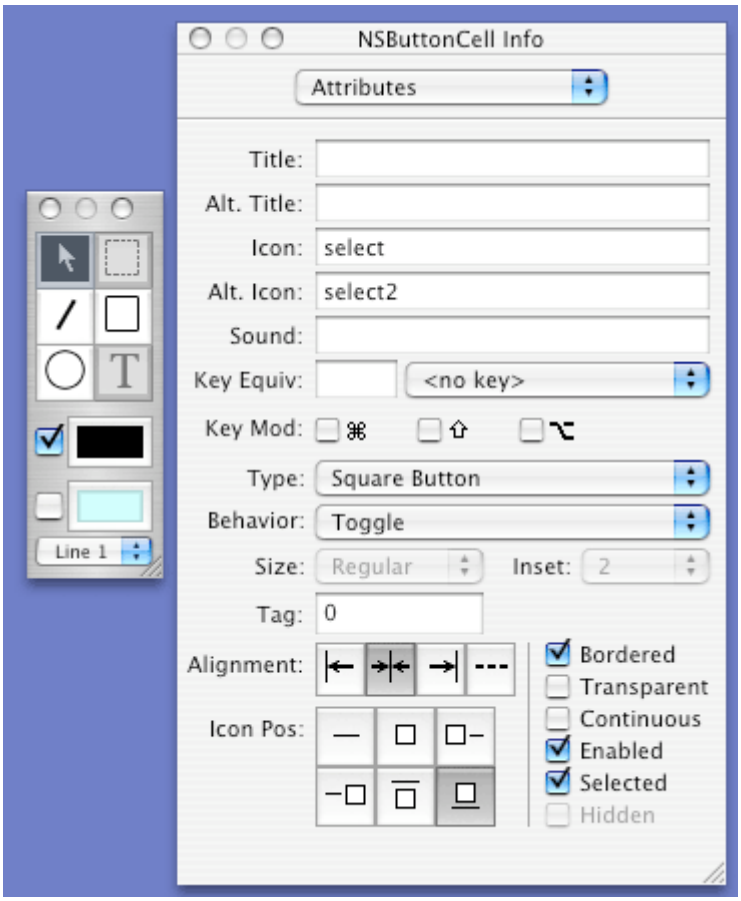


Come ogni buona applicazione per disegnare qualcosa, occorre predisporre una palette degli strumenti. Mi disegno quindi con Interface Builder una simpatica finestra che contenga qualche strumento di facile uso. Comincio con una matrice di pulsanti, per avere uno strumento di selezione, uno strumento per le linee, uno per i rettangoli, per le ellissi; per i testi lo prevedo ma lo lascio disabilitato, poi mi resta un buco vuoto, e lo lascio lì per future estensioni.





Aggiungo poi due `NSColorWell` con pulsante di spunta al seguito, per selezionare il colore del contorno e dell'eventuale riempimento. Infine, un menu pop-up dove è possibile selezionare alcuni spessori notevoli.



Ho già utilizzato una matrice di oggetti, quando ho preparato la finestra per la selezione delle preferenze di visualizzazione delle colonne. Da lì riciclerò concetti ed

anche pezzi di codice. La gestione delle `NSColorWell` ho visto essere cosa facile, ed anche il menu pop-up non mi preoccupa. In effetti il punto più difficile di tutta la finestra è stato il disegno dei pulsanti. Sono notoriamente incapace di disegnare qualcosa di visibile senza vergogna. C'è da considerare il fatto che ogni pulsante si presenta secondo due aspetti: selezionato e deselezionato. Confesso che non ho letto attentamente la documentazione, ma ho sperimentato un po'. Ho scoperto che un comportamento decente per la palette si ottiene configurando la matrice nel modo `radio`, che consente la selezione di un solo elemento alla volta (ed automaticamente deselezionando tutti gli altri); e poi configurando i pulsanti come `square button` con comportamento (*behaviour*) `Toggle`. Ho così disegnato per ogni pulsante due bitmap, una con aspetto normale ed una con aspetto *selezionato* (che è l'aspetto normale con colori invertiti), e le ho attribuite come `Icon` e come `Alt. Icon`. Termino così le operazioni con Interface Builder e torno in XCode per scrivere un po' di codice.

FINESTRA COVERPALWINCTL

Il controllore della finestra palette non è nulla di speciale; come al solito ci sono un po' di outlet, ed una manciata di metodi per facilitare il lavoro delle altre classi

```
@interface CoverPalWinCtl : NSWindowController
{
    // elemento grafico corrente
    Class                       currCCE ;
    // elementi dell'interfaccia
    IBOutlet NSMatrix           * toolMatrix ;
    IBOutlet NSColorWell        * currLineWell ;
    IBOutlet NSColorWell        * currFillWell ;
    IBOutlet NSButton           * fillCheckbox;
    IBOutlet NSButton           * lineCheckbox;
    IBOutlet NSPopUpButton      * lineWidth;
}
+ ( id ) sharedCoverPalWinCtrl ;
- (IBAction)handleToolSel:(id)sender;
- (NSColor*) getCurrLineColor ;
- (NSColor*) getCurrFillColor ;
- (Class) getCurrClassTool ;
- (BOOL) getStrokeSts ;
- (BOOL) getFillSts ;
- (float) getLineWidth ;
@end
```

C'è qui un metodo interessante per tenere conto di quale sia l'elemento correntemente selezionato nella palette. Piuttosto che conversare un generico identificare della selezione corrente, conservo una variabile d'istanza di tipo `Class`. La variabile vale `nil` quando non sono selezionati elementi (all'interno della matrice), o meglio, quando è selezionato il pulsante con la freccia, che funziona da strumento selezione. Negli altri casi, attribuisco a tale variabile proprio la classe corrispondente all'elemento selezionato (`CCE_Line`, ad esempio). Questo costrutto mi renderà le cose interessanti al momento del disegno.

Il metodo corrispondente, invocato ad ogni clic sopra un elemento della matrice, si scrive molto facile:

```
- (IBAction)
handleToolSel:(id)sender
{
    switch ( [ [toolMatrix selectedCell] tag ] ) {
    case 0 : currCCE = nil ; break ;
    case 1 : currCCE = nil ; break ;
    case 2 : currCCE = [ CCE_Line class ] ; break ;
    case 3 : currCCE = [ CCE_Rect class ] ; break ;
    case 4 : currCCE = [ CCE_Circle class ]; break ;
    case 5 : currCCE = [ CCE_Text class ] ; break ;
    }
}
```

A seconda del valore del tag del pulsante, modifico il valore della variabile `currCCE` (lasciate per il momento perdere i casi 1 e 5, che tanto non sono mai attivati, avendo disabilitato i pulsanti corrispondenti).

Tutti gli altri metodi sono messi a disposizione per recuperare i valori degli altri controlli presenti nella finestra (colori, spessori, eccetera). Sono semplici e non ne faccio parola.

Va da sé che occorre aggiungere una voce di menu in `MainManu.nib` e metodi corrispondenti, nel solito modo, per aprire la finestra.

TRATTARE IL MOUSE

Cosa succede quando l'utente fa clic da qualche parte? L'ambiente operativo si scatena, macina posizioni e numeri, e finisce col mandare un messaggio ad un oggetto `NSResponder` (o sottoclassi relative, di cui `NSView` è un esempio classico) che reputa essere in grado di rispondere. Nel mio caso, devo fare in modo che la coverView risponda a questo tipo di messaggi. Non devo fare altro che scrivere un metodo come il seguente:

```
- (void)
mouseDown:(NSEvent *)theEvent
{
     // vado a pigliare qual e' il tool correntemente selezionato nella
finestra
            Class   selCls   =   [[CoverPalWinCtl   sharedCoverPalWinCtrl]
getCurrClassTool];
    // se e' selezionato un elemento da disegnare
    if (selCls)
    {
        // costruisco l'elemento grafico
        CCE_BasicForm * curElem = [[ [selCls alloc ] init] autorelease ] ;
        // metto a posto un di attributi base dell'elemento
         [ curElem setCceLineWidth: [[CoverPalWinCtl sharedCoverPalWinCtrl]
getLineWidth] ];
         [ curElem setCceFillColor: [[CoverPalWinCtl sharedCoverPalWinCtrl]
getCurrFillColor] ];
         [ curElem setCceLineColor: [[CoverPalWinCtl sharedCoverPalWinCtrl]
getCurrLineColor] ];
          [ curElem setCceIsFilled: [[CoverPalWinCtl sharedCoverPalWinCtrl]
getFillSts] ];
         [ curElem setCceIsStroked: [[CoverPalWinCtl sharedCoverPalWinCtrl]
getStrokeSts] ];
```

```
        // deseleziono tutto
        [ theElements setCceIsSelected: NO ] ;
        // dico che l'elemento che sto disegnando e' selezionato
        [ curElem setCceIsSelected: YES ];
        // e che si tratta dell'elemento in lavorazione
        [ self setCurCreatingElem: curElem ] ;
        // gestisco la costruzione col mouse
        if ([curElem handleCreation: theEvent inView: self ])
        {
            // se la creazione e' andata a buon fine,lo aggiungo alla lista
            [theElements addElem:curElem ];
        }
        // non ci sono piu' elementi in lavorazione
        [ self setCurCreatingElem: nil ] ;
    }
    else
    {
        // non si deve disegnare un nuovo elemento
        // tratto un generico clic all'interno della finestra
        [self handleMouseClick: theEvent];
    }
}
```

Il comportamento dipende in maniera fondamentale dall'elemento correntemente selezionato nella palette. La prima cosa che mi chiedo è allora qual è la selezione corrente, recuperando la classe selezionata. Se questa è `nil` (ramo `else`), c'è selezionato lo strumento selezione, e quindi bisogna trattare il clic del mouse in maniera standard (vedrò questo più avanti). Per il momento mi focalizzo sulla situazione in cui il clic avviene nella finestra con un elemento da disegnare.

La prima cosa da fare è ovviamente costruire un oggetto di tipo appropriato. E qui entra in gioco il discorso del paragrafo precedente. Uso direttamente la variabile `selCls` come destinataria dei messaggi `alloc` ed `init`, per costruire un elemento, nominalmente di tipo `CCE_BasicForm`, ma in realtà è proprio del tipo corretto.

Tramite i metodi di comodo, attribuisco i valori correnti per linee, spessori, colori, eccetera.

Tralascio le due istruzioni successive, che riprendo più tardi parlando della selezione degli elementi. Poi, cominciano i problemi. Non commento l'assegnazione dell'elemento corrente come elemento in lavorazione (ci sarà un paragrafo dedicato), e passo a gestire la creazione dell'elemento con il metodo `handleCreation:inView:`; essenzialmente, il cuore della faccenda.

CREARSI DA SOLI

L'idea è di lasciare che ogni elemento tratti da sé la procedura di creazione. In questo modo è facile (polimorfismo!) gestire le diverse necessità di ogni oggetto in termini di operazioni richieste. Per il momento, ho diviso le operazioni richieste in due parti, la prima più o meno generica per ogni elemento, la seconda più specifica.

Quando si usa il mouse per disegnare una linea, un rettangolo, eccetera, in realtà si finisce spesso con l'individuare due punti: quello in cui è avvenuto il clic del mouse (*mouseDown*) e quello dove avviene il rilascio del mouse (*mouseUp*). Nel mezzo,

tramite operazioni di *drag* (spostamento del mouse con pulsante premuto), si procede al disegno di una linea o di un rettangolo. Ecco il motivo per cui ho messo alla base della creazione di un elemento un ciclo che continua ad aggiornare l'elemento in corso di creazione basandosi appunto sul punto di mouseDown e sul punto di mouseUp.

```
- (BOOL)
handleCreation: (NSEvent *)theEvent
    inView:        (CoverView *)view
{
    // recupero la coordinata locale
        NSPoint  point1  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
    NSPoint point ;
    // inizializzo l'aspetto dell'oggetto (praticamente vuoto)
    [ self buildMeWithStartPt: point1 andPt: point1 ] ;
    // continuo a ciclare in attesa di un mouseup
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
                                        theEvent     =     [[view     window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
        // recupero la coordinata locale
                  point  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
        // aggiusto l'aspetto dell'oggetto
        [ self buildMeWithStartPt: point1 andPt: point ] ;
        // forzo un ridisegno della finestra
        [ view setNeedsDisplay: YES ];
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
    // arrivato qui, ho finito il tutto
    return YES;
}
```

Il metodo comincia mettendo da parte (una volta convertite le coordinate del punto di clic dal sistema di riferimento della finestra al sistema di riferimento della coverView) il punto di mouseDown. Poi comincia un ciclo di operazioni, che sono ripetute fino a che non è intercettato un evento di mouseUp.

Per prima cosa si recuperano le coordinate correnti del mouse. Con queste coordinate si procede alla costruzione dell'elemento attraverso il metodo `buildMeWithStartPt:andPt:`, per poi concludere il tutto forzando il ridisegno della vista.

Prima di capire qualche dettaglio, è bene vedere il metodo di costruzione, ad esempio nel caso semplice della classe `CCE_Rect`:

```
- (void)
buildMeWithStartPt: (NSPoint) stPt andPt: (NSPoint) endPt
{
        float maxPx = ( stPt.x > endPt.x ) ? stPt.x : endPt.x ;
        float maxPy = ( stPt.y > endPt.y ) ? stPt.y : endPt.y ;
        float minPx = ( stPt.x < endPt.x ) ? stPt.x : endPt.x ;
        float minPy = ( stPt.y < endPt.y ) ? stPt.y : endPt.y ;
          NSRect aRect = NSMakeRect( minPx, minPy, maxPx - minPx, maxPy -
minPy );
        // per mia convenienza conservo il percorso
        NSBezierPath *path = [NSBezierPath bezierPathWithRect: aRect ];
```

```
        [ self setTheRect: path ];
        // metto a posto il rettangolo che lo contiene
        [self setElemLimit: aRect ];
}
```

Dati i due punti, determino le coordinate del punto in alto a sinistra (le coordinate verticali sono rovesciate) e le due grandezze larghezza ed altezza per determinare col metodo tipico `bezierPathWithRect:` il percorso che rappresenta il rettangolo.

Il metodo equivalente per una ellisse è quasi uguale, un po' diverso quello per la costruzione di una linea.

```
- (void)
buildMeWithStartPt: (NSPoint) stPt andPt: (NSPoint) endPt
{
        // per mia praticita', costruisco un percorso
        NSBezierPath *path = [NSBezierPath bezierPath];
        // assegno i punti di partenza ed arrivo
        startPoint = stPt ;
        endPoint = endPt ;
        // costruisco il percorso
        [path moveToPoint: startPoint ];
        [path lineToPoint: endPoint ];
        // lo assegno
        [ self setTheLine: path ];
        // metto a posto il rettangolo che lo contiene
        [self setElemLimit: [theLine bounds] ];
        // variabile di appoggio
          delta = 1.0F / sqrt( ( endPoint.x - startPoint.x)*( endPoint.x -
startPoint.x) +
                  ( endPoint.y - startPoint.y)*( endPoint.y - startPoint.y)
);
}
```

In questo caso (a parte l'ultima istruzione, che servirà più avanti) ho praticamente estratto dal metodo di inizializzazione tutte le istruzioni e le ho inserite qui. In effetti, il metodo è diventato:

```
-(id) initStartPt: (NSPoint) stPt toPt: (NSPoint) endPt
{
    self = [ super init ];
    if ( self )
    {
        [ self buildMeWithStartPt: stPt andPt: endPt ];
        [ self setCceIsStroked: TRUE ];
    }
    return ( self );
}
```

L'ELEMENTO CORRENTE

Mi sembrava di aver fatto tutto giusto e correttamente, eppure, non riuscivo a disegnare. Anzi no, la linea o il rettangolo disegnato comparivano, ma solo dopo il completamento dell'operazione, mentre a me interessava vedere il movimento elastico di un rettangolo mentre viene tracciato. Dopo lungo penare, ho capito il problema.

Chiamando il metodo `setNeedsDisplay` all'interno del ciclo di tracciatura del mouse, si forza il ridisegno della finestra. Ma il ridisegno dipinge tutti e solo gli elementi che si trovano all'interno del gruppo `theElements` della coverView. E l'elemento che sto disegnando non è ancora inserito all'interno di questo gruppo, ma lo sarà solamente al termine delle operazioni (e questo spiega il comportamento bizzarro).

Come gestire allora questo trascinamento dal vivo? Piuttosto che aggiungerlo al gruppo, ho deciso di tenerlo da parte, in una variabile d'istanza apposita della coverView.

```
    CCE_BasicForm        * curCreatingElem ;
```

Questa variabile ha normalmente il valore `nil`, e nulla accade. Quando si comincia a tracciare con il mouse, all'interno del metodo `mouseDown:`, assegno a questa variabile l'elemento appena creato

```
        [ self setCurCreatingElem: curElem ] ;
```

per poi distruggerlo al termine delle operazioni (ma non prima che questo sia appunto aggiunto al gruppo).

Tutto questo allo scopo di modificare il metodo che effettua il disegno della vista come segue:

```
- (void)drawRect:(NSRect)rect
{
    // sfondo bianco
    [[ NSColor whiteColor] set ];
    NSRectFill( rect );
    // poi dico di disegnarsi a tutti gli elementi
    [ [self theElements] drawElement: rect ];
    // e poi all'elemento in corso
    if ( curCreatingElem )
    {
        [ curCreatingElem drawElement: rect ] ;
    }
}
```

Semplicemente, quando si è in corso di creazione di elementi, si disegna anche l'elemento in corso di creazione. Così facendo, si ottiene il risultato sperato. Già che ci sono, ci sono due modifiche ai metodi propri dell'elemento gruppo.

La prima è banale ma pregna di conseguenze, e riguarda l'ordine con cui sono ridisegnati gli elementi del gruppo.

```
- (void) drawElement: (NSRect) aRect
{
    int        i , numElem ;
    CCE_BasicForm * elem ;

    numElem = [ elemArray count ];
    for ( i = numElem-1 ; i >= 0 ; i -- )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem drawElement: aRect ];
```

```
    }
}
```

Invece che eseguire la classica spazzolata dall'indice zero all'indice massimo, il ridisegno avviene in senso inverso. Parto dall'idea che gli elementi a indice più basso si trovano davanti ad elementi di indice più alto, ovvero che i primi coprono, se sovrapposti, i secondi. Per avere un buon disegno, occorre quindi partire dall'elemento dietro tutti gli altri, e procedere verso quelli davanti.

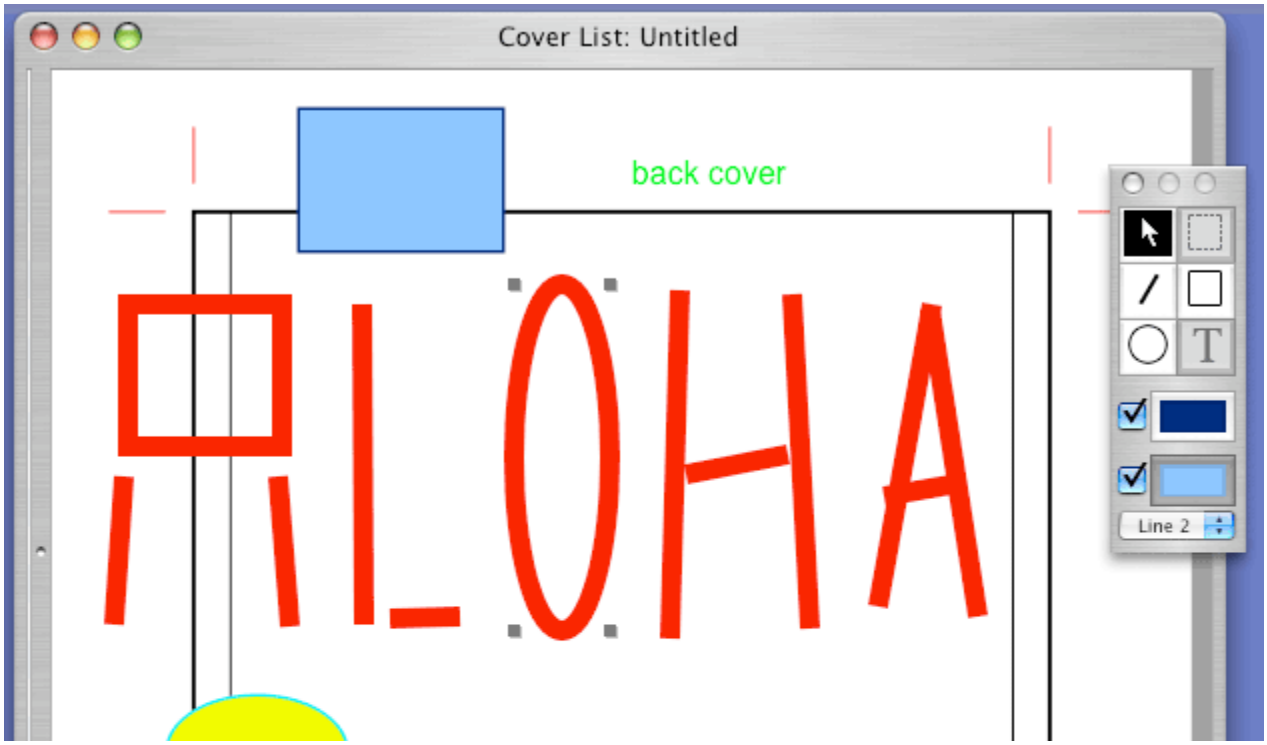
Allo stesso modo, decido che quando aggiungo un elemento ad un gruppo, lo metto nella posizione davati a tutti gli altri (perché si suppone che sia stato creato per ultimo).

```
- (void) addElem: (CCE_BasicForm *) elem
{
    // facile
    //    [ elemArray addObject: elem ];
    // troppo facile; inserisco davanti a tutto
    [ elemArray insertObject: elem atIndex:0];
}
```

Il metodo standard `addObject:`, come si evince dalla documentazione, infila il nuovo alla fine del vettore `NSMutableArray`. Tocca quindi indicare esplicitamente la posizione dove inserire il nuovo elemento.

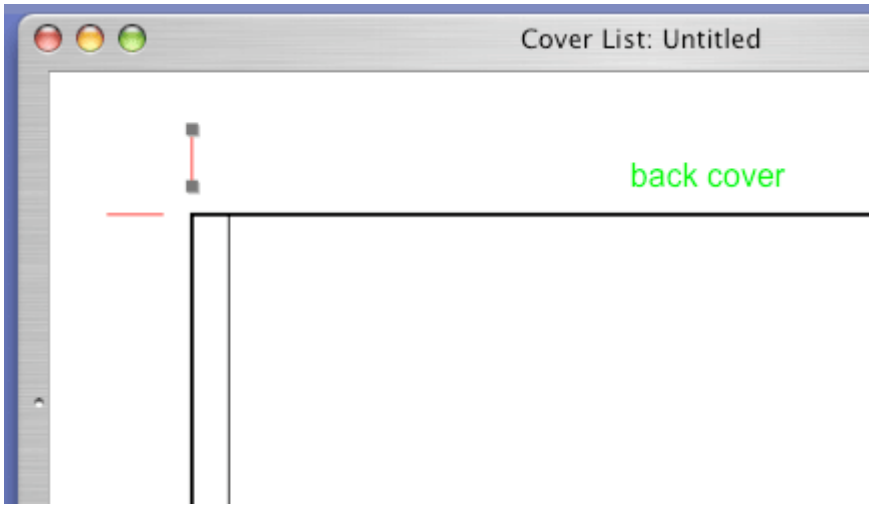
(Ora, i più accorti tra voi potrebbero osservare: ma se io lascio tutto com'era prima, funziona uguale: aggiungo elementi alla fine, ridisegno partendo dall'indice più basso, basta dire che gli elementi d'indice più basso si trovano dietro a quelli d'indice più alto, rovesciando la convenzione. Vero. Ma a me piace la convenzione opposta, e quindi va bene così).

MANICI



Arrivato a questo punto, mi trovo con una applicazione in grado di disegnare linee, rettangoli ed ellissi a mano libera, con contorni ed interni di colore a piacere, ed altre piacevolezze. Molto bello.

Ma rimane da trattare la parte del mouse quando si fa clic senza l'intento di disegnare nuovi elementi. Avevo infatti lasciato in sospeso la realizzazione del metodo `handleMouseClick:`, che appunto interviene quando è attivo lo strumento selezione.



Ma prima, devo riscrivere i metodi di disegno degli elementi per tenere conto del fatto che un elemento può essere selezionato o meno. Normalmente, quando in un programma di disegno, si seleziona un elemento, questo è evidenziato in qualche modo. Spesso, cambia aspetto, e viene ridisegnato mettendo bene in evidenza alcuni punti notevoli dell'elemento grafico. Ad esempio, selezionando una linea, questa può presentarsi con gli estremi evidenziati tramite un quadratino. Generalmente, attraverso questi quadratini l'utente può poi manipolare l'elemento, cambiandone l'aspetto successivamente alla creazione (è stupefacente come operazioni ormai naturali a tutti richiedano così tante parole per essere descritte). Per questo motivi i quadratini sono spesso chiamati *handle* o manici.

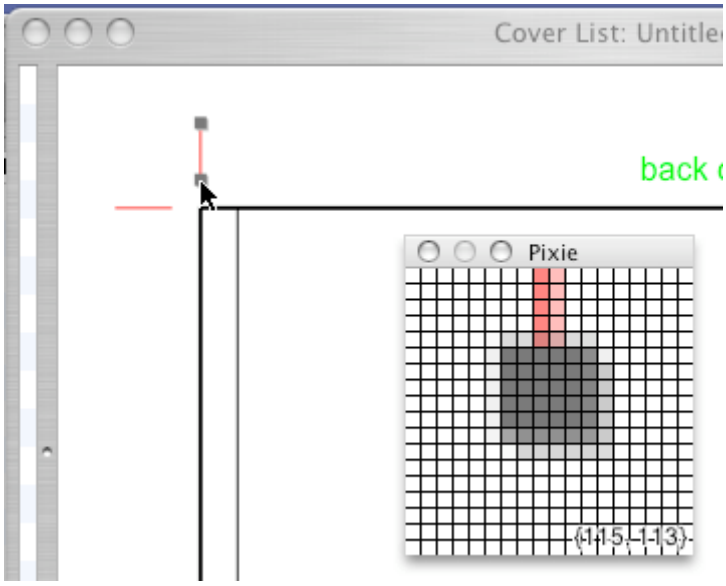
Nel mio caso, ecco una possibile realizzazione nel caso di un `CCE_Line`:

```
- (void) drawElement: (NSRect) inRect
{
    // imposto il colore
    [ cceLineColor set ];
    // imposto lo spessore della linea
```

```
    [ theLine setLineWidth: [ self cceLineWidth ]];
    // disengo la linea
    [ theLine stroke ];
    // se l'elemento e' selezionato
    if ( cceIsSelected )
    {
        // disegno i manici per la manipolazione
        ccex_drawHandleAt( startPoint );
        ccex_drawHandleAt( endPoint );
    }
}
```

Se la variabile d'istanza `cceIsSelected` è vera, chiamo due volte una funzione per disegnare un handle, uno nel punto iniziale ed uno nel punto finale. Nel caso di un rettangolo o di una ellisse, chiamerò la funzione quattro volte, per i quattro vertici del rettangolo stesso (o quello che include l'ellisse). La funzione è piuttosto pedestre:

```
void
ccex_drawHandleAt( NSPoint where )
{
    // per ora la dimensione del manico e' fissa
    int        handleDim = 3 ;
    NSRect    hRect, rRect ;
    // disegno un rettangolo pieno attorno al punto
    hRect = NSMakeRect( where.x, where.y,2*handleDim, 2*handleDim );
    rRect = NSOffsetRect( hRect , -handleDim, -handleDim);
    hRect = NSOffsetRect(rRect, 1.0, 1.0);
    [[NSColor lightGrayColor] set];
    NSRectFill(hRect);
    [[NSColor darkGrayColor] set];
    NSRectFill(rRect);
}
```



Uno handle è un quadrato centrato attorno al punto specificato come parametro. La variabile `handleDim` rappresenta metà della dimensione dello handle (tre pixel).

Determinato il rettangolo a partire dal punto selezionato, calcolo il nuovo rettangolo `rRect` centrandolo prorpio sul punto. Poi costruisco un altro rettangolo, spostato di un pixel, a mo' di ombra.

STRUMENTO SELEZIONE

Finalmente, il metodo per gestire un clic di selezione.

```
- (void)
handleMouseClick: (NSEvent *)theEvent
{
    NSPoint whereClick;
    CCE_BasicForm * curElem = nil;
    // vado a vedere dove e' avvenuto il click
          whereClick   =   [self   convertPoint:[theEvent   locationInWindow]
fromView:nil];
    // niente niente l'utente vuole selezionare un elemento
    curElem = [self getClickedElem:whereClick];
    // se c'e' un elemento sotto il mouse
    if ( curElem )
    {
        // abbiamo qualcosa sotto il mouse
        // se e' gia' selezionato, lo deseleziono
        if ( [ curElem cceIsSelected ] )
            [ curElem setCceIsSelected: NO ] ;
        else
        {
            // devo deselezionare tutto
            [ theElements setCceIsSelected: NO ] ;
            // e poi seleziono questo elemento
            [ curElem setCceIsSelected: YES ];
        }
    }
    else
    {
        // c'e' nulla sotto il mouse, deseleziono tutto
        [ theElements setCceIsSelected: NO ] ;
    }
    // ora che arrivo qui, faccio nulla, aspetto mouseup
    while (1)
    {
         theEvent = [[self window] nextEventMatchingMask: NSLeftMouseUpMask
];
        if ([theEvent type] == NSLeftMouseUp)
        {
            break;
        }
    }
    // ridisegno tutto, in ogni caso
    [ self setNeedsDisplay: YES ];
}
```

Il clic può avvenire in zona di nessuno (c'è nulla sotto il mouse), oppure il clic avviene proprio sopra un elemento. Nel primo caso, mi pare un buon comportamento deselezionare tutti gli elementi eventualmente selezionati (ramo `else` dello `if` principale). Se invece il clic avviene sopra un elemento (e questo lo scoprirò meglio quando esamino il metodo `getClickedElem:`), il comportamento dipende se l'elemento è selezionato oppure no. Nel primo caso, lo deseleziono, nel secondo, lo seleziono. Lo scenario cambierà quando dovrò considerare selezioni multiple, tasti modificatori e cose del genere. Per il momento mi accontento della situazione (che, va da sé, significa che c'è sempre e solo al massimo un elemento selezionato).

Il metodo si chiude con un ciclo di attesa del rilascio del mouse (che fa nulla; ma poi

qui andranno inserite le istruzioni per gestire le selezioni di più elementi trascinando il mouse in giro per la coverView), e forzando il ridisegno della vista stessa.

Interessante il meccanismo utilizzato per la deselezione di tutti gli elementi selezionati. In effetti, basta spazzolare tutti gli elementi presenti in `theElements` e porre la variabile `cceIsSelected` a falso. Ma io, astutamente, ho riscritto il metodo accessor per la classe `CCE_ElemGroup`.

```
- (void)
setCceIsSelected:(BOOL)newIsSelected
{
    int        i , numElem ;
    CCE_BasicForm * elem ;
    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem setCceIsSelected: newIsSelected ];
    }
}
```

In questo modo basta inviare il messaggio `setCceIsSelected` solo all'elemento `theElements`, che questo scatenerà per tutti gli elementi ed i gruppi previsti l'operazione richiesta (quando qualche capitolo fa parlavo della fecondità del concetto di elemento gruppo, pensavo a tutti questi giochini).

Bisogna adesso capire se il clic avviene sopra un elemento. È un lavoro (invero piuttosto banale) per:

```
- (CCE_BasicForm *)
getClickedElem:(NSPoint)point
{
    int                i , numElem ;
    CCE_BasicForm * elem ;
    NSMutableArray        * listaelem ;
    // esamino tutti gli elemento
    listaelem = [ theElements elemArray ];
    numElem = [ listaelem count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ listaelem objectAtIndex: i ];
        // e controllo se il click gli appartiene
        if ( [ elem checkClickOnMe: point ])
        {
            // ho finito, esco dal ciclo
            break;
        }
    }
    // potrei non avere trovato nulla
    if ( i == numElem )
        return ( nil );
    // se arrivo qui, ho trovato un elemento
    return ( elem );
}
```

Basta chiedere ad ogni elemento nel vettore `theElements` se il clic è avvenuto sopra di lui, e fermarsi non appena qualche oggetto risponde sì. Osservare bene l'ordine delle

richieste: si parte dall'elemento davanti agli altri (indici bassi) per poi passare, in caso di risposte sempre negativi, ad elemento dietro agli altri, con indici più alti. Ancora una volta, il grosso del lavoro è svolto altrove dal metodo `checkClickOnMe:`, che sfrutta ancora una colta il meccanismo del polimorfismo.

C'è comunque una realizzazione di default di questo metodo per la classe `CCE_BasicForm`, che spesso va bene anche per elementi più specializzati.

```
- (BOOL)
checkClickOnMe:(NSPoint)point
{
     // in generale, controllo se il clik e' avvenuto dentro il rettangolo
limite
    return (NSPointInRect(point, [self elemLimit])) ;
}
```

C'è una utile funzione che restituisce vero o falso a seconda se il punto (primo argomento) si trova nel rettangolo specificato come secondo argomento. Ad esempio, questo metodo è perfetto per un rettangolo, ma non va bene per un cerchio e per una linea. Per un cerchio, me la cavo ancora con poco:

```
- (BOOL)
checkClickOnMe:(NSPoint)point
{
    return ([theCircle containsPoint:point]) ;
}
```

Sfrutto uno dei metodi della classe `NSBezierPath` per capire se il punto sta o no dentro il percorso individuato dal path.

Con una linea, è un pasticcio. Un path lineare non contiene un punto, non c'è verso. Occorre inventare qualcos'altro. Ciò che mi è venuto in mente (ma ho visto che Sketch non se l'è cavata molto meglio) è di ricorrere ai miei studi liceali e di calcolare la distanza del punto di clic dalla retta. Si tratta di una formula di geometria analitica. Se non avete la più pallida di cosa sto parlando, prendete per buono il tutto ed andate avanti fiduciosi. Se sapete di cosa sto parlando, troverete tutto molto semplice (ma vi lascio fare un po' di calcoli se non siete convinti).

La formula che calcola la distanza del punto di coordinate `(x0, y0)` dalla retta `ax+by+c=0` è:

```
d = abs(a x0 + b y0 + c ) / sqrt( a^2 + b^2)
```

Onde evitare di fare troppi calcoli quando non servono, nella classe aggiungo una variabile d'istanza `delta` che tenga da parte il valore

```
delta = 1 / sqrt( a^2 + b^2).
```

In questo modo, il metodo diventa:

```
- (BOOL)
checkClickOnMe:(NSPoint)point
```

```
{
    float ev ;
    NSRect locRect ;
    // rettangolo in cui si trova la linea
    locRect = NSInsetRect( [self elemLimit], -2, -2 );
    // se sono dentro, proseguo
    if (NSPointInRect(point, locRect ) )
    {
        // retta per un punto, e distanza di un punto da una retta
        ev = ( endPoint.x - startPoint.x)* (point.y - startPoint.y) -
            ( endPoint.y - startPoint.y)* (point.x - startPoint.x) ;
        if ( ev < 0 ) ev = -ev ;
        // se la distanza e' inferiore allo spessore della linea piu'
        // una certa tolleranza per essere sicuri di beccare il punto...
        if ( ev * delta < ([ self cceLineWidth] + 3) )
            return ( TRUE ) ;
    }
    // se arrivo qui, mancato!
    return ( FALSE );
}
```

Per evitare calcoli quando non sono palesemente utili (ed anche alcuni divertenti problemi geometrici), mi chiedo in primo luogo se il punto cade dentro nel rettangolo che racchiude la linea. Ho considerato un rettangolo leggermente più grande per tenere conto di linee esattamente verticali od orizzontali (che avrebbero una delle due dimensioni nulle!). Solo se il punto si trova dentro questo rettangolo, proseguo col controllo. Alla fine verifico se la distanza del punto della retta (data dal prodotto `ev*delta`) è inferiore allo spessore della linea, aumentato di un valore costante per tenere conto che lo spessore potrebbe anche essere nullo (e per dare una tolleranza nel clic, altrimenti selezionare una linea sottile diventa una operazione da certosini).

PICCOLEZZE

Concludo con una piccolezza, ma è nei dettagli che si vede la grandezza del programmatore. Trovavo estremamente fastidioso il comportamento congiunto della palette e della finestra quando provavo a disegnare un elemento. Selezionavo ad esempio lo strumento rettangolo, facevo clic sulla coverView, e questo era sprecato per selezionare la finestra e portarla di fronte a tutte. Così, per disegnare, dovevo fare un primo clic per selezionare la finestra, e poi un secondo (tenendo il mouse premuto) per disegnare effettivamente l'elemento. Trovo molto più naturale che il primo clic non vada sprecato, ma utilizzato come punto di partenza di un nuovo elemento (la necessità di un doppio clic si ha anche negli altri casi, ad esempio, con lo strumento selezione occorrono due clic per selezionare un elemento).

Dopo averci bestemmiato un po' sopra, ho scoperto (navigando nella documentazione, come al solito), un metodo (di `NSView`) che fa proprio al caso mio:

```
- (BOOL)acceptsFirstMouse:(NSEvent *)theEvent
{
    return YES;
}
```

Neanche farlo apposta, la realizzazione di default restituisce NO, proprio perché il primo mouseDown in una vista è normalmente impiegato per selezionarla. Ma spesso,

dice la documentazione, non è quello che si vuole...

Capitolo 043 - Drag Zoom Zoom

In questo capitolo aggiungo la possibilità di spostare gli elementi grafici e un meccanismo per ingrandire e rimpicciolire la coverView.

Ancora una volta, piglio esempio da Sketch.

Primo inserimento: 26 febbraio 2004

SPOSTARE ELEMENTI

Dopo aver scritto le istruzioni per creare elementi e per selezionarli, il passo successivo nella manipolazione diretta degli elementi col mouse è certamente la possibilità di spostare gli elementi come si preferisce. Per fare questo occorre modificare un po' il metodo principale di manipolazione degli elementi all'interno della coverView. Riscrivo parzialmente il metodo `handleMouseClick:`, che interviene quando l'utente fa clic senza l'intenzione di creare nuovi elementi.

```
- (void)
handleMouseClick: (NSEvent *)theEvent
{
    NSPoint whereClick;
    CCE_BasicForm * curElem = nil;
    // vado a vedere dove e' avvenuto il click
          whereClick   =   [self   convertPoint:[theEvent   locationInWindow]
fromView:nil];
    // niente niente l'utente vuole selezionare un elemento
    curElem = [self getClickedElem:whereClick];
    // se c'e' un elemento sotto il mouse
    if ( curElem )
    {
        // abbiamo qualcosa sotto il mouse
        if ( ! [ curElem cceIsSelected ] )
        {
            // devo deselezionare tutto
            [ theElements setCceIsSelected: NO ] ;
            // e poi seleziono questo elemento
            [ curElem setCceIsSelected: YES ];
            // ridisegno tutto che la vista e' cambiata...
            [ self setNeedsDisplay: YES ];
        }
        // provo a vedere se e' il caso di muoverlo
        [ curElem handleMoveElem: theEvent inView: self];
    }
    else
    {
        // c'e' nulla sotto il mouse, deseleziono tutto
        [ theElements setCceIsSelected: NO ] ;
    }
    // ridisegno tutto, in ogni caso
    [ self setNeedsDisplay: YES ];
}
```

Le tre possibilità a questo punto diventano: faccio clic in zona di nessuno, ed allora deseleziono tutto; faccio clic sopra un elemento attualmente deselezionato, allora lo

seleziono; se invece l'elemento è selezionato, mi appresto a muoverlo. Tuttavia, risulta naturale entrare nella gestione del movimento anche in seguito al clic su di elemento non selezionato (ovvero, il secondo caso prospettato), per permettere un movimento fluido all'utente di selezione e spostamento dell'elemento.

Ancora una volta, il meccanismo di gestione del movimento è diviso in più parti, lasciando comunque all'elemento stesso la gestione del movimento.

La prima parte del meccanismo è comune a tutti gli elementi: si tratta quindi di un metodo della classe `CCE_BasicForm`, che gestisce gli eventi di spostamento del mouse.

```
- (void)
handleMoveElem: (NSEvent *)theEvent
    inView:        (CoverView *)view
{
    // recupero la coordinata locale
        NSPoint  point1  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
    NSPoint point2 ;
    // ciclo piu' o meno lungo di attesa eventi
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
                                        theEvent     =     [[view     window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
        // recupero la coordinata locale
                point2  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
        // aggiusto l'aspetto dell'oggetto
        [ self shiftMeFrom: point1 to: point2 ];
        // il nuovo punto di partenza
        point1 = point2 ;
        // forzo un ridisegno della finestra
        [ view setNeedsDisplay: YES ];
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
}
```

Si piglia e si mette da parte un primo punto, che all'inizio è quello dove l'utente ha fatto mouseDown. Poi si apre un bel ciclo in attesa del rilascio del mouse. Nel frattempo, si rintracciano tutti i movimenti (ecco il motivo della presenza della costante `NSLeftMouseDraggedMask`). Recupero il nuovo punto corrente, ed eseguo lo spostamento dell'elemento selezionato dal primo punto a questo secondo punto. Una volta eseguito questo spostamento, dico che il nuovo punto diventa il nuovo punto di partenza, e ricomincio il ciclo (non prima di aver forzato il ridisegno della coverView).

Ancora una volta, si sfrutta il polimorfismo degli elementi per eseguire lo spostamento in maniera propria per ciascun elemento, semplicemente realizzando in maniera opportuna il metodo

```
- (void) shiftMeFrom: (NSPoint) pt1 to: (NSPoint) pt2 ;
```

per ogni elemento da disegnare. Come al solito, mi limito a mostrare solo un paio di realizzazioni: la prima è per l'elemento linea `CCE_Line`:

```
- (void)
shiftMeFrom: (NSPoint) pt1 to: (NSPoint) pt2
{
    float deltax = pt2.x - pt1.x ;
    float deltay = pt2.y - pt1.y ;
       NSPoint  p1  =  NSMakePoint(  startPoint.x  +  deltax,  startPoint.y  +
deltay);
    NSPoint p2 = NSMakePoint( endPoint.x + deltax, endPoint.y + deltay);
    [ self buildMeWithStartPt: p1 andPt: p2 ];
}
```

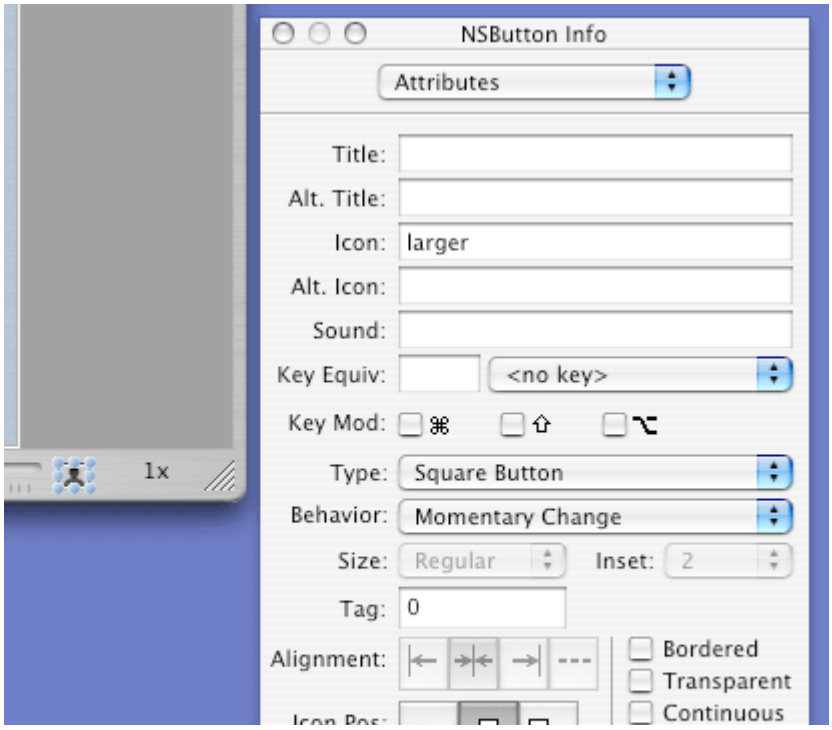
Qui riutilizzo il comodo metodo di costruzione `buildMeWithStartPt:andPt:` dopo aver calcolato i due nuovi punti limite della linea.

Per il rettangolo invece il metodo più stupido che mi è venuto in mente è il seguente:

```
- (void)
shiftMeFrom: (NSPoint) pt1 to: (NSPoint) pt2
{
    float deltax = pt2.x - pt1.x ;
    float deltay = pt2.y - pt1.y ;
    // sposto il rettangolo di base
    [ self setElemLimit: NSOffsetRect(elemLimit, deltax, deltay)];
            [   self   buildMeWithStartPt:   NSMakePoint(elemLimit.origin.x,
elemLimit.origin.y)
            andPt: NSMakePoint( elemLimit.origin.x + elemLimit.size.width,
                                elemLimit.origin.y + elemLimit.size.height)
];
}
```

dove sposto il rettangolo limite secondo quanto indicato, e poi ricostruisco il rettangolo con i nuovi punti limite.

FACILI ZOOM



Pensavo di dover penare come pochi per realizzare un meccanismo di zoom (in ingrandimento, ma anche in riduzione). In realtà è una cosa facilissima ed assai trasparente per ogni vista. In altre parole, gli oggetti `NSView` presentano incorporati meccanismi di zoom d'uso immediato.

Ma comincio con il preidsporre l'interfaccia utente per fare lo zoom. Mi ispiro un po' a quello che fa l'applicazione `iPhoto`. Predispongo cioè in basso a destra della finestra uno slider con due piccole immagini ai lati. Uso lo slider come un controllo per dare il fattore di zoom, e le immagini ai lati per fissare lo zoom massimo e lo zoom minimo.

Per fare tutto questo, ho predisposto la classe `CoverWinCtl` affinché presentasse un nuovo outlet e qualche metodo aggiuntivo da utilizzare per il meccanismo target/action:

```
@interface CoverWinCtl : NSWindowController
{
    IBOutlet CoverView * cvrView;
    IBOutlet NSTableView * volList;
    IBOutlet NSSlider * zoomSlider;
}
- (void) setMaxSlider: (id)sender ;
- (void) setMinslider: (id)sender ;
- (void) updateZoom: (id)sender ;
...
@end
```

L'outlet `zoomSlider` serve per impostare da programma il valore dello zoom, o semplicemente per leggerlo. I primi due metodi sono chiamati rispettivamente dal pulsante a destra (per fissare il massimo fattore di zoom) e a sinistra (per fissare il minimo fattore di zoom, o meglio il massimo fattore di rimpicciolimento) dello slider; il terzo metodo è quello chiamato dallo slider quando l'utente ci sta pasticciando sopra.

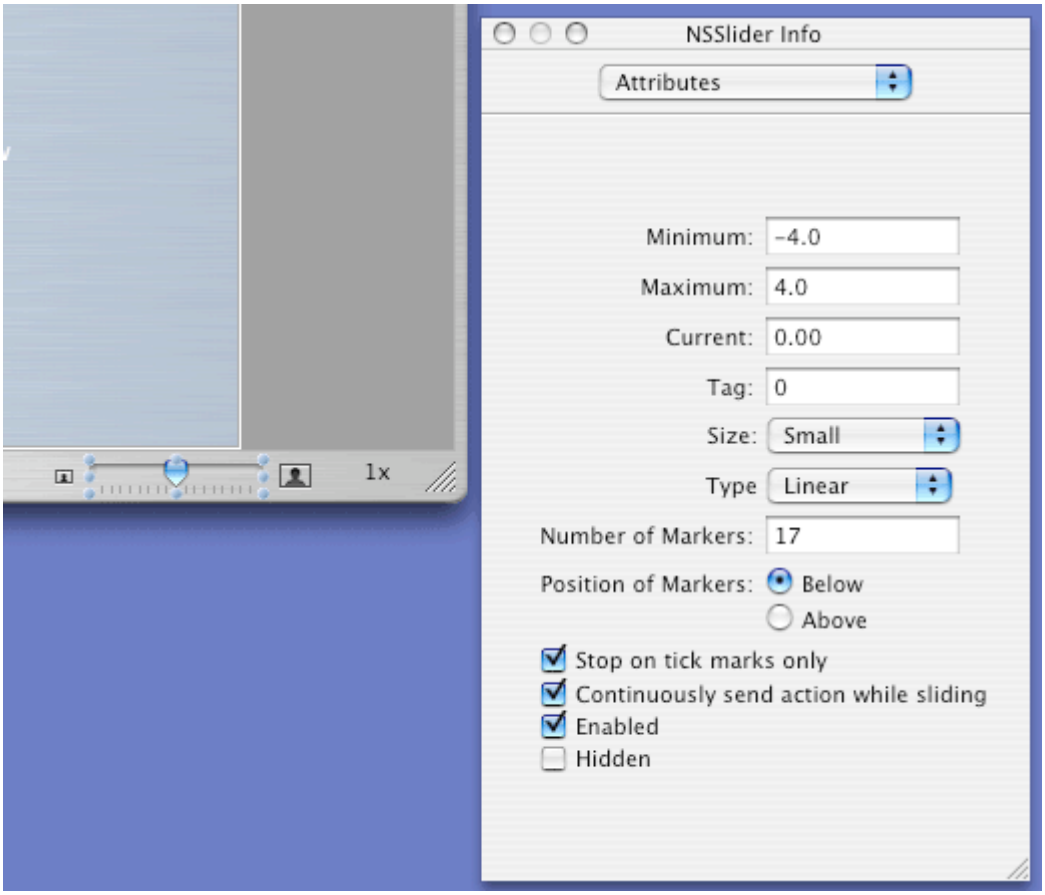
I primi due metodi sono di immediata realizzazione:

```
- (void)
setMaxSlider: (id)sender
{
    [ zoomSlider setFloatValue: [ zoomSlider maxValue]];
    [ self updateZoom: sender];
}
- (void)
setMinslider: (id)sender
{
    [ zoomSlider setFloatValue: [ zoomSlider minValue]];
    [ self updateZoom: sender];
}
```

Dopo aver impostato il valore corrente dello slider, chiama il terzo metodo simulando in pratica un movimento dello slider da parte dell'utente.

Il terzo metodo è stringato ma lungo da spiegare.

```
- (void)
updateZoom: (id)sender
{
    float        czl, rl ;
    czl = [ zoomSlider floatValue ] ;
    if ( czl >= 0 )
            rl = czl + 1;
    else    rl = 1 / (-czl+1) ;
    [cvrView setZoom: rl ];
    [ zoomValue setStringValue: displayZoomFactor(rl) ];
    [cvrView setNeedsDisplay: YES ];
}
```



Comincio da come è costruito lo slider. Questo restituisce numeri compresi tra -4 e +4, ma solo se corrispondono ad una delle 17 suddivisioni che ho impostato. In altre parole, restituisce un qualsiasi valore compreso tra -4 e +4 a passi di 0.5. Dico che valori negativi dello slider sono fattori di rimpicciolimento, mentre valori positivi sono ingrandimenti. Il valore centrale di Zero è il normale fattore di scala di uno ad uno. A questo punto dico che il valore 1 dello slider corrisponde ad un fattore di scala di 2, e così via, fino ad arrivare al valore 4 dello slider che ingrandisce di cinque volte la coverView. D'altra parte, un valore di -1 significa che la vista è ridotta alla metà, e via così fino al valore di -4 che rimpicciolisce la vista ad un quinto dell'originale. Il fattore di scale è quindi il valore dello slider più Uno se tale valore è negativo, mentre è dato dall'espressione

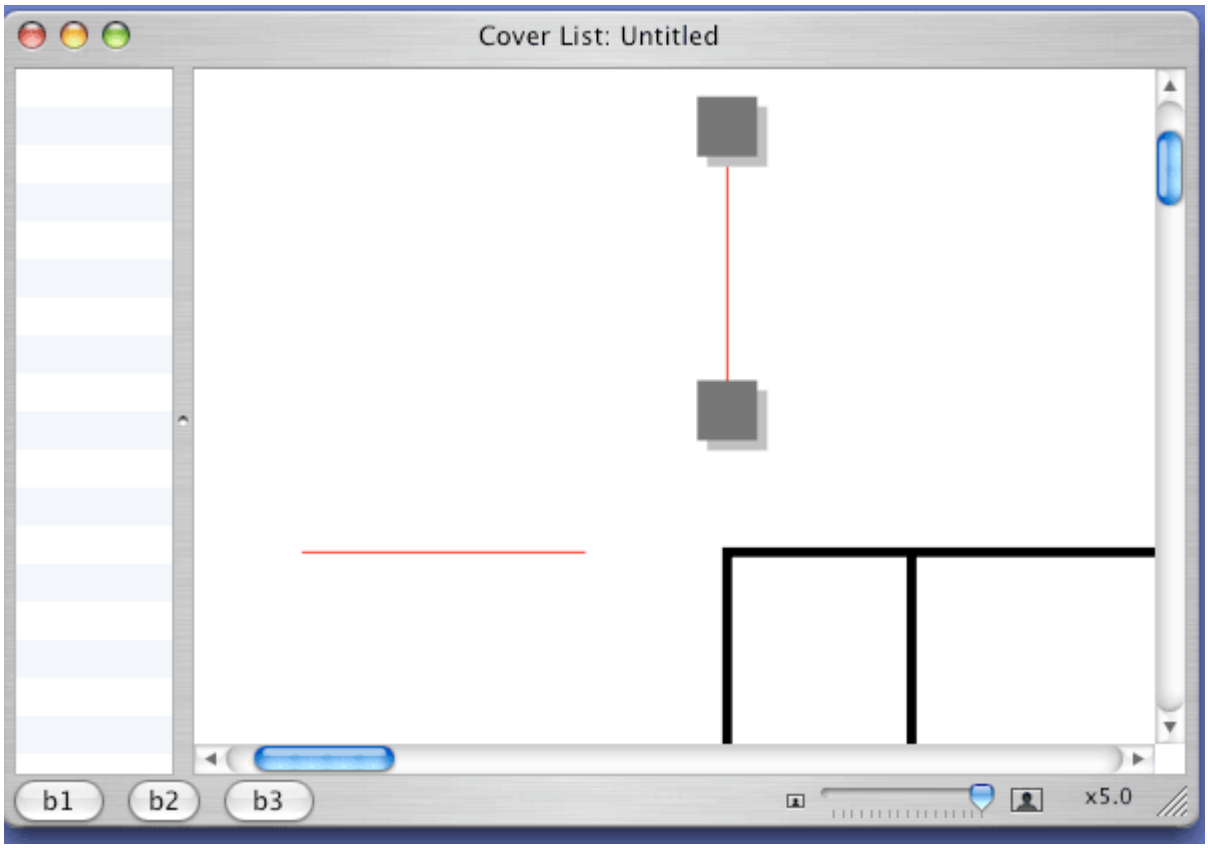
```
1 / (1 - [valore dello slider])
```

nel caso in cui il valore sia negativo (faccio una prova: se il valore dello slider è -2, il fattore di scala è 1/(1-(-2))=1/3).

Tutto questo pasticcio per impostare un fattore di zoom massimo di cinque (cinque volte più grande) e un fattore di riduzione pure pari a cinque (un quinto delle dimensioni originarie).

Tuttavia, il lavoro deve ancora essere fatto. È infatti compito del metodo `setZoom:`, che nella sua innocenza è di una potenza estrema.

```
- (void)
setZoom:(float)scaleFactor
{
    NSRect frame = [self frame];
    NSRect bounds = [self bounds];
    frame.size.width = bounds.size.width * scaleFactor;
    frame.size.height = bounds.size.height * scaleFactor;
    [self setFrameSize: frame.size];
    [self setBoundsSize: bounds.size];
}
```

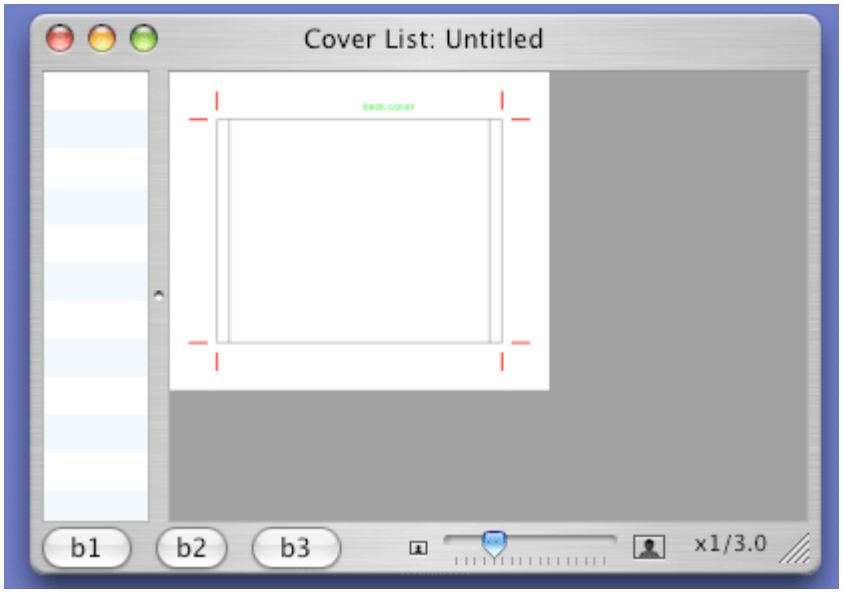


Ora, confesso che non ho proprio capito tutto per bene ciò che succede.

Provo a descriverlo, sperando nel farlo di capirci qualcosa di più. Comincio dai concetti di `frame` e di `bounds`. Ho già scritto che frame riporta il rettangolo della view nelle coordinate della sua superview. In particolare, le dimensioni orizzontali e verticali indicano quanto si sviluppa nel piano la view stessa. Il rettangolo bounds invece indica lo stesso rettangolo, ma visto all'interno delle coordinate della view. Lo guardo da un'altra parte. Il rettangolo bounds mi da le dimensioni della vista, che poi è piazzata

nella superview seguendo il rettangolo frame. Ora, normalmente, senza troppe giravolte all'interno della superview, le dimensioni `width` e `height` di bounds e frame sono le stesse.

Per fare lo zoom di una view, aumento o riduco le dimensioni della view stessa. Piglio allora la dimensione orizzontale e verticale e le moltiplico per il fattore di scala a cui voglio ridimensionare il tutto (terza e quarta riga del metodo precedente). Lascio inalterata la posizione della view, visto che è ancorata in alto a sinistra (e mi va bene così). Adesso impongo il nuovo rettangolo frame con la quinta linea del metodo.



Questa istruzione modifica anche il rettangolo bounds (potete verificarlo con il debugger, oppure facendo qualche stampa con la comoda `NSLog`), che l'istruzione successiva riporta alla dimensione standard. E questo è tutto. Come mai funziona? Non ho le idee ben chiare. Vado a leggere il manuale e trovo alcune delucidazioni all'interno della documentazione relativa a `setBoundsSize`. Il metodo impone la dimensione del rettangolo bounds, "modificando il sistema di coordinate relativamente al rettangolo frame".

Interpreto la cosa in questo modo: il rettangolo bounds dà le dimensioni effettive della vista. Tuttavia, questo rettangolo è rappresentato sullo schermo esattamente all'interno del rettangolo frame. Se quindi il rettangolo frame è più piccolo di bounds, ho una rappresentazione su scala ridotta, se frame è più grande di bounds, ho una rappresentazione ingrandita.

Le cose sono più chiare adesso? A me, no.

Però funziona. E tanto basta.

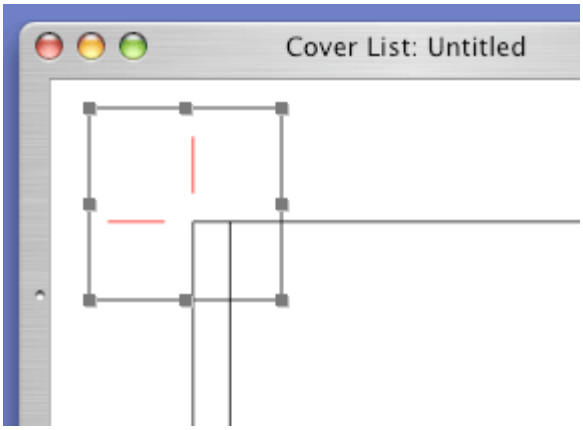
Capitolo 044 - Ridimensionamento e Selezione

In questo capitolo aggiungo la possibilità di ridimensionare gli elementi disegnati trascinando gli *handle*, ovvero i quadratini che individuano meglio un elemento selezionato. Già che ci sono, aggiungo la funzione di selezione di più elementi attraverso il mouse.

Faccio tutto da solo leggendo la documentazione.

Primo inserimento: 25 marzo 2004

NUOVI HANDLE



Ho già parlato degli handle come di quei quadratini che contornano e individuano un oggetto selezionato. Per prima cosa, e per meglio poi rintracciare un clic all'interno di uno di questi handle, riscrivo la parte che li riguarda. Semplicemente, alla costruzione di un oggetto, costruisco anche gli handle corrispondenti, e salvo i punti individuati in un vettore; come effetto collaterale, dovrei ridisegnare la vista più velocemente, dal momento che non devono essere calcolati ogni volta.

Inserisco quindi un vettore di otto `NSPoint` all'interno della struttura dati di un elemento grafico, completato dal numero effettivo di handle presenti.

```
    int             numOfHdl ;
    NSPoint         hdlList[8] ;
```

Il numero massimo è otto: sono i quattro spigoli ed i quattro punto di mezzo dei lati del rettangolo che circoscrive l'elemento. Nel caso di un segmento, i punti saranno solo due (i due estremi).

La classe `CCE_BasicForm` prevede un metodo di default adatto alla maggior parte dei casi:

```
- (void) buildHdlList
{
        // gli handle sono genericamente ai quattro estremi del
        // rettangolo limite e nei quattro punti mediani
        hdlList[0] = NSMakePoint( elemLimit.origin.x, elemLimit.origin.y) ;
          hdlList[1] = NSMakePoint( elemLimit.origin.x, elemLimit.origin.y+
elemLimit.size.height) ;
                        hdlList[2]   =   NSMakePoint(   elemLimit.origin.x   +
elemLimit.size.width, elemLimit.origin.y+ elemLimit.size.height) ;
                        hdlList[3]   =   NSMakePoint(   elemLimit.origin.x   +
elemLimit.size.width, elemLimit.origin.y) ;
        hdlList[4].x = (hdlList[0].x + hdlList[1].x ) * 0.5 ;
        hdlList[4].y = (hdlList[0].y + hdlList[1].y ) * 0.5 ;
        hdlList[5].x = (hdlList[2].x + hdlList[1].x ) * 0.5 ;
        hdlList[5].y = (hdlList[2].y + hdlList[1].y ) * 0.5 ;
        hdlList[6].x = (hdlList[2].x + hdlList[3].x ) * 0.5 ;
        hdlList[6].y = (hdlList[2].y + hdlList[3].y ) * 0.5 ;
        hdlList[7].x = (hdlList[0].x + hdlList[3].x ) * 0.5 ;
        hdlList[7].y = (hdlList[0].y + hdlList[3].y ) * 0.5 ;
}
```

Qui si considera il rettangolo: da questo sono presi i quattro vertici ed i quattro punti di mezzo dei lati. Semplici considerazioni sulle coordinate cartesiane danno ragione delle espressioni.

Questo metodo può essere sovrascritto da ogni classe discedente per migliorare le proprie esigenze. È il caso ad esempio di `CCE_Line`, dove diventa:

```
- (void) buildHdlList
{
        // gli handle sono genericamente ai quattro estremi del
        // rettangolo limite e nei quattro punti mediani
        hdlList[0] = startPoint ;
        hdlList[1] = endPoint ;
}
```

Qui le operazioni sono ancora più semplici. I due handle sono gli estremi del segmento.

Ora, bisogna fare in modo che questo metodo sia chiamato ogni volta che cambiano le dimensioni di un oggetto. Non ho trovato soluzioni migliori che invocare il metodo all'interno del metodo `setElemLimit`, ovvero all'interno del metodo accessor che permette di impostare il rettangolo che circoscrive l'elemento.

```
- (void)setElemLimit:(NSRect)newElemLimit {
        elemLimit = newElemLimit;
        [ self buildHdlList ];
}
```

Poiché ogni metodo di inizializzazione chiama prima o poi questo accessor, ecco che automaticamente sono costruiti tutti gli handle. Purtroppo rimane fuori l'inizializzazione della variabile `numOfHdl`, che va fatta caso per caso. In realtà, al momento, sono solo due i casi. Il primo nel metodo `init` di `CCE_BasicForm`, che assegna il valore di Otto (perché di default tutto sta dentro ad un rettangolo). Il secondo nel metodo `init` di `CCE_Line`, il quale, dopo aver invocato il metodo `init` di `super` (e quindi di `CCE_BasicForm`), cambia il valore di default a due.

Per disegnare gli handle, ho poi predisposto un nuovo metodo

```
- (void) drawHandles
{
    if ( cceIsSelected )
    {
        int i;
        // disegno i manici per la manipolazione
        for ( i = 0 ; i < numOfHdl ; i ++ )
            ccex_drawHandleAt( hdlList[i] );
    }
}
```

Questo metodo disegna gli handle solo se l'elemento è considerato selezionato. In tal caso, sfrutta la buona vecchia funzione `ccex_drawHandleAt` per disegnare effettivamente lo handle. Questo metodo deve essere invocato all'interno dei metodi `drawElement` di ogni singolo elemento, sostituendo la parte corrispondente presente nella precedente realizzazione. Ad esempio, per un rettangolo il metodo è:

```
- (void) drawElement: (NSRect) inRect
{
    if ( cceIsStroked )
    {
        // imposto il colore del contorno
        [ cceLineColor set ];
        // imposto lo spesosre del contorno
        [ theRect setLineWidth: [ self cceLineWidth ]];
        // disegno il contorno
        [ theRect stroke ];
    }
    // se il rettangolo e' pieno
    if ( cceIsFilled )
    {
        // imposto il colore del pieno
        [ cceFillColor set ];
        // riempi tutto
        [ theRect fill ];
    }
    [ self drawHandles ] ;
}
```

Fino a questo punto, nulla di nuovo; ho solo riorganizzato le cose.

SELEZIONARE UN HANDLE

Il vantaggio di tenere memorizzate le coordinate degli handle viene comodo quando si tratta di capire dove avviene un clic del mouse. Per prima cosa, costruisco un metodo della classe `CCE_BasicForm` in grado di capire se il clic del mouse è avvenuto sopra un handle:

```
- (short)
clicIntoHandle: (NSPoint) point
{
    int i;
    // disegno i manici per la manipolazione
    for ( i = 0 ; i < numOfHdl ; i ++ )
```

```
    {
        // se il punto designato e lo handle sono abbastanza vicini
        float a1 = hdlList[i].x - point.x ;
        float a2 = hdlList[i].y - point.y ;
        if ( a1 * a1 + a2 * a2 <= 4 )
            return ( i );
    }
    return ( -1 ) ;
}
```

Il metodo è piuttosto banalotto. Esamina ordinatamente gli handle, e verifica se i due punti (quello dello handle e quello passato come argomento, sarà il punto dove il mouse ha fatto clic) sono abbastanza vicini (per chi mastica di geometria, il metodo verifica che la distanza tra i due punti sia inferiore a due pixel). Il metodo restituisce l'indice dello handle cliccato, oppure `-1` per indicare che il clic è andato altrove.

Uso questo metodo per modificare l'esame del clic del mouse:

```
- (CCE_BasicForm *)
getClickedElem:(NSPoint)point
{
    int                i , numElem ;
    CCE_BasicForm * elem ;
    NSMutableArray        * listaelem ;
    // esamino tutti gli elemento
    listaelem = [ theElements elemArray ];
    numElem = [ listaelem count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ listaelem objectAtIndex: i ];
        // e controllo se il click gli appartiene
        if ( [ elem checkClickOnMe: point ])
        {
            // ho finito, esco dal ciclo
            break;
        }
        // controllo un eventuale click su handle
        if ( [ elem clicIntoHandle: point] != -1 )
            break ;
    }
    // potrei non avere trovato nulla
    if ( i == numElem )
        return ( nil );
    // se arrivo qui, ho trovato un elemento
    return ( elem );
}
```

In pratica, considero un clic sopra un elemento quando il mouse si trova proprio sopra questo elemento, oppure se il mouse è sopra uno degli handle (che, tecnicamente, sono anche un po' al di fuori del rettangolo che contiene l'elemento).

Ancora una volta, fino a qui non ci sono grosse novità. Comincia a muoversi qualcosa all'interno del metodo che tratta il movimento degli elementi. Questa volta, se il clic è sopra uno degli handle, piuttosto che muovere un elemento, lo ridimensiono.

```
- (void)
handleMoveElem: (NSEvent *)theEvent
    inView:        (CoverView *)view
```

```
{
    // recupero la coordinata locale
        NSPoint point1 = [view convertPoint:[theEvent locationInWindow]
fromView:nil];
    NSPoint point2 ;
    short hdlIdx = [ self clicIntoHandle: point1 ] ;
    // potrebbe aver fatto clic su uno degli hangle
    if ( hdlIdx == -1 )
    {
        // no, clic in zona generica
        // ciclo piu' o meno lungo di attesa eventi
        do {
            // recupero il prossimo evento, che sia un movimento del mouse
            // oppure un mouseup (che finisce il tutto)
                                        theEvent = [[view window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
            // recupero la coordinata locale
                  point2 = [view convertPoint:[theEvent locationInWindow]
fromView:nil];
            // aggiusto l'aspetto dell'oggetto
            [ self shiftMeFrom: point1 to: point2 ];
            // il nuovo punto di partenza
            point1 = point2 ;
            // forzo un ridisegno della finestra
            [ view setNeedsDisplay: YES ];
        }
        while ( [theEvent type] != NSLeftMouseUp) ;
    }
    else
    {
        // clic su uno degli angle, ridimensiono
        [ self handleResize: theEvent inView: view movingHandle: hdlIdx ];
    }
}
```

Si tratta del metodo già visto in precedenza, al quale è stato aggiunto a contorno un controllo sulla locazione del clic. Se il clic è su uno degli handle, allora il controllo passa al metodo `handleResize:...`.

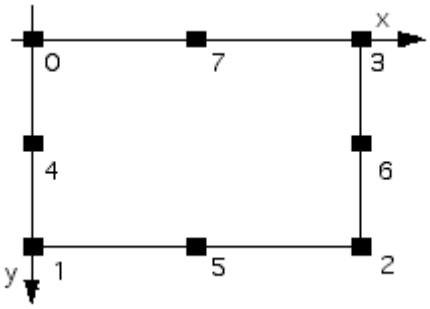
E qui le cose cominciano a complicarsi.

Come sono abituato a ridimensionare un elemento (diciamo un rettangolo, per fissare le idee) attraverso gli handle? Se piglio uno degli handle sugli spigoli, voglio modificare l'intero rettangolo. Allora tengo fisso lo spigolo opposto (rispetto al centro del rettangolo) e muovo a piacimento lo spigolo sul quale ho fatto clic. Quando rilascio il mouse, ho un nuovo rettangolo, determinato dal punto fisso e dal punto di rilascio.

Se invece piglio uno degli handle in mezzo ai lati, voglio modificare solo una dimensione del rettangolo. Se seleziono il punto di mezzo di uno dei lati verticali, ridimensiono la larghezza del rettangolo, ma l'altezza del rettangolo deve rimanere fissa. Viceversa, selezionando uno degli handle nei lati orizzontali, si modifica l'altezza, ma non la larghezza.

Riassumo queste due operazioni, che sembrano piuttosto diverse tra loro, in una cornice unificante. Per ogni ridimensionamento, individuo un punto fisso, come un vertice che rimane comunque fermo nel processo di manipolazione. Individuo poi un vertice mobile, come il vertice che poi, alla fine delle operazioni, concorre col punto fermo alla determinazione del rettangolo. Le coordinate del punto mobile possono

cambiare contemporaneamente per le ascisse e per le ordinate (è il primo caso, in cui lo handle in movimento è uno dei vertici), oppure solo una tra ascissa o ordinata (il secondo caso, in cui lo handle selezionato è uno dei punti di mezzo dei lati). Identifico ogni handle con l'indice che questo handle ha all'interno del vettore `hdlList`.



Bene, basandomi su quanto detto, e guardando con attenzione la figura, costruisco tre vettori; il primo indica quale handle rimane fisso nella manipolazione, il secondo quale si muove, ed il terzo vettore indica con un codice quale coordinata si muove. Supponiamo che il clic avvenga sullo handle numero 3 (il punto in alto a destra). Allora, il punto fisso è il vertice numero 1, lo handle mobile è proprio quello numero 3, e nel ridimensionamento cambiano sia l'ascissa sia l'ordinata. Questo è il motivo perchè il quarto elemento di ogni vettore (quindi, quello di indice 3) è rispettivamente 1, 3 e 3 (codice che indica ascissa e ordinata).

```
    static short fixedHandle[8] = { 2, 3, 0, 1, 2, 0, 0, 2 };
    static short movingHandle[8] = { 0, 1, 2, 3, 0, 2, 2, 0 };
    static short movingPoints[8] = { 3, 3, 3, 3, 2, 1, 2, 1 };
```

Le cose si complicano quando il clic è sopra uno degli handle di mezzo. Supponiamo clic sullo handle 7. Posso indifferentemente scegliere il vertice 1 o 2: entrambi non cambieranno coordinate. Come handle mobile, posso scegliere ancora i vertici 0 o 3; in ogni caso, entrambi variano la coordinata y (che ho indicato col codice 1). Quindi, l'ottavo elemento dei vettori (indice numero 7) sono rispettivamente 2, 0 e 1.

Capito il meccanismo, si riempiono facile le altre posizioni del vettore.

In effetti, tutto questo discorso serve proprio ad individuare tre variabili:

```
    NSPoint fixedPoint = hdlList[ fixedHandle[ hdlIdx ] ];
    NSPoint movingPoint = hdlList[ movingHandle[hdlIdx] ];
    short movingCoord = movingPoints[ hdlIdx ];
```

dove appunto conservare le coordinate del punto fisso, le coordinate del punto variabile ed il codice che dice quali coordinate devono variare. A questo punto, costruisco il solito loop che aspetta il rilascio del mouse:

```
    do {          // recupero il prossimo evento, che sia un movimento del
mouse          // oppure un mouseup (che finisce il tutto)          theEvent
=    [[view    window]    nextEventMatchingMask:(NSLeftMouseDraggedMask    |
NSLeftMouseUpMask)];          // recupero la coordinata locale          point
= [view convertPoint:[theEvent locationInWindow] fromView:nil];          ...
          // faccio qualcosa col punto           ...           // forzo un
```

```
ridisegno della finestra            [ view setNeedsDisplay: YES ];      }
    while ( [theEvent type] != NSLeftMouseUp) ;
```

Cosa fare del nuovo punto determinato dall'utente col suo girovagare? Devo costruire un nuovo rettangolo: per farlo, devo determinare il secondo punto. E qui le operazioni dipendono da quale coordinata è in movimento. Se varia la coordinata x, il nuovo punto è dato dalla coordinata x del punto nuovo, ma la coordinata y va pescata dal punto `movingPoint`. Se varia la coordinata y, le due coordinate si scambiano di ruolo. Infine, se entrambe le coordinate devono variare, utilizzo entrambe le nuove coordinate (ed in questo caso il punto `movingPoint` serve a nulla...). Ecco quindi il metodo nella sua completezza.

```
- (BOOL)
handleResize: (NSEvent *)theEvent
    inView:        (CoverView *)view
    movingHandle: (short) hdlIdx
{
    // indice dello handle che tengo fisso
    static short fixedHandle[8] = { 2, 3, 0, 1, 2, 0, 0, 2 };
    // indice dello handle che devo muovere (opposto)
    static short movingHandle[8] = { 2, 3, 0, 1, 0, 2, 2, 0 };
    // identificatore delle coordinate che posso muovere:
    // con 3, x e y; con 2, solo x, con 1 solo y
    static short movingPoints[8] = { 3, 3, 3, 3, 2, 1, 2, 1 };

    short movingCoord = movingPoints[ hdlIdx ];
    NSPoint fixedPoint = hdlList[ fixedHandle[ hdlIdx ] ];
    NSPoint movingPoint = hdlList[ movingHandle[hdlIdx] ];
    NSPoint point ;
    // continuo a ciclare in attesa di un mouseup
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
                                        theEvent    =    [[view    window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
        // recupero la coordinata locale
                point  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
        // aggiusto l'aspetto dell'oggetto
        switch ( movingCoord ) {
        case 1 :
                [ self buildMeWithStartPt: fixedPoint andPt: NSMakePoint(
movingPoint.x, point.y) ] ;
            break ;
        case 2 :
                [ self buildMeWithStartPt: fixedPoint andPt: NSMakePoint(
point.x, movingPoint.y) ] ;
            break ;
        case 3 :    // muovo entrambe le coordinate
            [ self buildMeWithStartPt: fixedPoint andPt: point ] ;
            break ;
        }
        // forzo un ridisegno della finestra
        [ view setNeedsDisplay: YES ];
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
    // arrivato qui, ho finito il tutto
    return YES;
}
```

Questo metodo va bene per tutti gli elementi univocamente determinati attraverso il rettangolo che li delimita. Quindi funziona sia per `CCE_Rect` che per `CCE_Circle` (e, per il momento, anche per `CCE_Text` senza colpo ferire. Addirittura, stabilisco che questo metodo è la realizzazione di default; gli oggetti che si comportano in maniera diversa, devono riscriverlo.

È questo, ovviamente, il caso di un segmento. Il meccanismo è più semplice (uno dei due handle è sempre fisso, l'altro si muove in entrambe le direzioni), per cui il codice dovrebbe essere autoesplicativo.

```
- (BOOL)
handleResize: (NSEvent *)theEvent
    inView:         (CoverView *)view
    movingHandle: (short) hdlIdx
{
    NSPoint fixedPoint = ( hdlIdx == 0) ? hdlList[ 1 ] : hdlList[ 0 ];
    NSPoint point ;
    // continuo a ciclare in attesa di un mouseup
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
                                        theEvent     =     [[view     window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
        // recupero la coordinata locale
                  point   =   [view   convertPoint:[theEvent   locationInWindow]
fromView:nil];
        // aggiusto l'aspetto dell'oggetto
        [ self buildMeWithStartPt: fixedPoint andPt: point ] ;
        // forzo un ridisegno della finestra
        [ view setNeedsDisplay: YES ];
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
    // arrivato qui, ho finito il tutto
    return YES;
}
```

A questo punto, ho messo in piedi un meccanismo per modificare le dimensioni di ogni oggetto presente all'interno di una coverView.

SELEZIONI MULTIPLE

Colto dalla foga, ho anche realizzato un meccanismo per effettuare selezioni multiple (non che c'entri molto col resto, ma insomma...). Ho utilizzato lo stile classico: un utente col mouse fa clic e poi, mantenendolo premuto, racchiude con un rettangolo tutti gli elementi che intende selezionare. La cosa si presenta facile, dal momento che riciclo molti concetti già utilizzati. Intanto, devo visualizzare un rettangolo effimero all'interno della coverView. Opero come col rettangolo che ho utilizzato per dare riscontro all'utente quando costruisce un elmento rettangolare: definisco una nuova variabile d'istanza della classe `CoverView`, normalmente nullo, ma che diventa effettivo quando è in corso l'opera di selezione. Ho chiamato la variabile

```
    CCE_Rect            * curSelectRect ;
```

ed ho definito i soliti metodi accessor. Ho poi modificato il metodo

`handleMouseClick` perché nella parte in cui il clic è in terra di nessuno invochi invece un metodo apposta per cominciare con la selezione multipla.

```
- (void)
handleMouseClick: (NSEvent *)theEvent
{
    ...
    // se c'e' un elemento sotto il mouse
    if ( curElem )
    {
        ...
    }
    else
    {
        NSRect selRect ;
        // comincio la selezione multipla
        selRect = [ self handleRectSelect: theEvent ];
        [ theElements selectMeIfInRect: selRect ];
    }
    ...
}
```

Le due operazioni in sequenza sono: la determinazione (ed il disegno, e la gestione dell'interattività) del rettangolo di selezione, e poi la selezione di tutti gli elementi che si trovano all'interno del rettangolo individuato.

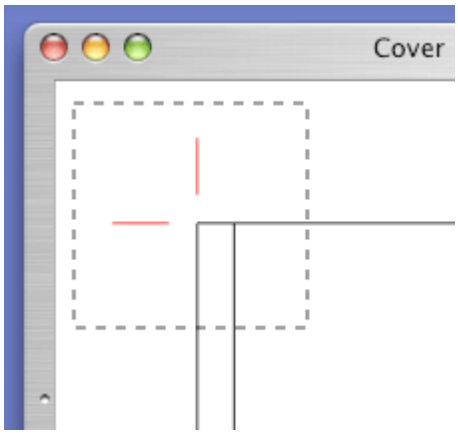
Per determinare il rettangolo, il metodo è il seguente.

```
- (NSRect)
handleRectSelect: (NSEvent *)theEvent
{
    NSRect        selRect = NSMakeRect(1, 1, 0, 0);
    // costruisco l'elemento grafico
    CCE_Rect * curElem = [[ [CCE_Rect alloc ] init] autorelease ] ;
    // metto a posto un di attributi base dell'elemento
    [ curElem setCceLineWidth: 0.0F ];
    [ curElem setCceLineColor: [NSColor blackColor] ];
    [ curElem setCceFillColor: [NSColor lightGrayColor] ];
    [ curElem setCceIsFilled: NO ];
    [ curElem setCceIsStroked: YES ];
    // deseleziono tutto
    [ theElements setCceIsSelected: NO ] ;
    // dico che l'elemento che sto disegnando e' selezionato
    [ curElem setCceIsSelected: NO ];
    [ self setCurSelectRect: curElem ] ;
    // gestisco la costruzione col mouse
    [curElem handleCreation: theEvent inView: self ] ;
    selRect = [ curElem elemLimit] ;
    [ self setCurSelectRect: nil ] ;
    return ( selRect );
}
```

Qui si procede come per la normale costruzione di un rettangolo. Si costruisce l'oggetto, se ne predispongono alcuni parametri, si assegna il rettangolo alla variabile d'istanza creata apposta, poi si gestisce il mouse con metodo `handleCreation:` (che è sempre quello utilizzato per gli elementi normali). Alla fine, si rimette a `nil` la variabile, e si restituisce il rettangolo individuato.

C'è da modificare il metodo di disegno della coverView

```
- (void)drawRect:(NSRect)rect
{
    // sfondo bianco
    [[ NSColor whiteColor] set ];
    NSRectFill( rect );
    // poi dico di disegnarsi a tutti gli elementi
    [ [self theElements] drawElement: rect ];
    // e poi all'elemento in corso
    if ( curCreatingElem )
    {
        [ curCreatingElem drawElement: rect ] ;
    }
    if ( curSelectRect )
    {
        NSBezierPath    * curPath = [ curSelectRect theRect ];
        float            strokePattern[2] ={ 5, 5 } ;
        [ curPath setLineDash: strokePattern count: 2 phase: 0.0];
        [ curSelectRect setTheRect: curPath ];
        [ curSelectRect drawElement: rect ] ;
    }
}
```



Ho aggiunto, oltre al normale disegno del rettangolo, una migliore caratterizzazione del rettangolo, imponendo il bordo tratteggiato. Consultando la documentazione della classe `NSBezierPath` ho notato appunto il metodo `setLineDash:...` che permette di impostare il tratteggio attraverso un vettore di numeri; questi numeri determinano la parte piena e la parte vuota del tratteggio. Nel caso, c'è un tratto pieno di cinque pixel seguito da un uguale tratto vuoto.

Infine, non rimane che dettagliare il metodo `selectMeIfInRect` , invero piuttosto semplice, ma che ha bisogno di tre diverse realizzazioni, quella di default (per gli elementi rettangolari), per la classe `CCE_Line` e per la classe `CCE_ElemGroup`. Nel primo caso, si usa la banale funzione `NSContainsRect` per vedere se il rettangolo dell'elemento è incorporato nel rettangolo di selezione; nel secondo caso si verifica che entrambi i punti estremi risiedono nel rettangolo di selezione; nel terzo caso, infine, occorre chiedere separatamete a tutti gli elementi del gruppo se sono o meno includi all'interno del rettangolo di selezione.

```
- (void) selectMeIfInRect: (NSRect) selRect
{
```

```
    NSRect mioRect = [self elemLimit] ;
    if (NSContainsRect( selRect, mioRect ) )
            [ self setCceIsSelected: YES ];
    else    [ self setCceIsSelected: NO ];
}
...
- (void)
selectMeIfInRect: (NSRect) selRect
{
      if ( NSPointInRect(startPoint, selRect) && NSPointInRect(endPoint,
selRect) )
            [ self setCceIsSelected: YES ];
    else    [ self setCceIsSelected: NO ];
}
...
- (void) selectMeIfInRect: (NSRect) selRect
{
    int        i , numElem ;
    CCE_BasicForm * elem ;
    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem selectMeIfInRect: selRect ] ;
    }
}
```

Capitolo 045 - Shift Me Up!

In questo capitolo tratto alcuni argomenti legati tra loro dalla tastiera, ed in particolar modo dal tasto maiuscole. Nel fare ciò scopro qualcosa diinteressante sul sistema operativo.

Da Sketch, ma poi documentazione Apple.

Primo inserimento: 5 aprile 2004

SELEZIONI MULTIPLE

Attraverso il mouse, ho imparato come selezionare un elemento (facendoci sopra clic) o più elementi contemporaneamente (disegnando un rettangolo, e selezionando gli elementi compresi nel suo interno). Tuttavia, un metodo classico di selezione multipla di elementi consiste nell'utilizzare il tasto delle maiuscole. In pratica, esiste il concetto di selezione corrente, come insieme degli elementi selezionati. Normalmente vuota (nessun elemento selezionato), facendo clic sopra un elemento non ancora selezionato col tasto maiuscole premuto, l'elemento si aggiunge alla selezione; se l'elemento è già selezionato, il tipico risultato è di toglierlo dall'insieme. Provo a riassumere il tutto in quattro casi possibili:

Caso 1:
elemento selezionato, tasto maiuscole premuto: l'elemento esce dall'insieme ;
Caso 2:
elemento selezionato, tasto maiuscole non premuto (situazione standard): probabilmente l'utente intende muovere l'elemento;
Caso 3:
elemento non selezionato, tasto maiuscole premuto: l'elemento è aggiunto all'insieme;
Caso 4:
elemento non selezionato, tasto maiuscole non premuto (situazione standard): ho un nuovo insieme, costituito da questo solo elemento.

Ora, ero partito veloce definendo un gruppo (temporaneo) `CCE_ElemGroup` dove conservare tutti gli elementi selezionati, ma poi mi sono ricreduto. In effetti, non occorre definire esplicitare l'insieme: basta considerare implicitamente l'insieme come composto da tutti quegli elementi che hanno la variabile d'istanza `cceIsSelected` a Vero. Quindi, le operazioni di gestione si limitano a cambiare lo stato di questa variabile nei vari casi:

Caso 1:
`cceIsSelected` passa da Vero a Falso
Caso 2:
`cceIsSelected` rimane Vero
Caso 3:
`cceIsSelected` passa da Falso a Vero
Caso 4:

`cceIsSelected` passa da Falso a Vero; tuttavia, per tutti gli altri elementi la variabile deve diventare Falso.

Sembra una cosa facile, ma come tutte le cose facili nasconde un problema enorme (si fa per dire). Il meccanismo di spostamento non funziona più. Se eseguo una selezione multipla, mi aspetto che l'operazione di spostamento lavori su tutti gli elementi selezionati. Tuttavia, per come ho finora predisposto i metodi, il meccanismo di spostamento è *ad personam*: incarico l'elemento stesso di eseguire lo spostamento (gestendo gli eventi successivi legati al movimento del mouse) invocando il metodo della classe `CCE_BasicForm`

```
- (void) handleMoveElem: (NSEvent *)theEvent inView: (CoverView *)view ;
```

C'è di peggio: all'interno di questo metodo trattavo anche il caso in cui il clic avviene sopra uno degli handle. Allora interpretavo l'intenzione dell'utente non più come uno spostamento, ma come un ridimensionamento. Nel caso di selezione multipla, le due operazioni si pestano i piedi a vicenda: se, con una selezione multipla, l'utente fa clic sopra uno degli handle, come interpreto la cosa? Mi sono risposto nel seguente modo (è solo una delle possibili soluzioni): se l'utente fa clic sopra un handle, intende ridimensionare solo l'elemento cui lo handle appartiene; esegue il ridimensionamento, e l'insieme degli elementi selezionati non subisce modifiche. Se l'utente fa clic all'interno di uno qualsiasi degli elementi selezionati, intende spostare tutti gli elementi selezionati.

Per mettere in pratica quanto detto, ho riscritto quasi completamente il seguente metodo:

```
- (void)
handleMouseClick: (NSEvent *)theEvent
{
    NSPoint         whereClick;
    CCE_BasicForm * curElem = nil;
      BOOL                  shiftKeyPress =  (([theEvent  modifierFlags]  &
NSShiftKeyMask) ? YES : NO);
    // vado a vedere dove e' avvenuto il click
          whereClick   =   [self   convertPoint:[theEvent   locationInWindow]
fromView:nil];
    // niente niente l'utente vuole selezionare un elemento
    curElem = [self getClickedElem:whereClick];
    // potrebbe aver fatto clic su uno degli hangle
    // se c'e' un elemento sotto il mouse
    if ( curElem )
    {
        short clicInHandle = [ curElem clicIntoHandle: whereClick ] ;
        // abbiamo qualcosa sotto il mouse
        // vedo se e' gia' selezionato
        if ( [ curElem cceIsSelected ] )
        {
         if ( shiftKeyPress )    // caso 1
         {
                // lo tolgo dagli elementi selezionati
                [ curElem setCceIsSelected: NO ];
           }
           else                    // caso 2
           {
               // provo a vedere se e' il caso di muoverlo
```

```
                if ( clicInHandle == -1 )
                    [ self handleMoveAll: theEvent ];
                       else [ curElem handleResize: theEvent inView: self
movingHandle: clicInHandle ];
            }
        }
        else
        {
            if ( ! shiftKeyPress )        // caso 4
            {
                // deseleziono tutto
                [ theElements setCceIsSelected: NO ] ;
            }
            // da qui in poi, caso 3 (ed anche caso 4)
            // seleziono questo elemento
            [ curElem setCceIsSelected: YES ];
            // ridisegno tutto che la vista e' cambiata...
            [ self setNeedsDisplay: YES ];
            // provo a vedere se e' il caso di muoverlo
            if ( clicInHandle == -1 )
                [ self handleMoveAll: theEvent ];
                     else [ curElem handleResize: theEvent inView: self
movingHandle: clicInHandle ];
        }
    }
    else
    {
        NSRect selRect ;
        // comincio la selezione multipla
        selRect = [ self handleRectSelect: theEvent ];
                [ theElements selectMeIfInRect: selRect addingToSelez:
shiftKeyPress ];
    }
    // ridisegno tutto, in ogni caso
    [ self setNeedsDisplay: YES ];
}
```

Per prima cosa, cerco di capire se l'utente ha premuto il tasto maiuscole, guardando una delle tante informazioni accessorie fornite con l'evento (appunto, se il tasto shift -maiuscole- è premuto). Metto da parte questa informazione, che mi serve poi.

Ho spostato qui alcune istruzioni prima presenti nel metodo `handleMoveElem:...`, per capire se il clic dell'utente è sopra un elemento e se è addirittura sopra uno degli handle.

Finalmente, comincio a valutare i quattro casi, annidando un paio di istruzioni `if`. Nel caso 2, sostituisco la precedente chiamata al metodo `handleMoveElem` con una ulteriore istruzione `if`, che si chiede se il clic è avvenuto sopra un handle oppure no. Nel caso di clic sopra un handle, ritorno ad utilizzare il metodo `handleResize...`, ed è finita lì. Se invece si intende spostare gli elementi, ho il nuovo messaggio `handleMoveAll:...` che gestisce per l'intera view il movimento del mouse e di tutti gli elementi selezionati (ne parlo poi). Il caso 3 ed il caso 4 sono molto simili; il caso 4, prima di tutto, svuota la selezione corrente col tipico meccanismo di porre a Falso `cceIsSelected` dell'insieme di tutti gli elementi (che poi a catena si riverbera su tutti gli altri). In entrambi i casi, seleziono l'elemento considerato, e continuo a pormi la solita questione se il cli è sopra un handle o meno.

AGGIUNTE MULTIPLE

C'è da fare una ulteriore modifica, perché adesso anche la selezione muovendo il mouse deve tenere conto del tasto maiuscole. Il metodo `handleRectSelect` ha subito una piccola modifica.

```
- (NSRect)
handleRectSelect: (NSEvent *)theEvent
{
    NSRect        selRect = NSMakeRect(1, 1, 0, 0);
    // costruisco l'elemento grafico
    CCE_Rect * curElem = [[ [CCE_Rect alloc ] init] autorelease ] ;
    BOOL shiftKeyPress = (([theEvent modifierFlags] & NSShiftKeyMask) ? YES
: NO);
    // metto a posto un di attributi base dell'elemento
    [ curElem setCceLineWidth: 0.0F ];
    [ curElem setCceLineColor: [NSColor blackColor] ];
    [ curElem setCceFillColor: [NSColor lightGrayColor] ];
    [ curElem setCceIsFilled: NO ];
    [ curElem setCceIsStroked: YES ];
    if ( ! shiftKeyPress )
    {
        // deseleziono tutto
        [ theElements setCceIsSelected: NO ] ;
    }
    // dico che l'elemento che sto disegnando e' selezionato
    [ curElem setCceIsSelected: NO ];
    [ self setCurSelectRect: curElem ] ;
    // gestisco la costruzione col mouse
    [curElem handleCreation: theEvent inView: self ] ;
    selRect = [ curElem elemLimit] ;
    [ self setCurSelectRect: nil ] ;
    return ( selRect );
}
```

Molto semplicemente, se il tasto maiuscole è premuto, non devo cancellare la selezione corrente.

Più complicata invece la gestione di questa selezione rettangolare, che mi ha portato a modificare sostanzialmente il metodo utilizzato da questa istruzione:

```
                [ theElements selectMeIfInRect: selRect addingToSelez:
shiftKeyPress ];
```

Ho aggiunto un parametro, che dice se aggiungere o meno gli elementi compresi nel rettangolo `selRect` alla selezione corrente. Anche qui, ci sono un po' di considerazioni da fare, con quattro casi possibili:

Caso a:
: l'elemento non è compreso nel rettangolo, e il tasto non è premuto: `cceIsSelected` è Falso ;

Caso b:
: l'elemento non è compreso nel rettangolo, e il tasto è premuto: `cceIsSelected` va lasciato invariato;

Caso c:

l'elemento è compreso nel rettangolo, e il tasto non è premuto: `cceIsSelected` è Vero ;

Caso d:

l'elemento è compreso nel rettangolo, e il tasto è premuto: `cceIsSelected` è Vero.

La versione standard del metodo (classe `CCE_BasicForm`) è la seguente.

```
- (void)
selectMeIfInRect: (NSRect) selRect addingToSelez: (BOOL) add2sel
{
    NSRect mioRect = [self elemLimit] ;
    BOOL    inrect = NSContainsRect( selRect, mioRect ) ;
    // se non devo aggiungere alla selezione
    if ( ! add2sel )    // casi a e c
    {
        // vince il criterio del rettangolo
        [ self setCceIsSelected: inrect ];
    }
    else
    {
        // qui e' da aggiungere alla selezione
        // se sono dentro il rettangolo
        if ( inrect )    // caso d
            // per certo lo seleziono
            [ self setCceIsSelected: YES ];
        // altrimenti lascio la selezione precedente, caso b
    }
}
```

Le classi `CCE_Line` e `CCE_ElemGroup` possiedono versioni specializzate (la linea utilizza un criterio differente, mentre il gruppo itera il metodo sugli elementi appartenenti).

SPOSTAMENTI DI GRUPPO

Torno adesso sul metodo che avevo lasciato indietro, quello necessario per effettuare movimenti di tutti gli elementi selezionati. Non è altro che una versione riveduta e corretta del metodo `handleMoveElem::`

```
- (void)
handleMoveAll: (NSEvent *)theEvent
{
    // recupero la coordinata locale
        NSPoint  point1  =  [self  convertPoint:[theEvent  locationInWindow]
fromView:nil];
    NSPoint point2 ;

    // ciclo piu' o meno lungo di attesa eventi
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
                                    theEvent     =     [[self     window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
        // recupero la coordinata locale
                point2  =  [self  convertPoint:[theEvent  locationInWindow]
fromView:nil];
```

```
        // aggiusto l'aspetto dell'oggetto
                                        [     theElements     moveMeIfSelectBy:
NSMakePoint(point2.x-point1.x,point2.y-point1.y)];
        // il nuovo punto di partenza
        point1 = point2 ;
        // forzo un ridisegno della finestra
        [ self setNeedsDisplay: YES ];
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
}
```

La parte interessante è che, una volta stabilita l'entità dello spostamento, mando a tutti gli elmenti presenti nella view (all'elemento `theElements`, e da qui a cascata su tutti gli altri, sempre col solito meccanismo) il messaggio `moveMeIfSelectBy:`, di facile realizzazione (è sostanzialmente il vecchio `shiftMeFrom:to:`):

```
- (void)
moveMeIfSelectBy: (NSPoint) distance
{
    if ( cceIsSelected )
    {
                [  self  setElemLimit:  NSOffsetRect(elemLimit,  distance.x,
distance.y)];
               [  self  buildMeWithStartPt:  NSMakePoint(elemLimit.origin.x,
elemLimit.origin.y)
                                    andPt:  NSMakePoint(  elemLimit.origin.x  +
elemLimit.size.width,
                                                          elemLimit.origin.y  +
elemLimit.size.height) ];
    }
}
```

L'unica modifica è che l'operazione è svolta solamente se l'elemento è selezionato. Ancora una volta, le classi `CCE_Line` e `CCE_ElemGroup` sono dotate di un metodo proprio, per trattare le differenze col caso standard.

CERCHI QUADRATI E LINEE DIRITTE

Un altro uso classico del tasto shift si ha nel momento in cui si costruisce un elemento, ad esempio un rettangolo; tenendo premuto il tasto maiuscole, molti programmi di disegno forzano il rettangolo ad essere un quadrato, come pure il tracciamento di una ellisse diventa quello di un cerchio. Voglio fare la stessa cosa, con in più la possibilità di forzare il tracciamento di segmenti orizzontali, verticali e inclinati a 45 gradi quando il solito tasto maiuscole è tenuto premuto.

Per fare questo, devo modificare il metodo `handleCreation`. Comincio dalla versione per quadrati e cerchi, che mi pare più comprensibile. L'idea è di inserire una correzione sul secondo punto (quello in movimento) per far si che questo punto sia l'altro vertice di un quadrato.

Allora, per prima cosa, calcolo le due distanze `delta1` e `delta2`, orizzontale e verticale, tra il punto fisso `point1` ed il punto mobile `point`. Tra le due, scelgo quella più piccola: l'effetto è di *tirare* il quadrato da fuori. Devo tenere conto del fatto che *più piccola* significa in valore assoluto, e devo anche tenere conto della posizione relativa del

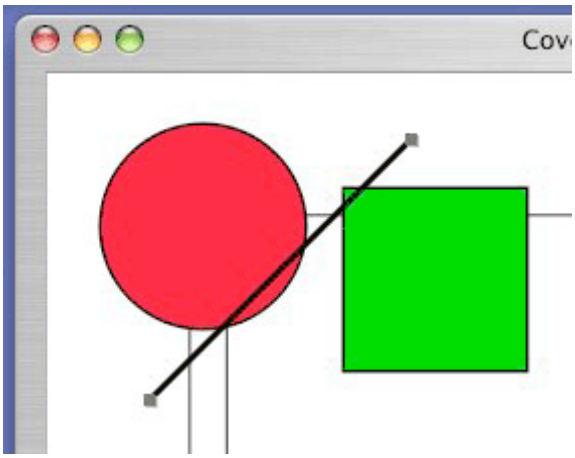
secondo punto rispetto al primo. Per questo tengo conto della posizione relativa con due variabili `dirx` e `diry` che dicono, col valore positivo, che il punto `point` sta a destra (sopra) il punto `point1`, e col valore negativo che sta a sinistra (o sotto). Stabilito questo, forzo l'una o l'altra coordinata del punto mobile in modo da dare luogo ad un quadrato. È come al solito più difficile da spiegare che da vedere.

```
- (BOOL)
handleCreation: (NSEvent *)theEvent
    inView:        (CoverView *)view
{
    // recupero la coordinata locale
        NSPoint  point1  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
    NSPoint point ;
    BOOL shiftKeyPress ;
    // inizializzo l'aspetto dell'oggetto (praticamente vuoto)
    [ self buildMeWithStartPt: point1 andPt: point1 ] ;
    // continuo a ciclare in attesa di un mouseup
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
                                        theEvent     =     [[view     window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
         shiftKeyPress = (([theEvent modifierFlags] & NSShiftKeyMask) ? YES
: NO);
        // recupero la coordinata locale
                  point  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
        if ( shiftKeyPress )
        {
            // costringo il rettangolo ad essere quadrato
            float delta1 = ( point.x - point1.x ) ;
            float delta2 = ( point.y - point1.y ) ;
            float    dirx = 1 ;
            float    diry = 1 ;

            if ( delta1 < 0 )
            {
                delta1 = - delta1 ; dirx = -1 ;
            }
            if ( delta2 < 0 )
            {
                delta2 = - delta2 ; diry = -1 ;
            }
            // piglio la dimensione piu' corta
            if ( delta1 < delta2 )
            {
                point.y = point1.y + diry * delta1 ;
            }
            else
            {
                point.x = point1.x + dirx * delta2 ;
            }
        }
        // aggiusto l'aspetto dell'oggetto
        [ self buildMeWithStartPt: point1 andPt: point ] ;
        // forzo un ridisegno della finestra
        [ view setNeedsDisplay: YES ];
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
    // arrivato qui, ho finito il tutto
    return YES;
```

```
}
```

Per quanto riguarda la classe `CCE_Line`, mi complico la cosa perché voglio tracciare linee orizzontali, verticali e a 45 gradi. Devo capire bene dove si posiziona il punto mobile rispetto al punto fisso. Ecco il frammento di codice eseguito a tasto maiuscole premuto:

```
            // costringo il rettangolo ad essere quadrato
            float delta1 = ( point.x - point1.x ) ;
            float delta2 = ( point.y - point1.y ) ;
            float    dirx = 1 ;
            float    diry = 1 ;

            if ( delta1 < 0 )
            {
                delta1 = - delta1 ; dirx = -1 ;
            }
            if ( delta2 < 0 )
            {
                delta2 = - delta2 ; diry = -1 ;
            }
            // piglio la dimensione piu' corta
            if ( delta1 < delta2*0.2 )
            {
                point.x = point1.x ;
            }
            else if ( delta2 < delta1*0.2 )
            {
                point.y = point1.y ;
            }
            else if ( delta1 < delta2 )
            {
                point.y = point1.y + diry * delta1 ;
            }
            else
            {
                point.x = point1.x + dirx * delta2 ;
            }
```



Dopo aver calcolato, come nel caso precedente, la differenza tra le ascisse e le ordinate, ed aver predisposto la direzione di movimento, mi chiedo se la distanza in ascissa è molto più piccola della distanza in ordinata. In tal caso, presuppongo di dover tracciare una riga in verticale, e quindi impongo al punto mobile la stessa ascissa del punto fisso. Se invece vale il caso opposto, la differenza in verticale è molto minore della differenza

in orizzontale, presumo di dover tracciare una linea orizzontale, ed impongo uguali ordinate. In tutti gli altri casi, ricado nella situazione precedente, e calcolo la nuova ascissa o ordinata nel modo solito.

UN ERRORE ED UNA CATTIVA SOLUZIONE

Ho trovato un errore nel meccanismo di costruzione degli oggetti di classe `CCE_Line`. Il problema riguarda la variabile d'istanza `numOfHdl`. Se la costruzione di una `CCE_Line` avviene passando attraverso il metodo:

```
- (id) initStartPt: (NSPoint) stPt toPt: (NSPoint) endPt ;
```

la cosa funziona correttamente. Se invece l'inizializzazione avviene attraverso un metodo generico di init, come ad esempio accade all'interno del metodo `mouseDown:` della classe `CoverView`, la variabile rimane inizializzata a otto, invece che a due come vorrei.

Per risolvere la cosa, non ho trovato nulla di meglio che assegnare direttamente il valore all'interno del metodo:

```
- (void) buildMeWithStartPt: (NSPoint) stPt andPt: (NSPoint) endPt ;
```

Non mi pare una buona soluzione (la variabile è continuamente assegnata), ma non ho voglia di cercare una soluzione più intelligente.

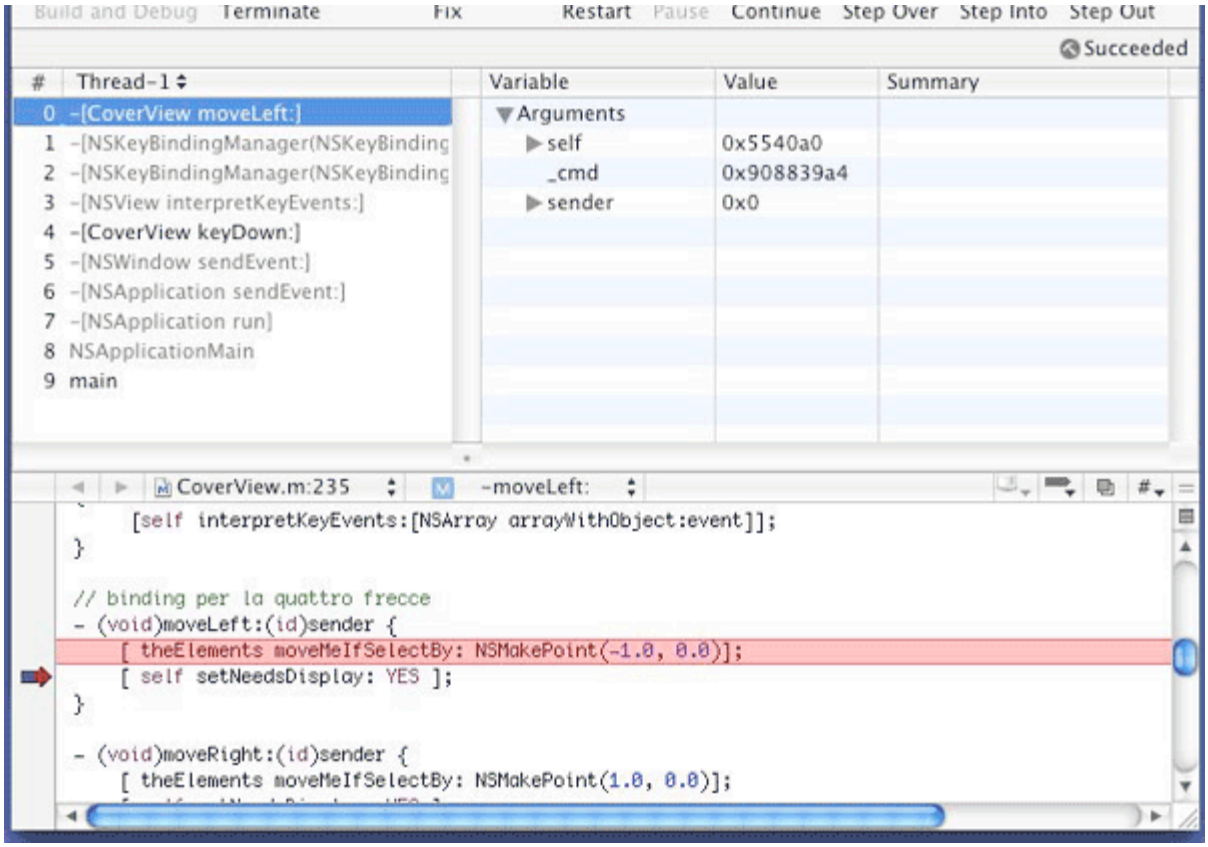
MUOVERE OGGETTI CON LA TASTIERA

Visto che ho cominciato ad utilizzare in qualche misura la tastiera, volevo rendere disponibile una funzionalità tipica di un programma per il disegno, ovvero lo spostamento degli oggetti utilizzando i tasti freccia della tastiera. Per capire come fare, ho guardato il solito esempio Sketch fornito da Apple. Qui in effetti è disponibile tale funzionalità; bene, mi son detto, copio biecamente. Ho visto un semplice metodo, con una unica istruzione:

```
- (void)keyDown:(NSEvent *)event
{
    [self interpretKeyEvents:[NSArray arrayWithObject:event]];
}
```

E basta. Nient'altro. Non ho trovato da nessun parte un meccanismo che associasse alla tastiera l'esecuzione di particolari operazioni. Guardando qui e lì il codice, ho trovato un po' di metodi che a quanto pare eseguono il lavoro necessario; ad esempio, per lo spostamento verso sinistra, è disponibile

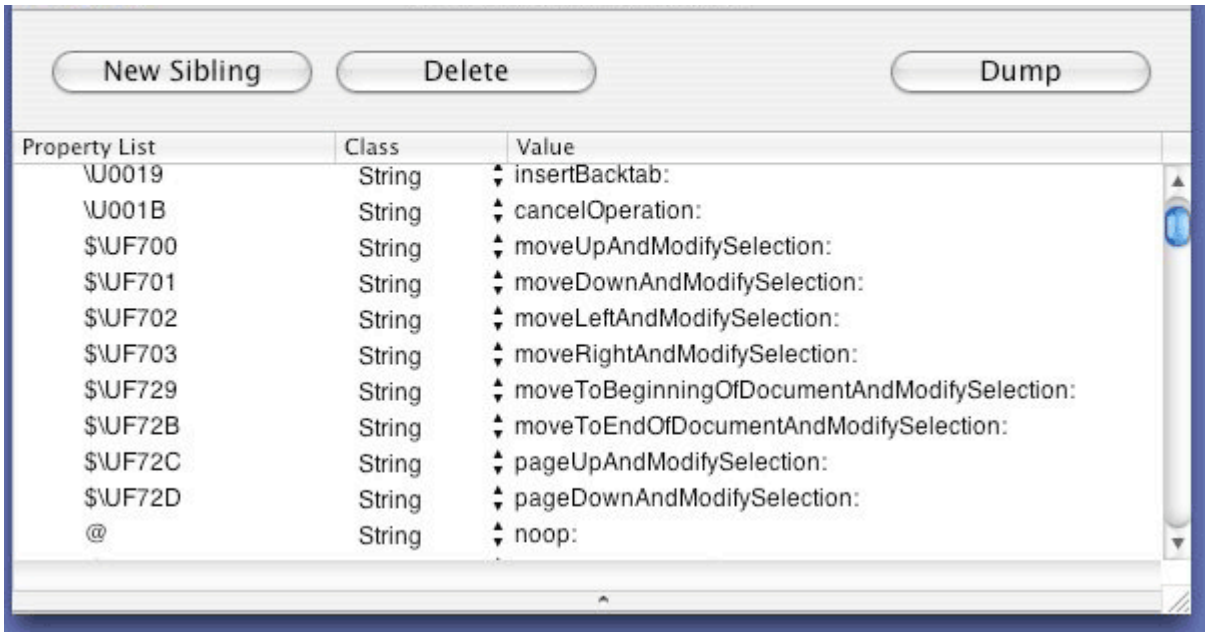
```
- (void)moveLeft:(id)sender ;
```



Non c'è stato a lungo verso che trovassi l'associazione tra i tasti freccia e l'invocazione di questi metodi. Dopo infinita pena, uso estensivo del Debugger, ma soprattutto lettura della documentazione, ho trovato il bandolo della matassa. Si tratta di una funzionalità standard offerta da Cocoa nel quadro del sistema di gestione del testo. Spendo due parole per ricapitolare la documentazione letta.

La gestione del testo utilizza tre classi: un *input server*, che si preoccupa di ricevere caratteri, elaborarli e poi inviarli ad un sistema di visualizzare; una *text view*, che appunto visualizza i caratteri; un *input manager* che si piazza davanti ad un input server, gestendo direttamente la tastiera e mandando messaggi al server stesso. La situazione classica vede la text view che riceve i caratteri grezzi; li passa al suo input manager, che li filtra. Normalmente, la maggior parte dei caratteri è passata allo input server che se li manipola, li trasforma, ne fa quel che crede, e poi li restituisce alla text view perché li visualizzi. Vi sembra complicato? in effetti lo è, ma ci sono molte cose interessanti eseguite in tutti questi passaggi. Ad esempio, lo input manager potrebbe accorgersi che l tasto premuto non è un normale carattere alfanumerico, ma, ad esempio, una freccia (!). Ora, non serve passare tale informazione al server, che non saprebbe che farsene. In effetti, lo input manager cerca di chiamare un metodo apposito. Lo input manager ha quindi a disposizione un dizionario, il *key-binding dictionary*, in cui a particolari combinazioni di tasti sono associati dei metodi. Dov'è questo dizionario? Basta leggere la documentazione per scoprire che esiste il seguente file:

```
/System/Library/Frameworks/AppKit.framework/Resources/StandardKeyBinding.dict
```



dove sono appunto conservate le associazioni standard. Sono andato a guardare questo dizionario, ad esempio con l'applicazione *Property List Editor* (come al solito, si tratta di un file in un formato XML). E qui sono presenti circa duecento combinazioni di tasti, descritti in un incomprensibile formato unicode; ad alcune combinazioni sono associati dei nomi di metodo. Verso il fondo della lista, ad esempio, sono indicati quattro metodi, `moveUp`, `moveDown`, `moveLeft` e `moveRight`, che sono associati normalmente alle quattro frecce. Se poi la freccia è premuta assieme al tasto maiuscole, i metodi di riferimento sono `moveUpAndModifySelection`, `moveDownAndModifySelection`, `moveLeftAndModifySelection` e `moveRightAndModifySelection` (come ho fatto a saperlo? Prove, prove e prove).

A questo punto, tutto si risolve in un gioco da ragazzi. Basta scrivere i seguenti metodi, i quali, appunto, di volta in volta chiedono gentilmente di spostarsi del giusto (di un solo pixel nel caso di freccia semplice, di dieci pixel quando c'è anche il tasto maiuscole premuto) a tutti gli elementi selezionati.

```
// binding per la quattro frecce
- (void)moveLeft:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(-1.0, 0.0)];
    [ self setNeedsDisplay: YES ];
}

- (void)moveRight:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(1.0, 0.0)];
    [ self setNeedsDisplay: YES ];
}

- (void)moveUp:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(0.0, -1.0)];
    [ self setNeedsDisplay: YES ];
}

- (void)moveDown:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(0.0, 1.0)];
    [ self setNeedsDisplay: YES ];
}
```

```
// binding per la quattro frecce con tasto maiuscole
- (void)moveLeftAndModifySelection:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(-10.0, 0.0)];
    [ self setNeedsDisplay: YES ];
}

- (void)moveRightAndModifySelection:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(10.0, 0.0)];
    [ self setNeedsDisplay: YES ];
}

- (void)moveUpAndModifySelection:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(0.0, -10.0)];
    [ self setNeedsDisplay: YES ];
}

- (void)moveDownAndModifySelection:(id)sender {
    [ theElements moveMeIfSelectBy: NSMakePoint(0.0, 10.0)];
    [ self setNeedsDisplay: YES ];
}
```

È forse finita qui? Nemmeno per sogno. Infatti, non funziona. Ancora una volta, dopo lungo peregrinare, esame di esempi e di documentazione, mi sono accorto di una cosa ovvia. Occorre che la `CoverView` sia indicata come la prima destinataria degli eventi associati alla tastiera (che altrimenti son dirottati verso altre gerarchie). Per fare questo, occorre aggiungere, all'interno del metodo `windowDidLoad` della classe `CoverWinCtl` la seguente istruzione (magari verso la fine del metodo stesso):

```
    [[self window] makeFirstResponder:cvrView];
```

che appunto stabilisce che la `CoverView` è il primo destinatario di ogni evento (in particolare, del fatto che qualcuno pesti sui tasti).

Capitolo 046 - Sposta, Allinea, Raggruppa e Blocca

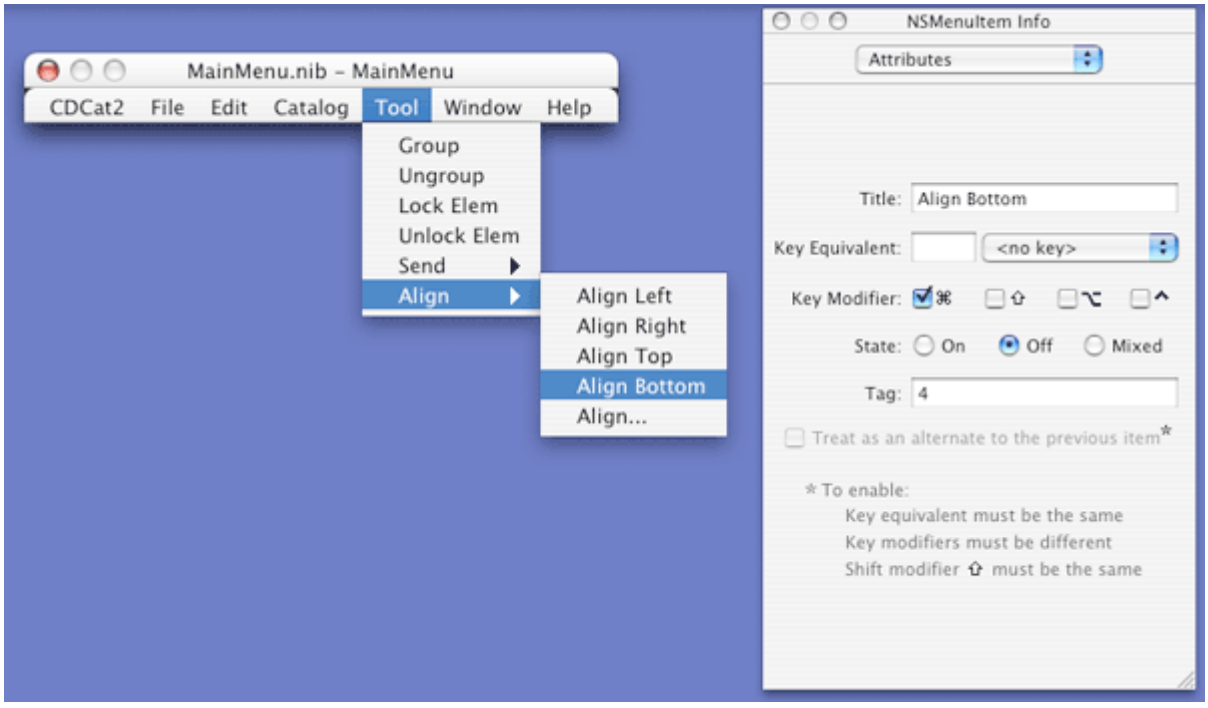
Questo capitolo, piuttosto lungo, realizza alcune tipiche funzioni presenti in un programma di disegno vettoriale: lo spostamento nell'ordine di visualizzazione; l'allineamento di più elementi secondo un certo criterio; la creazione e la distruzione di gruppi di elementi; il blocco e lo sblocco di elementi, in modo che non siano spostati o mossi accidentalmente. Tante cose, ma tutte piuttosto semplici.

Documentazione Apple.

Primo inserimento: 22 aprile 2004

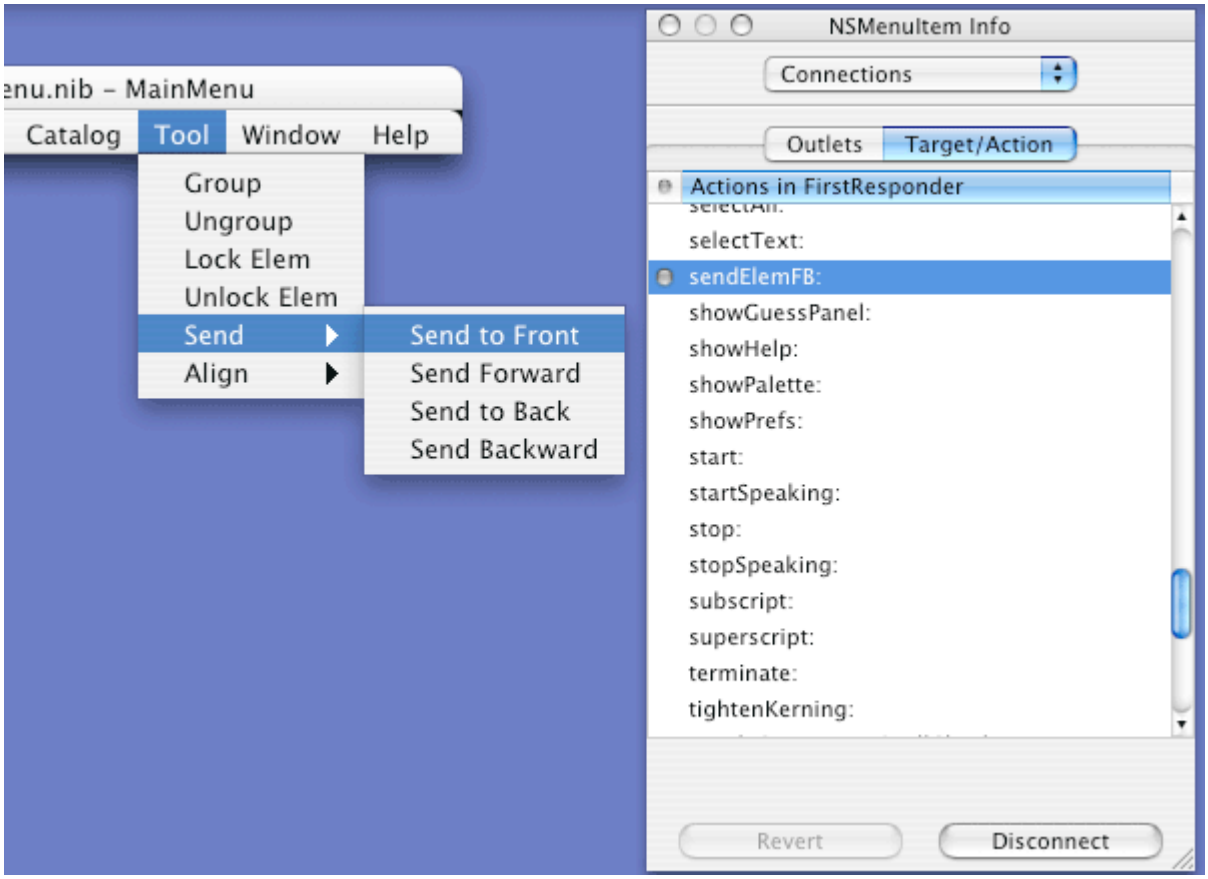
TRE COSE

Ma prima di cominciare, voglio predisporre un po' di cose.



La prima è una quasi indispensabile modifica all'interfaccia, l'inserimento di un menu per effettuare tutte le operazioni prima descritte. Ho chiamato il menu `Tool` ed ho aggiunto una serie di voci del menu e di sottomenu. È importante notare come ho assegnato alle varie voci un tag di diverso valore. Ho quindi una voce `Group` con tag 200 ed una voce `Ungroup` con tag 201; nei due sottomenu per l'allineamento e per lo spostamento dell'ordine di visualizzazione, le voci hanno tag che vanno da 1 a 4 o 5. Tutto questo mulinare di tag serve per associare un solo metodo a più di una voce. Ad esempio, tutte le voci relative all'allineamento invocano lo stesso metodo. Poi, all'interno del metodo, chiederò qual è il tag dell'elemento chiamante per capire l'operazione da

effettuare. In questo modo evito l'eccessivo proliferare di metodi, raggruppando funzionalità simili (o opposte) all'interno dello stesso blocco di codice.



La seconda funzionalità che ho aggiunto è un meccanismo per capire come sono organizzati gli elementi all'interno della vista. Ho quindi associato ad uno dei pulsanti di servizio presenti sulla finestra un metodo da eseguire quando si fa clic su di esso:

```
- ( void)
clicBtn02: (id)sender
{
    [ [ cvrView theElements] printMeWithLevel: 0 ];
}
```

Per ogni elemento ho quindi aggiunto il metodo `printMeWithLevel:` che cerca di descrivere come sia costruito l'elemento. Detto così, sembra un duplicato del metodo generico `description:`, che appunto riporta alcune informazioni sull'oggetto. Qui invece ho aggiunto un parametro, il livello, che aggiunge un po' di spazi davanti alla riga che lo descrive, in dipendenza del fatto che l'elemento appartenga o meno ad un gruppo. Il metodo per un rettangolo è piuttosto semplice:

```
- (void)
printMeWithLevel: (int) lev
{
    int i;
    char    blankChar[40];
    NSString    * msg ;
    if ( lev > 10 ) lev = 10 ;
    for ( i = 0; i < lev*4 ; i ++ )
```

```
        blankChar[i]= ' ';
    blankChar[lev*4] = 0 ;
    msg = [ NSString stringWithCString: blankChar];
    NSLog( @"%@ -> Rect (R: %7.2f %7.2f S: %7.2f %7.2f) Lw: %6.2f, F: %2d,
S: %2d, Sel: %2d, Lk: %2d", msg,
        elemLimit.origin.x, elemLimit.origin.y,
        elemLimit.size.width, elemLimit.size.height,
         cceLineWidth, cceIsFilled,cceIsStroked, cceIsSelected, cceIsLocked
);
}
```

La prima operazione del metodo consiste nel costruire una stringa di spazi, pari al numero di livello per quattro. Poi, la funzione `NSLog` scrive sulla console una stringa con un po' informazioni relative: le coordinate del rettangolo, lo spessore della linea, se l'elemento è selezionato, cose del genere.

Il metodo diventa più interessante per un elemento `CCE_ElemGroup`:

```
- (void)
printMeWithLevel: (int) lev
{
    int        i , numElem ;
    char    blankChar[40];
    CCE_BasicForm * elem ;
    NSString    * msg ;
    if ( lev > 10 ) lev = 10 ;
    for ( i = 0; i < lev*4 ; i ++ )
        blankChar[i]= ' ';
    blankChar[lev*4] = 0 ;
    msg = [ NSString stringWithCString: blankChar];
    NSLog( @"%@ -> ElemGroup of %d elements (R: %7.2f %7.2f S: %7.2f %7.2f)
Sel: %2d, Lk: %2d",
        msg, [elemArray count],
        elemLimit.origin.x, elemLimit.origin.y,
        elemLimit.size.width, elemLimit.size.height,
        cceIsSelected, cceIsLocked );

    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem printMeWithLevel: (lev+1) ];
    }
}
```

Oltre alle operazioni standard, poi c'è un ciclo su tutti gli elementi del gruppo, per i quali si ripete la procedura, ma partendo da un livello più alto di uno. Si ottiene così, in risposta ad un clic sul pulsante, una descrizione un po' criptica ma indentata per livello degli elementi presenti. La cosa sarà molto utile più avanti per capire se il meccanismo di raggruppamento/separazione funziona correttamente.

```
2004-04-19  20:56:03.845  CdCat2[1054]  ->  ElemGroup  of  11  elements
(R:    0.00    0.00 S:    1.00    1.00) Sel: 0, Lk: 0
2004-04-19 20:56:03.845 CdCat2[1054]     -> Line (P: 480.46 70.86 P: 480.46
405.35) Lw: 1.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.846 CdCat2[1054]      -> Line (P: 89.29 70.86 P: 89.29
405.35) Lw: 1.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.846 CdCat2[1054]     -> Rect (R: 70.86 70.86 S: 428.02
```

```
334.48) Lw: 1.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.846 CdCat2[1054]        -> Line (P: 513.06 405.35 P:
541.41 405.35) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.846 CdCat2[1054]      -> Line (P: 28.35 405.35 P: 56.69
405.35) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.846 CdCat2[1054]        -> Line (P: 498.89 419.52 P:
498.89 447.87) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.847 CdCat2[1054]      -> Line (P: 70.86 419.52 P: 70.86
447.87) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.847 CdCat2[1054]     -> Line (P: 513.06 70.86 P: 541.41
70.86) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.847 CdCat2[1054]      -> Line (P: 28.35 70.86 P: 56.69
70.86) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.848 CdCat2[1054]     -> Line (P: 498.89 28.35 P: 498.89
56.69) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
2004-04-19 20:56:03.848 CdCat2[1054]      -> Line (P: 70.86 28.35 P: 70.86
56.69) Lw: 0.00, F: 0, S: 1, Sel: 0, Lk: 0
```

Infine, ci sono alcune modifiche a livello di `CCE_ElemGroup`. Ho scritto un metodo specifico della classe, utile in momenti successivi.

```
- (NSMutableArray *)
getSelectedGraphics
{
    NSMutableArray * selArr = [[ NSMutableArray alloc] init ] ;
    int        i , numElem ;
    CCE_BasicForm * elem ;
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        if ([ elem cceIsSelected] )
            [ selArr addObject: elem ];
    }
    return ( selArr );
}
```

Molto semplicemente, costruisco un vettore con tutti gli elementi correntemente selezionati.

Poi, ho rimosso la versione di `setCceIsSelected:` specifica della classe, trasformandola nel metodo `cascadeSelection:`, ed ho trasformato il metodo `moveMeIfSelectBy:` in `moveSelectedElements:`. Il motivo di tutto ciò sarà chiaro molto più avanti.

BLOCCO E SBLOCCO

Può accadere talvolta di eseguire operazioni non volute, perché un elemento sul quale non volevamo operare era accidentalmente selezionato. In più, nel disegno di copertine, dovrei contare sul fatto che alcuni elementi grafici non dovrebbero essere manipolati per alcun motivo. Sto ad esempio parlando dei rettangoli che segnano i contorni della copertina stessa, delle linee di taglio, eccetera. Per facilitare le cose, ho introdotto il concetto di blocco di un elemento. Dico che un elemento può essere bloccato (`Lock Elem`), ed in tal caso non partecipa alle operazioni di movimento e spostamento, insomma, non può essere manipolato; normalmente, invece, l'elemento è sbloccato, è può essere manipolato a piacere. Ho racchiuso questo stato dell'elemento in una nuova

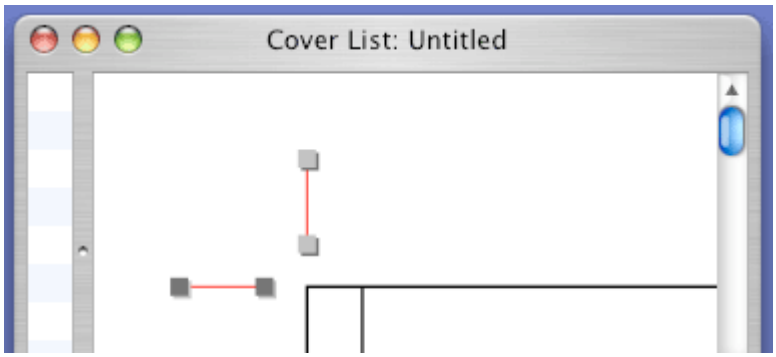
variabile d'istanza di `CCE_BasicForm`, chiamata `cceIsLocked`. Questa variabile nasce con valore falso, ma può essere modificata con i soliti metodi accessor. A livello di interfaccia utente, ci sono due voci di menù che fanno riferimento ad un unico metodo:

```
- (void)
lockElems: (id)sender
{
    NSMutableArray * selArr ;
    int         numElem, i ;
    CCE_BasicForm * elem ;
    int         menuTag = [ sender tag ];

    selArr = [ theElements getSelectedGraphics ];
    numElem = [ selArr count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ selArr objectAtIndex: i ];
        if ( menuTag == 100 )
                [ elem setCceIsLocked: YES ];
        else    [ elem setCceIsLocked: NO ];
    }
    [ self setNeedsDisplay: YES ];
}
```

Il metodo raccoglie tutti gli elementi correntemente selezionati, e li costringe ad impostare il valore della variabile `cceIsLocked` secondo quanto deciso dall'utente; la volontà dell'utente è nota attraverso il valore del tag della voce di menu, che ho opportunamente assegnato ai valori 100 per il blocco e 101 per lo sblocco.

Va da sé che il metodo accessor va riscritto per la classe `CCE_ElemGroup` in modo da propagare la modifica fino in fondo della gerarchia.

```
- (void)
setCceIsLocked:(BOOL)newVal
{
    int         i , numElem ;
    CCE_BasicForm * elem ;
    //assegno il valore a me stesso
    [ super setCceIsLocked: newVal ];
    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem setCceIsLocked: newVal ];
    }
}
```



Ho preferito utilizzare un richiamo al metodo della superclasse piuttosto che la semplice assegnazione diretta del valore per buona pratica di programmazione object-oriented.

Ho anche inserito una modifica alla funzione `ccex_drawHandleAt` per evidenziare in qualche modo gli elementi bloccati da quelli liberi. Adesso ogni chiamata alla funzione richiede un parametro booleano che dice se l'elemento è bloccato o meno (il parametro serve perchè si tratta di una funzione, e in quanto tale, diversamente da un metodo di una classe, non fa parte di alcuna classe e tanto meno ha accesso alle variabili d'istanza):

```
void
ccex_drawHandleAt( NSPoint where, BOOL how )
{
    // per ora la dimensione del manico e' fissa
    int         handleDim = 3 ;
    NSRect    hRect, rRect ;
    // disegno un rettangolo pieno attorno al punto
    hRect = NSMakeRect( where.x, where.y,2*handleDim, 2*handleDim );
    rRect = NSOffsetRect( hRect , -handleDim, -handleDim);
    hRect = NSOffsetRect(rRect, 1.0, 1.0);
    if ( how )
            [[NSColor darkGrayColor] set];
    else    [[NSColor lightGrayColor] set];
    NSRectFill(hRect);
    if ( how )
            [[NSColor lightGrayColor] set];
    else    [[NSColor darkGrayColor] set];
    NSRectFill(rRect);
}
```

In questo modo un elemento bloccato e selezionato presenta gli handle di un color grigio, piuttosto che il solito nero.

SPOSTAMENTO DI LIVELLO

Gli elementi presenti in una coverView sono conservati in un vettore, `NSMutableArray`; giocoforza, gli elementi hanno un loro ordine, che si riflette nell'ordine con cui sono disegnati. Ciò ha importanti conseguenze nel caso di elementi pieni, che coprono elementi disegnati in precedenza. Questa è una situazione tipica di tutti i programmi di disegno, e praticamente tutti i programmi di disegno possiedono un meccanismo per modificare l'ordine con cui sono rappresentati gli elementi grafici. In pratica, esistono quasi sempre quattro comandi, per spostare un elemento dietro a tutti gli altri (*back*), dietro di un livello (*backward*), avanti di un livello (*forward*), davanti a tutti gli altri (*front*). Le quattro voci di menu fanno riferimento ad uno stesso metodo, `sendElemFB:`, che sfrutta l'accennato meccanismo del tag per capire cosa fare.

```
- (void)
sendElemFB: (id)sender
{
```

```
    int        menuTag = [ sender tag ];
    switch ( menuTag ) {
    case 1 : [ theElements sendElemFrontAndBack: +2 ]; break ;
    case 2 : [ theElements sendElemFrontAndBack: +1 ]; break ;
    case 3 : [ theElements sendElemFrontAndBack: -1 ]; break ;
    case 4 : [ theElements sendElemFrontAndBack: -2 ]; break ;
    default :
        return ;
    }
    [ self setNeedsDisplay: YES ];
}
```

Ho convenuto quattro valori, tra -2 e +2, per indicare le quattro possibilità citate. Come spesso accade, il lavoro sporco viene compiuto da un altro metodo. Questo metodo, `sendElemFrontAndBack:`, è specifico della classe `CCE_ElemGroup`, e merita una discussione approfondita. Fondamentalmente, il metodo è diviso in due parti: prima mi chiedo in quale direzione mi devo spostare (in avanti o all'indietro), e poi eseguo gli spostamenti richiesti. Da notare che l'operazione sarà eseguita solo sugli elementi selezionati e non bloccati, quindi con le variabili d'istanza `cceIsSelected` a Vero e `cceIsLocked` a Falso.

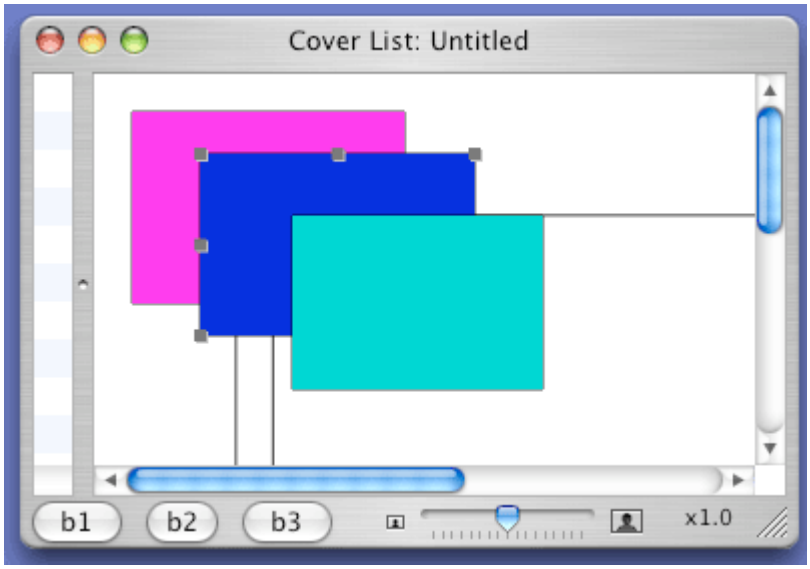
```
- (void )
sendElemFrontAndBack: (int) howMuch
{
    int        i , numElem, newIdx ;
    CCE_BasicForm * elem ;
    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    // devo distinguere tra l'avanti e l'indietro
    if ( howMuch > 0 )
    {
        // sposto in avanti
    }
    else
    {
        // sposto indietro
    }
}
```

Ho diviso il commento in tre blocchi. Nel blocco precedente c'è lo scheletro di massima, di seguito gli altri due blocchi.

Gli spostamenti in avanti sono regolati dal seguente blocco di codice. Giova ricordare che spostare in avanti un elemento significa spostare la sua posizione all'interno del vettore degli elementi verso indici più bassi.

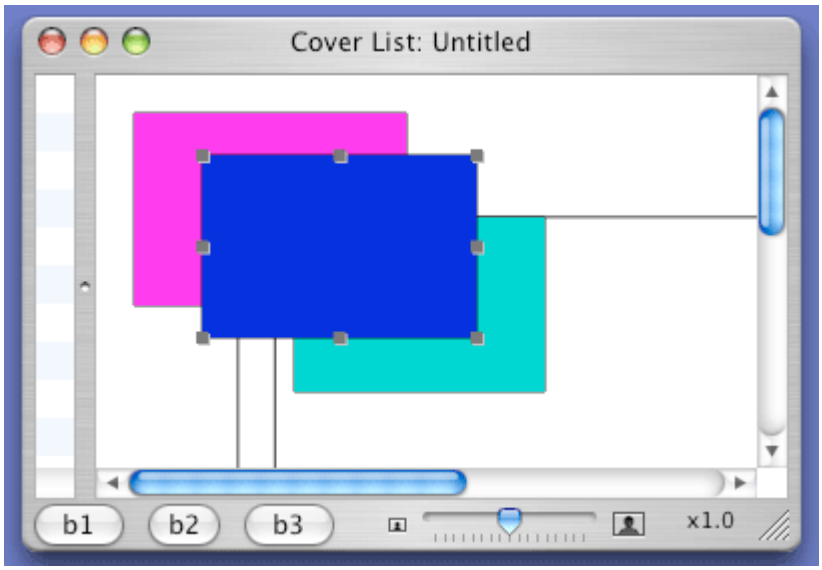
```
        for ( i = 0 ; i < numElem ; i ++ )
        {
            elem = [ elemArray objectAtIndex: i ];
            if ( [elem cceIsLocked] || ! [ elem cceIsSelected ] )
                continue ;
            // se arrivo qui, devo spostare l'elemento
            [ elem retain ];
            // comunque vada; lo tolgo dallo array
            [ elemArray removeObjectAtIndex: i ];
            // vedo cosa ne devo fare
            if ( howMuch == +1 )
                    newIdx = i-1 ;
```

```
            else    newIdx = 0 ;
            // adesso ho un nuovo indice, che aggiusto
            if ( newIdx < 0 ) newIdx = 0;
            [ elemArray insertObject: elem atIndex: newIdx ];
            [ elem release ];
        }
```



Il ciclo procede da elementi ad indice più basso verso elementi ad indice più alto. In questo modo lo spostamento avviene verso zone del vettore già esaminate, che non producono ulteriori effetti.

In primo luogo esamino l'elemento; se questo è bloccato, oppure non è selezionato, passo al successivo con l'istruzione continue. Se invece l'elemento è selezionato e libero, eseguo una operazione di `retain` e poi lo estraggo dal vettore. Il messaggio di `retain`è fondamentale (provate a toglierlo: l'applicazione muore al solito in maniera spettacolare). Infatti, rimuovendo l'elemento dal vettore, l'ambiente operativo gli invia un messaggio di `release`. Se l'elemento è posseduto solamente dal vettore (cosa che succede normalmente) il famoso contatore va a zero e l'elemento è deallocato (cosa sconveniente, visto che lo sto solo spostando).



A questo punto, devo rimpiazzare l'elemento estratto. Se lo spostamento è davanti a tutti gli altri, la cosa è semplice: basta inserirlo all'indice zero del vettore. Se invece lo sposto in avanti di una sola posizione, lo devo inserire ad un nuovo indice, che l'indice che aveva l'elemento in precedenza decrementato di 1. Dopo questa operazione, il vettore ha di nuovo `numElem` elementi, e si può procedere coll'elemento successivo. Occorre però ricordarsi di inviare un messaggio di `release` all'elemento. Infatti l'inserimento all'interno del vettore esegue anche una operazione di `retain` sull'elemento, che a questo punto è una volta di troppo.

Speculare il blocco di codice che esegue lo spostamento all'indietro, verso indici più alti. Qui è bene partire da elementi con indici alti per scender via via, in modo da evitare di considerare nuovamente elementi già spostati.

```
for ( i = numElem-1 ; i >= 0 ; i -- )
{
    elem = [ elemArray objectAtIndex: i ];
    if ( [elem cceIsLocked] || ! [ elem cceIsSelected ] )
        continue ;
    // se arrivo qui, devo spostare l'elemento
    [ elem retain ];
    // comunque vada; lo tolgo dallo array
    [ elemArray removeObjectAtIndex: i ];
    // vedo cosa ne devo fare
    if ( howMuch == -1 )
            newIdx = i+1 ;
    else    newIdx = numElem-1 ;
    // adesso ho un nuovo indice, che aggiusto
    if ( newIdx >= numElem ) newIdx = numElem-1 ;
    [ elemArray insertObject: elem atIndex: newIdx ];
    [ elem release ];
}
```

Da notare l'indice da utilizzare quando si vuole spostare l'elemento in fondo, che è uno di meno della dimensione originale del vettore. In effetti, ho appena estrato un elemento, quindi il vettore è più piccolo di uno.

In entrambi i pezzi di codice, prima dell'effettivo inserimento dell'elemento, c'è un controllo sul valore dell'indice; se troppo basso (minore di zero o superiore a `numElem`), occorre aggiustarlo. La cosa accade, ad esempio, quando intendo spostare in avanti un elemento che è già davanti a tutti (e che quindi ha l'indice zero; spostandoli in avanti sottraggo uno...).

Ecco, come ho scritto la frase precedente, mi sono accorto di un miglioramento immediato. In effetti, non occorre considerare il primo elemento del vettore negli spostamenti in aventi, e l'ultimo negli spostamenti all'indietro. Questi elementi non si sposteranno comunque. Ecco quindi che le istruzioni di controllo sull'indice non servono più, a patto che i due cicli for diventino rispettivamente:

```
for ( i = 1 ; i < numElem ; i ++ )
for ( i = numElem-2 ; i >= 0 ; i -- )
```

A questo punto, ci si può divertire a disegnare elementi sulla coverView, e poi, attraverso semplici comandi di menu, modificare l'aspetto del disegno, spostando avanti ed indietro gli elementi nell'ordine di disegno.

ALLINEAMENTO

Un'altra operazione di manipolazione spesso presente nei programmi di disegno è quella di allineamento di un insieme di oggetti secondo diversi criteri: a destra, a sinistra, in alto, in basso, al centro (orizzontalmente), nel mezzo (verticalmente). Per non altre particolari ragioni se non complicarmi la vita, ho costruito quattro voci di menu per allineare secondo i primi quattro criteri, poi ho aggiunto una quinta voce di menu per aprire una finestra dove poter allineare gli elementi utilizzando anche gli altri due criteri rimasti. Le quattro voci di menu fanno riferimento allo stesso metodo `elemAlign:`, che distingue il suo comportamento attraverso il tag della voce di menu da cui il metodo è stato invocato.

Ma prima di procedere con l'allineamento di oggetti, mi occorrono un po' di metodi utili. Poiché in seguito mi occorre sapere quali sono le coordinate limite destra, sinistra, superiore ed inferiore di un elemento, mi premuro di definire i metodi che le calcolino per ogni elmento grafico. Sono piuttosto facili, ne riporto solo un paio:

```
- (NSNumber*) getElemBottom
{
          return   (   [   NSNumber   numberWithFloat:   (elemLimit.origin.y   +
elemLimit.size.height)] );
}
- (NSNumber*) getElemMiddle
{
       return   (   [   NSNumber   numberWithFloat:   (elemLimit.origin.y   +   0.5   *
elemLimit.size.height)] );
}
```

Ad esempio, la coordinata *bottom*, inferiore, di un elemento, è data dalla ordinata dell'origine del rettangolo cui è sommata la dimensione verticale (da ricordare qui che le coordinate veriticali, nella coverView, sono rovesciate). La coordinata *middle*, centrale in verticale, è banalmente l'ordinata più metà dell'altezza. I più accorti dei lettori potranno chiedersi perché diavolo costruisco un oggetto della classe `NSNumber` per fare questa semplice operazione, dove un valore di ritorno `float` sarebbe stato più che sufficiente. La cosa sarà chiara tra qualche riga.

Torno adesso al metodo invocato direttamente dalle voci di menu. Conviene soffermarsi perché contiene qualche concetto nuovo, specifico di Cocoa.

```
- (void)
elemAlign: (id)sender
{
    int        menuTag = [ sender tag ];
    [ self executeAlignment: menuTag ];
}
```

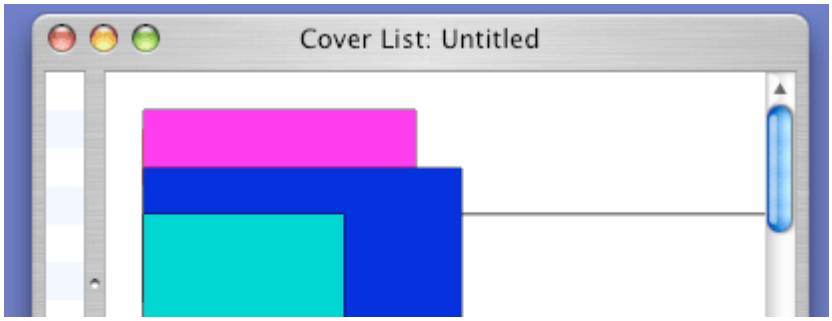
Come al solito, vi ho mentito. Il lavoro sporco è svolto da un altro metodo. Qui mi limito ad estrarre il tag ed a comandare una ulteriore operazione.

Per eseguire un allineamento, occorre determinare la coordinata di allineamento. Fissiamo le idee sull'allineamento a sinistra. Allora, occorre determinare tra tutti gli elementi selezionati una ascissa sulla quale eseguire l'allineamento. Tipicamente, si tratta dell'ascissa più piccola degli estremi sinistri dei rettangoli dei vari elementi

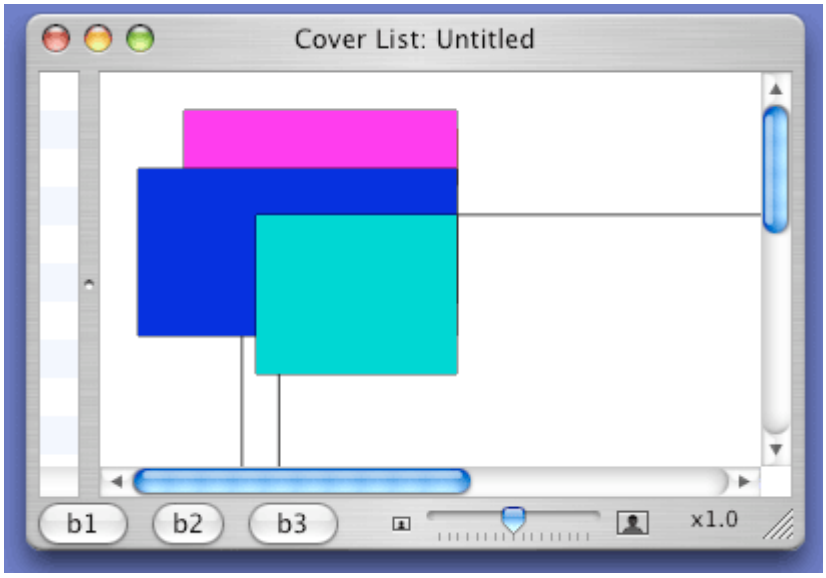
(`getElemLeft`). Per una allineamento a destra, bisogna cercare l'ascissa più grande degli estremi destri (`getElemRight`), e così via per l'allineamento in alto (`getElemTop`) ed in basso (`getElemBottom`).

Se uno non sapesse programmare, scriverebbe quattro metodi distinti, uno per ciascuna direzione di allineamento. Ma io invece ho deciso di avere un unico metodo in grado di fare tutte le cose.

```
- (void)
executeAlignment: (int) menuTag
{
    NSMutableArray * selArr ;
    int        numElem, i ;
    float curTop, newTop, delta ;
    CCE_BasicForm * elem ;
    BOOL    chooseDir ;
    SEL        getDim ;

    switch ( menuTag ) {
    case 1 : chooseDir = TRUE ; getDim = @selector(getElemLeft) ; break ;
    case 2 : chooseDir = FALSE ; getDim = @selector(getElemRight) ; break ;
    case 3 : chooseDir = TRUE ; getDim = @selector(getElemTop) ; break ;
     case 4 : chooseDir = FALSE ; getDim = @selector(getElemBottom) ; break
;
    default : return ;
    }
    selArr = [ theElements getSelectedGraphics ];
    numElem = [ selArr count ];
    elem = [ selArr objectAtIndex: 0 ];
    curTop = [[ elem performSelector: getDim ] floatValue ];
    for ( i = 1 ; i < numElem ; i ++ )
    {
        elem = [ selArr objectAtIndex: i ];
        newTop = [[ elem performSelector: getDim ] floatValue ];
        if ( chooseDir && newTop < curTop )
            curTop = newTop ;
        else if ( ! chooseDir && newTop > curTop )
            curTop = newTop ;
    }
    // adesso che ho il top, sposto tutto
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ selArr objectAtIndex: i ];
        delta = curTop - [[ elem performSelector: getDim ] floatValue ];
        if ( menuTag == 1 || menuTag == 2 )
                [ elem moveMeIfSelectBy: NSMakePoint( delta, 0 )];
        else    [ elem moveMeIfSelectBy: NSMakePoint( 0, delta )];
    }
    [ self setNeedsDisplay: YES ];
}
```



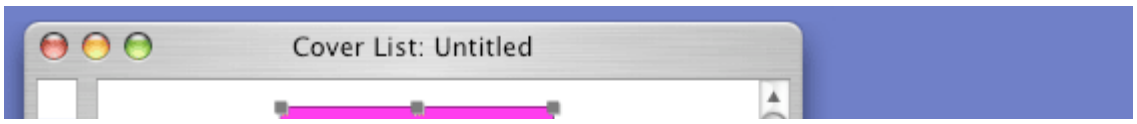
Per prima cosa, determino due variabili che mi saranno molto utili. La prima, `chooseDir`, mi serve per capire se devo cercare un minimo (`TRUE`) o un massimo (`FALSE`). La seconda variabile è una cosa strana: è un metodo, o selettore, nella più corretta terminologia ObjC. In altre parole: nella variabile `getDim` si trova il metodo che devo utilizzare per determinare l'estremo. Da qui in poi è tutta discesa: costruisco un vettore con tutti gli elementi selezionati, e su questo vettore eseguo la ricerca di estremo. Il valore lo ottengo eseguendo il metodo indicato dalla variabile `getDim` sull'elemento, ed il criterio di scelta dipende dal valore di `chooseDim`. Ora, il meccanismo `performSelector:` richiede che il selettore restituisca un oggetto, ed è questo il motivo per cui i metodi `getElemQualcosa` restituiscono tutti un `NSNumber`, invece che un più semplice `float`.

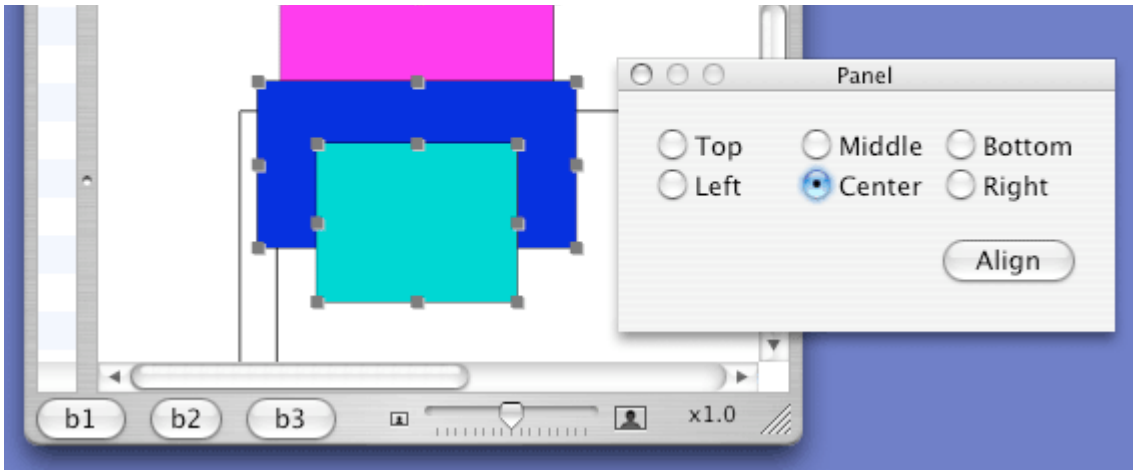


Quando, alla fine del primo ciclo `for`, ho determinato la coordinata estremale, eseguo il secondo ciclo `for`, che esegue tutti gli spostamenti. Anche qui, sfrutto il meccanismo `performSelector:` per determinare l'ammontare dello spostamento (che è sempre in una sola direzione, per poi discriminare la direzione con il valore del tag.

CENTRAGGIO

Argomento molto simile all'allineamento è il centraggio, in orizzontale e verticale, di un insieme di elemento. Ero rimasto con il fatto che queste operazioni sono invocate attraverso una finestra piuttosto che da menu.





La finestra che ho prodotto è particolarmente brutta, ma svolge il suo ruolo. Come al solito, ho dovuto costruire una classe controllore `AlignPaletteWinCtl` con un po' di metodi aggiuntivi per costruirla, aprirla, e tenere traccia in maniera acconcia della coverView presente in prima fila. L'unico pulsante chiama il metodo `executeAlign`, all'interno del quale si recupera quale dei sei pulsanti è correntemente selezionato. Ho fatto in modo che il tag di questi pulsanti corrispondesse a quello della voce di menu con la stessa funzione. Quindi il codice è molto semplice.

```
- (IBAction)executeAlign:(id)sender
{
    // recupero il tag della cella selezionata
    if ( theDrawView )
    {
        int selTag = [[ selectedAlign selectedCell] tag] ;
        if ( selTag == 5 || selTag == 6 )
                [ theDrawView executeCentering: selTag ];
        else    [ theDrawView executeAlignment: selTag ];
    }
}
```

Se c'è una coverView davanti a tutti, recupero il tag. Se questo è compreso tra 1 e 4, ricado nel caso di allineamento, e quindi chiamo il metodo `executeAlignment:`. Negli altri due casi, chiamo un nuovo metodo, `executeCentering:`, che esegue quanto necessario.

Ancora una volta mi complico la vita, e decido che la coordinata di riferimento è calcolata in modo particolare. Calcolo per ogni elemento la coordinata massima e quella minima, e cerco ciascun valore estremale. Alla fine, la coordinata di riferimento è quella di mezzo. Il giochino è sempre lo stesso: determino quale metodo serve per calcolare la coordinata. Poi eseguo una ricerca contemporanea di massimo e di minimo sulla stessa coordinata.

```
- (void)
executeCentering: (int) menuTag
{
    NSMutableArray * selArr ;
    int         numElem, i ;
    float curMax, curMin, curMid, newTop, delta ;
    CCE_BasicForm * elem ;
    SEL         getDim ;
```

```
    // middle: centro le ordinate, center: centro le ascisse
    switch ( menuTag ) {
    case 5 : getDim = @selector(getElemMiddle) ; break ;
    case 6 : getDim = @selector(getElemCenter) ; break ;
    default : return ;
    }
    selArr = [ theElements getSelectedGraphics ];
    numElem = [ selArr count ];
    elem = [ selArr objectAtIndex: 0 ];
    curMax = curMin = [[ elem performSelector: getDim ] floatValue ];
    for ( i = 1 ; i < numElem ; i ++ )
    {
        elem = [ selArr objectAtIndex: i ];
        newTop = [[ elem performSelector: getDim ] floatValue ];
        if ( newTop < curMin )
            curMin = newTop ;
        else if ( newTop > curMax )
            curMax = newTop ;
    }
    // adesso che ho il max ed il min, sposto tutto
    curMid = 0.5 * (curMax + curMin );
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ selArr objectAtIndex: i ];
        delta = curMid - [[ elem performSelector: getDim ] floatValue ];
        if ( menuTag == 6 )
                [ elem moveMeIfSelectBy: NSMakePoint( delta, 0 )];
        else    [ elem moveMeIfSelectBy: NSMakePoint( 0, delta )];
    }
    [ self setNeedsDisplay: YES ];
}
```

La coordinata di riferimento è la semisomma dei due estremi. Poi, spazzolo tutti gli elementi e li sposto secondo necessità. Tutto come prima, senza troppi problemi.

GRUPPI

È arrivato finalmente il momento di parlare dei gruppi (è da qualche capitolo che mi tiro dietro questo argomento). Ho già le due solite voci di menu per costruire un gruppo (`Group`) e per disfarlo (`ungroup`). Il metodo è lo stesso, e distingue le operazioni in base al tag.

```
- (void)
groupElem: (id)sender
{
    int         menuTag = [ sender tag ];

    if ( menuTag == 200 )
    {
        [ theElements makeGroupFromSelected ];
    }
    else
    {
    // istruzioni per disfare un gruppo
    }
    [ self setNeedsDisplay: YES ];
}
```

Il metodo chiama come al solito un altro metodo, `makeGroupFromSelected`, per

costruire un gruppo, mentre esegue direttamente le operazioni per disfarlo (ma le mostro poi).

Il metodo è piuttosto lungo e lo commento in vari segmenti.

```
- (void)
makeGroupFromSelected
{
    int        i , numElem, firstIdx = -1, lastIdx ;
    CCE_BasicForm * elem ;
    CCE_ElemGroup * newGrp ;
    float        xmin, xmax, ymin, ymax ;
    // ciclo su tutti gli elementi del gruppo
    lastIdx = numElem = [ elemArray count ];
    newGrp = [[[ CCE_ElemGroup alloc] init ] autorelease ];
    for ( i = numElem-1 ; i >= 0; i -- )
    {
        elem = [ elemArray objectAtIndex: i ];
        if ( [elem cceIsLocked] || ! [ elem cceIsSelected ] )
            continue ;
        // se arrivo qui, l'elemento e' libero e selezionato
        [ elem retain ];
        if ( firstIdx == -1 )
            firstIdx = i ;
        if ( lastIdx > i )
            lastIdx = i ;
        // comunque vada; lo tolgo dallo array
        [ elemArray removeObjectAtIndex: i ];
        // e lo aggiungo al nuovo gruppo
        [ [newGrp elemArray] addObject: elem ];
        [ elem release ];
    }
    // quando arrivo qui, ho costruito un nuovo gruppo
```

L'idea è di esaminare tutti gli elementi e, nel caso siano selezionarli, toglierli dal vettore principale ed inserirli in un nuovo oggetto della classe `CCE_ElemGroup` creato allo scopo. Il vettore è spazzolato da indici alti fino a zero dal momento che, estraendo elementi, il vettore stesso si accorcia. Nello spazzolare il vettore, mi metto da parte due indici: la variabile `firstIdx` conserva l'indice del primo elemento selezionato, mentre `lastIdx` conserva l'indice dell'elemento selezionato di fronte a tutti gli altri.

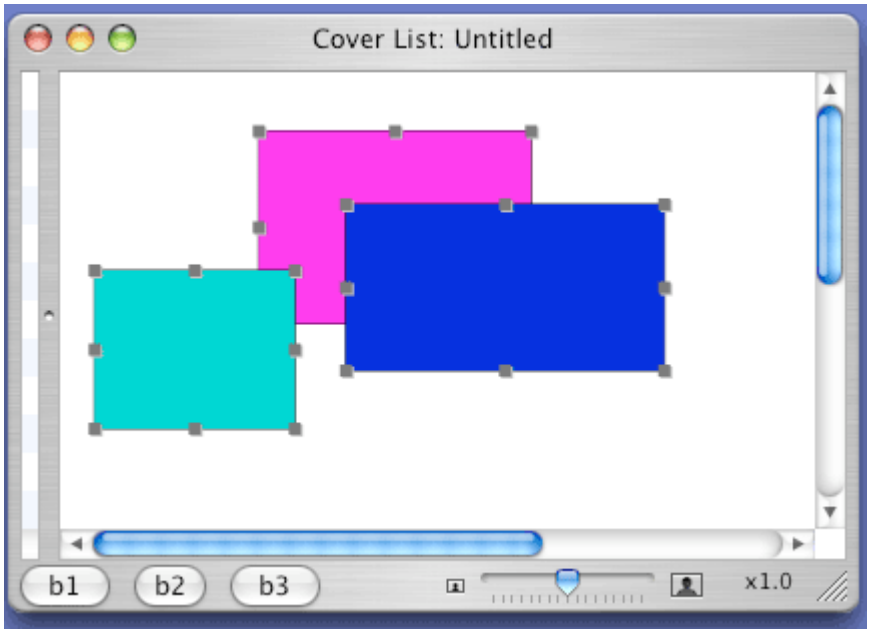
```
    // se non c'e' nulla, faccio nulla
    if ( firstIdx == -1)
        return ;
    // se c'e' un solo elemento, lo rimetto dov'era
    if ( [ [newGrp elemArray] count ] == 1 )
    {
            [ elemArray insertObject: [[newGrp elemArray] objectAtIndex:0]
atIndex: firstIdx];
        return ;
    }
```

L'indice `firstIdx` è utilizzato allo scopo di considerare due casi particolari. In primo luogo, se la variabile non è stata modificata, significa che non ci sono elementi selezionati; non ha quindi senso proseguire con le operazioni. Se invece è selezionato un singolo elemento, non ha senso costruire un gruppo di un solo elemento. Quindi, lo reinserisco nel posto in cui si trovava, ed è finita lì.

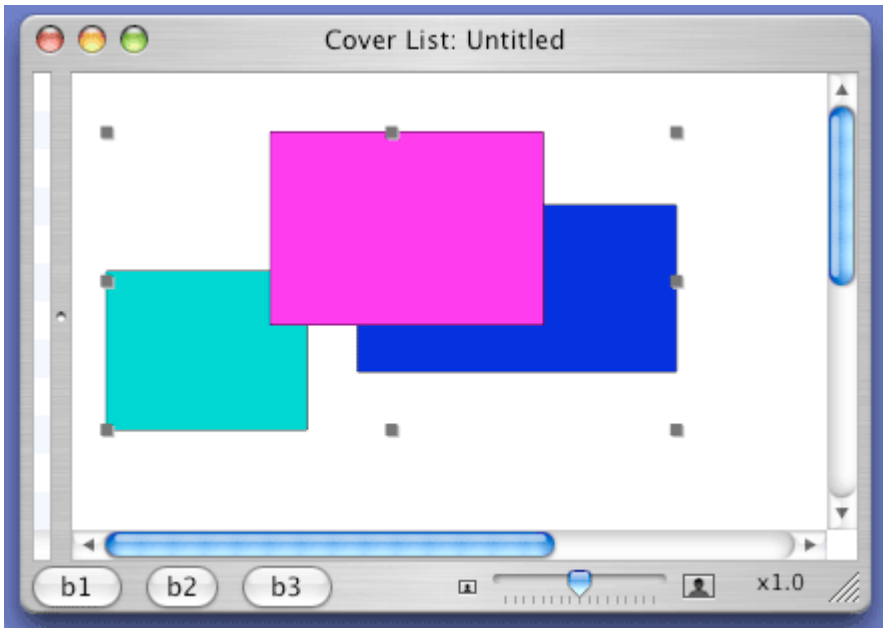
```
    // finalmente, se arrivo qui, ho un gruppo vero e proprio
    // aggiusto tutti i valori del gruppo
    // calcolo il rettangolo che include il tutto
    numElem = [ [newGrp elemArray] count ];
    elem = [ [newGrp elemArray] objectAtIndex: 0 ];
    ymin = [[ elem getElemTop] floatValue];
    xmin = [[ elem getElemLeft] floatValue];
    xmax = [[ elem getElemRight] floatValue];
    ymax = [[ elem getElemBottom] floatValue];
    // ciclo su tutti gli elementi del gruppo
    for ( i = 1 ; i < numElem ; i ++ )
    {
        elem = [ [newGrp elemArray] objectAtIndex: i ];
        // cerco i limiti del rettangolo circoscritto
        if ( ymin > [[ elem getElemTop] floatValue])
            ymin = [[ elem getElemTop] floatValue];
        if ( xmin > [[ elem getElemLeft] floatValue] )
            xmin = [[ elem getElemLeft] floatValue];
        if ( xmax < [[ elem getElemRight] floatValue] )
            xmax = [[ elem getElemRight] floatValue];
        if ( ymax < [[ elem getElemBottom] floatValue] )
            ymax = [[ elem getElemBottom] floatValue];
    }
    // attribuisco il rettangolo che include il gruppo
    [ newGrp setElemLimit: NSMakeRect( xmin, ymin, xmax-xmin, ymax-ymin) ];
```

Una volta costruito l'elemento gruppo, occorre predisporre alcune sue variabili interne. Ce ne è una che è particolarmente utile, ovvero il rettangolo che racchiude tutti gli elementi. Il gioco è facile: basta spazzolare tutti gli elementi del gruppo, e pigliare le coordinate estreme. Alla fine, costruisco il rettangolo in maniera molto semplice.

```
    // tutti gli elementi interni non sono selezionati
    [ newGrp cascadeSelection: NO ];
    // il gruppo invece rimane selezionato
    [ newGrp setCceIsSelected: YES ];
    // inserisco il gruppo nel posto opportuno
    [ elemArray insertObject: newGrp atIndex: lastIdx];
}
```



Sono arrivato alla fine del metodo. Occorre in primo luogo deselezionare tutti gli elementi che adesso sono all'interno del gruppo, attraverso il metodo `cascadeSelection:`, per trasferire la proprietà di essere selezionata al gruppo stesso. Qui si vede la ragione per cui il metodo `setCceIsSelected:` non ha più una realizzazione specifica per `CCE_ElemGroup`: nella vecchia realizzazione, tutti gli elementi del gruppo avrebbero ereditato la caratteristica di essere selezionati. Ciò non è normalmente un problema, se non fosse che un elemento selezionato disegna tutti gli handle del contorno, cosa che invece io voglio evitare. Voglio che un gruppo selezionato mostri solo gli handle relativi al contorno (rettangolo) principlae, e non tutti quelli dei singoli elementi.



Infine, ultima istruzione, inserisco il gruppo all'interno del vettore degli elementi della coverView. Lo faccio a partire dall'indice `lastIdx`: in questo modo, il gruppo si posiziona, nell'ordine fronte/retro, nel posto in cui si trovava il primo degli elementi selezionati che adesso fanno parte del gruppo.

C'è adesso da esaminare come effettuare la rottura del gruppo, eliminando quindi l'elemento della classe `CCE_ElemGroup`, rimpiazzandolo con tutti gli elementi del gruppo stesso. Ecco il pezzo di codice relativo.

```
        NSMutableArray * selArr ;
        int         numElem, i ;
        CCE_BasicForm * elem ;

        selArr = [ theElements elemArray ];
        numElem = [ selArr count ];
        for ( i = numElem-1 ; i >= 0; i -- )
        {
            elem = [ selArr objectAtIndex: i ];
            if ( ! [ elem cceIsSelected ])
                continue ;
            // se arrivo qui, l'elemento e' selezionato
            // ma potrebbe non essere un gruppo
            if ( ! [elem isKindOfClass:[CCE_ElemGroup class]] )
                continue ;
            // se arrivo qui, l'elemento e' selezionato ed e' un gruppo
```

```
            [ elem retain ];
            // lo elimino dall'insieme degli oggetti
            [ [theElements elemArray] removeObjectAtIndex: i ];
            // e poi appiccico gli elementi
            [ elem ungroupAndInsertIn: theElements startingFrom: i ];
            [ elem release ];
        }
```

Anche qui, si spazzolano gli elementi dal più alto al più basso. Se l'elemento non è selezionato, o non è un gruppo, si salta. Se si arriva ad un vero e proprio gruppo, lo si toglie dal vettore, e poi, utilizzando un altro metodo, si inseriscono gli elementi del gruppo all'interno del vettore principale.

```
- (void )
ungroupAndInsertIn: (CCE_ElemGroup *) masterArray startingFrom: (int) idx
{
    int        i , numElem ;
    CCE_BasicForm * elem ;
    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    // per ogni elemento presento, lo aggiungo
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem setCceIsSelected: YES ] ;
        [ [masterArray elemArray] insertObject: elem atIndex: idx ];
    }
}
```

La cosa è molto semplice: si spazzolano tutti gli elementi del gruppo, e li si inseriscono uno dopo l'altro al posto giusto. Da notare come ogni singolo elemento adesso è selezionato (sono partito da un gruppo selezionato, ed ottengo un insieme di elementi selezionati).

Non è finita qui. Ho scoperto molti problemi nella gestione dei gruppi. Sono tante piccole cose, che però ci metterei troppo a discutere, e questo capitolo è fin troppo lungo. Cercate nel codice, anche se temo che, nei prossimi capitoli, scoprirò tante cose che non funzionano.

Capitolo 047 - Grigliate

Un capitolo facile e breve che discute di come aggiungere una griglia per facilitare il posizionamento degli oggetti.

Come al solito, documentazione Apple.

Primo inserimento: 5 maggio 2004

GRIGLIA

Ancora una volta, ricorro alla suggestione dei soliti programmi di disegno per introdurre una nuova funzione. Voglio che all'interno della coverView sia possibile posizionare gli elementi sui punti di griglia. Ovviamente l'utente deve poter decidere se utilizzare o meno la griglia, la spaziatura della griglia, il colore delle linee guida... Riassumo il tutto attraverso quattro variabili d'istanza (ed i relativi metodi accessor), che pongo all'interno della classe `CoverView`. In questo modo ogni coverView ha la sua griglia personalizzata.

```
    // parametri per la visualizzazione della griglia
    int                 gridSpacing ;
    BOOL                showGrid ;
    BOOL                snapToGrid ;
    NSColor         * gridLineColor ;
```

C'è un intero che rappresenta la spaziatura della griglia; è intero perché rappresenta pixel. Le due variabili booleane successive indicano rispettivamente se la griglia è da visualizzare o meno, e se il tracciamento degli elementi all'interno della coverView debba avvenire o meno solo sui punti della griglia. Oltre al tracciamento, la griglia regola anche lo spostamento ed il ridimensionamento degli elementi. L'ultima variabile tiene conto del colore con il quale è disegnata la griglia.

All'interno del metodo `drawRect:` della classe `CoverView` aggiungo poi la chiamata ad un metodo specifico per il disegno della griglia

```
// il disegno della vista
- (void)drawRect:(NSRect)rect
{
    // sfondo bianco
    [[ NSColor whiteColor] set ];
    NSRectFill( rect );
    // se c'e' la griglia, la disegno
    if ( showGrid )
        [ self drawGrid: rect ];
    // poi dico di disegnarsi a tutti gli elementi
    [ [self theElements] drawElement: rect ];
...

}
```

Il disegno della griglia è una delle prime operazioni di disegno (dopo il riempimento della finestra col colore di sfondo) proprio per evitare che il disegno della griglia si

sovrapponga a quello degli elementi.

Il metodo `drawGrid:` è piuttosto semplice dal punto di vista concettuale, e si complica nei dettagli.

Ma prima di cominciare, due utili funzioni.

```
NSPoint
getNearestSnapPoint( NSPoint point, CoverView * view)
{
    BOOL    gridOn = [ view snapToGrid ];

    if ( gridOn == NO )
        return ( point );
    return ( getNearestGridPoint(point, view) );
}

NSPoint
getNearestGridPoint( NSPoint point, CoverView * view)
{
    int         gridSpacing = [ view gridSpacing ];
    NSPoint     retPt ;

    retPt.x = round(point.x / (float) gridSpacing) * gridSpacing ;
    retPt.y = round(point.y / (float) gridSpacing) * gridSpacing ;
    return ( retPt );
}
```

Scopo delle funzioni è di trasformare una coppia di coordinate generiche (un punto) in una coordinata sulla griglia. La prima funzione serve nei casi in cui debbo trasformare un punto per ottenere una locazione di *snap*, ovvero un punto dove cominciare o terminare una operazione di disegno o di spostamento. Il punto rimane immodificato se la variabile `snapToGrid` è `NO`, altrimenti si trasforma in accordo alla seconda funzione.

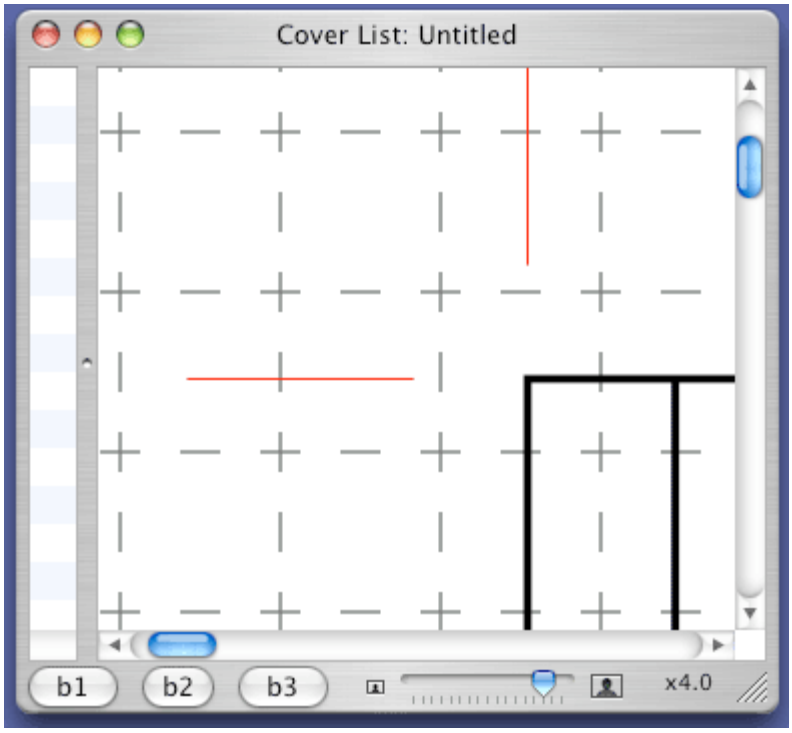
Qui c'è un giochino matematico per calcolare il punto sulla griglia: prima divido la coordinata per la spaziatura della griglia. Arrotondo il risultato all'intero più vicino, usando la funzione standard C `round(.)`, poi moltiplico nuovamente per la spaziatura. In questo modo ottengo la coordinata intera, multipla della spaziatura, più vicina alla coordinata di partenza.

DISEGNO DELLA GRIGLIA

Per disegnare la griglia, prevedo una variabile `NSBezierPath` che tenga conto delle linee veriticali ed orizzontali. Per evitare di coprire il mondo intero col disegno di una griglia, lo faccio solo all'interno del rettangolo visualizzato. Trovo quindi le due coppie di coordinate sulla griglia più vicine a due vertici opposti del rettangolo. Queste saranno le coordinate di partenza.

```
- (void)
drawGrid:(NSRect)rect
{
    NSPoint         topLeft, bottomRight ;
    float          kk, strokePattern[2] ={0.25*gridSpacing, 0.25*gridSpacing
} ;
```

```
    NSBezierPath * curPath1 = [[[ NSBezierPath init ] alloc ] autorelease ]
;

    topLeft = getNearestGridPoint( rect.origin, self );
         bottomRight   =   NSMakePoint(   rect.origin.x   +   rect.size.width,
rect.origin.y + rect.size.height);
    bottomRight = getNearestGridPoint( bottomRight, self );
    [ gridLineColor set ];
    for ( kk = topLeft.x ; kk <= bottomRight.x; kk += gridSpacing )
    {
        [curPath1 moveToPoint: NSMakePoint( kk, topLeft.y-gridSpacing) ];
         [curPath1 lineToPoint: NSMakePoint( kk, bottomRight.y+gridSpacing)
];
    }
    for ( kk = topLeft.y ; kk <= bottomRight.y ; kk += gridSpacing )
    {
        [curPath1 moveToPoint: NSMakePoint( topLeft.x-gridSpacing, kk) ];
         [curPath1 lineToPoint: NSMakePoint( bottomRight.x+gridSpacing, kk)
];
    }
    [ curPath1 setLineWidth: 0];
           [   curPath1   setLineDash:   strokePattern   count:   2   phase:
gridSpacing*0.125];
    [ curPath1 stroke ];
}
```



Il primo ciclo `for` serve a disegnare le linee verticali. Muovo la penna in testa, poi disegno una linea verticale fino in fondo. In realtà, la coordinata verticale potrebbe essere dentro o fuori al rettangolo (l'arrotondamento funziona nei due sensi). Per questo motivo, piuttosto che stare a chiedermi se la coordinata è interna o esterna, o fare altre considerazioni sull'argomento, mi limito a disegnare una linea un po' più lunga, e sicuramente fuori dal rettangolo. Ripeto lo stesso giochetto sulle linee orizzontali. Poi, finalmente, imposto i parametri della linea. Dico che lo spessore sia il minimo possibile (zero, appunto), e poi dico che la griglia è composta da linee tratteggiate. Con il metodo `setLineDash:count:phase:` dico appunto come è fatto il tratteggio. Ho costruito un

vettore che dice che il tratteggio è formato da un quarto di griglia (0.25) pieno ed un quarto di griglia vuoto. Così facendo, ad ogni spaziatura ho due tratti pieni e due tratti vuoti. Ma nel disegno dico di spostare il tratteggio di un ottavo (0.125) di griglia. Ecco quindi che ottengo un interessante effetto: i punti della griglia sono individuati da una croce, e tra una croce e la successiva si trova un segmento perfettamente centrato.

DISEGNARE CON LA GRIGLIA

Un utente potrebbe desiderare la griglia solo per il gusto di avere delle linee di riferimento, ma il più delle volte la griglia è utilizzata per posizionare in maniera opportuna gli elementi. Ecco quindi che devo modificare tutti i metodi di disegno e di spostamento per tenere conto di queste esigenze.

La cosa è più semplice di quanto si possa pensare. In pratica, occorre modificare solo i metodi `handleCreation:` e `handleResize:` della classe `CCE_BasicForm` (e di tutte le sottoclassi che li sovrascrivono). In pratica, si tratta solo di modificare i metodi di `CCE_BasicForm` e `CCE_Line`. In effetti, basta eseguire la seguente funzione

```
    point1 = getNearestSnapPoint( point1, view);
```

su ogni punto che si intende piazzare. La funzione stessa si preoccupa di capire se la griglia è attiva (e quindi esegue delle modifiche sulle coordinate del punto) oppure no (e quindi lascia inalterato il punto stesso.

Poi, c'è da fare una cosa molto simile sul metodo `handleMoveAll` della classe `CoverView`, per gestire il fatto che, con la griglia attiva, il movimento può avvenire solo per coordinate multiple della spaziatura della griglia. Anche qui, la chiamata della funzione `getNearestSnapPoint` su ogni punto notevole del metodo è sufficiente allo scopo.

Infine, ci sono altri metodi che spostano gli elementi, e sono quelli governati dalla tastiera. Con la griglia attiva, i quattro tasti freccia non è corretto che spostino gli elementi selezionati di un solo pixel: è meglio che li spostino di una intera spaziatura di griglia. Ecco quindi che ad esempio devo modificare il metodo `moveRight:`

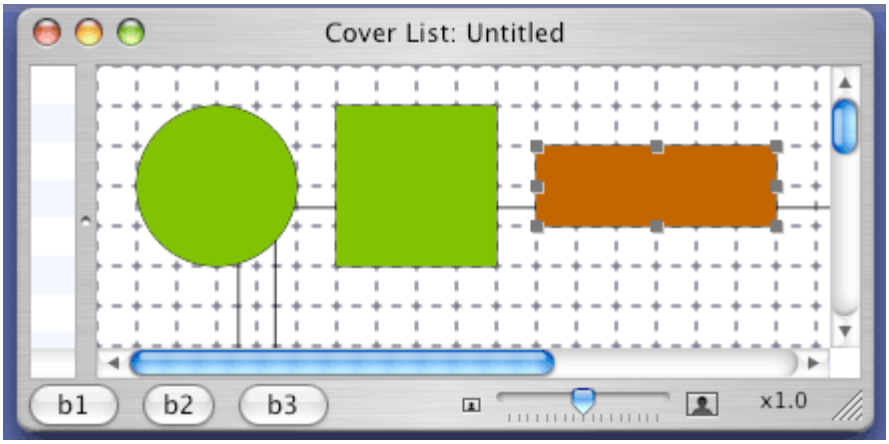
```
- (void)moveRight:(id)sender {
    float delta = snapToGrid ? gridSpacing : 1.0 ;
    [ theElements moveSelectedElements: NSMakePoint(delta, 0.0)];
    [ self setNeedsDisplay: YES ];
}
```

La quantità `delta` vale 1 in caso di griglia disattiva, è la spaziatura della griglia altrimenti.

Simile sorte la seguono i metodi associati agli spostamenti tramite frecce e tasto maiuscole. Non sapendo bene che quantità di spazio utilizzare nel caso di griglia attiva, ho deciso arbitrariamente per cinque.

```
- (void)moveRightAndModifySelection:(id)sender {
    float delta = snapToGrid ? (gridSpacing*5) : 10.0 ;
```

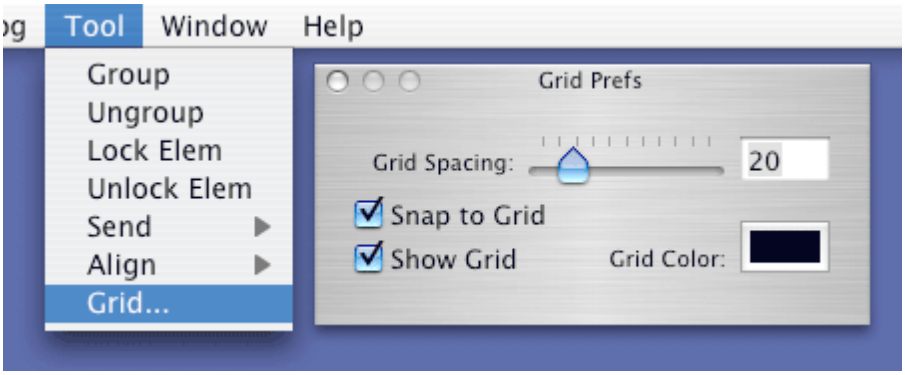
```
    [ theElements moveSelectedElements: NSMakePoint(delta, 0.0)];
    [ self setNeedsDisplay: YES ];
}
```



Con queste semplici correzioni, adesso anche gli spostamenti funzionano correttamente sulla griglia.

IL DIALOGO DELLA GRIGLIA

Non rimane a questo punto che permettere all'utente di determinare a proprio piacere i quattro parametri che caratterizzano la griglia: si tratta di impostare i valori delle quattro variabili d'istanza citate all'inizio del capitolo.



Per farlo, non c'è nulla di meglio di un dialogo chiamato da una voce di menu. Ecco quindi che modifico l'interfaccia aggiungendo la voce di menu `Grid...` che va ad aprire una nuova finestra `GridDialog`.

Nella finestra di dialogo ci sono due pulsanti di spunta per determinare se la griglia è visibile o meno, e se gli elementi vanno piazzati o meno sulla griglia. C'è poi una `NSColorWell` per determinare il colore con cui è disegnata la griglia, ed infine un campo testo ed uno slider per determinare la spaziatura. Lo slider ed il campo testo sono associati tra loro: l'utente può determinare la spaziatura scrivendo direttamente il numero nel capo testo, oppure manipolando lo slider.

Come ormai sono abituato a fare, aggiungo un metodo alla classe `AppDelegate`, che richiama l'istanza condivisa all'intera applicazione della classe controllore `GridDialogWinCtl`, che ho costruito associandola al file nib. La classe controllore

contiene i soliti metodi

```
+ ( id ) sharedGridDialogWinCtl ;
- (void)mainWindowChanged:(NSNotification *)notification ;
- (void)mainWindowResigned:(NSNotification *)notification ;
- (void) setMainWindow: (NSWindow *) mainWindow ;
```

per tenere traccia di se e quale finestra in prima piano contiene una coverView.

Le variabili d'istanza sono gli outlet verso gli elementi del dialogo ed appunto il puntatore alla coverView.

Poiché ogni coverView ha una caratterizzazione della griglia differente dalle altre coverView, è bene sistemare gli elementi del dialogo in accordo a tali variabili. È compito del metodo `setMainWindow:`

```
- (void)
setMainWindow: (NSWindow *) mainWindow
{
    NSWindowController *controller = [mainWindow windowController];
    // se c'e' una finestra ed e' del tipo giusto
    if (controller && [controller isKindOfClass:[CoverWinCtl class]])
    {
        theDrawView = [(CoverWinCtl *)controller cvrView];
        [ gridColor setColor: [ theDrawView gridLineColor]];
         [ gridSnap setState: ([ theDrawView snapToGrid] == YES ? NSOnState
: NSOffState)];
        [ sliderSpacing setIntValue: [theDrawView gridSpacing ] ];
        [ gridSpacing setIntValue: [theDrawView gridSpacing ] ];
         [ showGrid setState: ([ theDrawView showGrid] == YES ? NSOnState :
NSOffState)];
    }
    else
    {
        theDrawView = nil;
    }
}
```

Il metodo appunto recupera le informazioni dalla `CoverView` e predispone attraverso gli outlet l'aspetto dei pulsanti di spunta, dello slider, del testo e della `NSColorWell`.

Viceversa, ad ogni operazione su un elemento del dialogo, deve corrispondere ad una variazione dell'aspetto della griglia. I pulsanti di spunta ed il controllo `NSColorWell` chiamano direttamente il metodo seguente:

```
- (IBAction)updateGrid:(id)sender
{
    if ( theDrawView == nil )
        return ;
    [ theDrawView setGridLineColor: [gridColor color ]];
    [ theDrawView setSnapToGrid:     ([gridSnap state] == NSOffState ? NO :
YES) ] ;
    [ theDrawView setGridSpacing: [gridSpacing intValue ] ];
    [ theDrawView setShowGrid:         ([showGrid state] == NSOffState ? NO
: YES) ] ;
    [ theDrawView setNeedsDisplay: YES ];
}
```

Senza stare a farsi troppe domande, il metodo verificare che ci sia una `CoverView` in primo piano, e poi modifica brutalmente tutte le variabili d'istanza relative alla griglia con i valori ottenuti dagli outlet. Poi, forza un ridisegno della vista per modificare l'aspetto della griglia in accordo coi nuovi valori.

Per lo slider ed il campo testo c'è bisogno di un passaggio in più, in quanto occorre anche aggiustare l'altro controllo. Allo slider, ad esempio, è associato il metodo:

```
- (IBAction)gridSpacingFromSlider:(id)sender
{
    int sliderValue = [ sender intValue ] ;
    [ gridSpacing setIntValue: sliderValue ];
    [ self updateGrid: sender ];
}
```

Dopo aver recuperato il valore dello slider, lo impone al campo testo; poi, ricade nel metodo `updateGrid` visto sopra.

Assolutamente simmetrico il metodo associato al campo testo:

```
- (IBAction)gridSpacingFromText:(id)sender
{
    int sliderValue = [ sender intValue ] ;
    [ sliderSpacing setIntValue: sliderValue ];
    [ self updateGrid: sender ];
}
```

che svolge le stesse operazioni, ma con i fattori invertiti.

Capitolo 048 - Cancella tutto, anzi no

Come spesso accade quando si è messo mano ad un lavoro in tempi successivi e distanti, ci sono molti e diversi argomenti, più o meno compiuti, a far parte di questo capitolo. Diciamo che fondamentalmente si parla del meccanismo dello *Undo* (Annulla/Ripristina), che tanto bene ci ha fatto, ci fa e ci farà nell'uso normale del computer. Tuttavia, si parte da tutta altra parte, ed alla fine ci saranno altri argomenti piccoli e sparsi.

Documentazione Apple

Primo inserimento: 29 luglio 2004

Cancellare elementi

Dopo che nei capitoli precedenti mi ero occupato di spostare e ridimensionare elementi, ho pensato anche di aggiungere la possibilità di eliminarli. Per compiere questa operazione, l'utente ha a disposizione due possibilità: selezionare una voce di menu o premere qualche tasto significativo sulla tastiera (ad esempio, il tasto di cancellazione). Per la voce di menu, uso Interface Builder per associare alla voce *Cancella* o *Clear* del meni *Edit* il metodo `clear` della CoverView. Per quanto riguarda la tastiera, ricorrendo ancora una volta ai meccanismi esoterici descritti nel capitolo 45, il compito è svolto da due metodi specifici, qui di seguito rappresentati assieme al metodo `clear`.

```
- (void)deleteForward:(id)sender {
    [self clear:sender];
}

- (void)deleteBackward:(id)sender {
    [self clear:sender];
}

- (void) clear: (id)sender
{
    NSMutableArray * selArr = [ theElements getSelectedGraphics ];
    int                numElem = [ selArr count ];
    if ( numElem == 0 )
        return ;
    [self deleteElem:sender];
}
```

I metodi `deleteForward` e `deleteBackward` dovrebbero corrispondere ai tasti di cancellazione all'indietro ed in avanti, presenti separatamente sulle tastiere estese (generalmente il tasto di cancellazione in avanti non è presente sui portatili, come accade a me). Entrambi si limitano ad invocare il metodo `clear`. A sua volta `clear`, piuttosto che sporcarsi le mani, verifica che ci siano degli elementi selezionati e, nel caso, chiama un ulteriore metodo `deleteElem`. Anche questo metodo non è troppo sicuro di sé, e quindi mostra un bel dialogo preavvertendo l'utente che sta per cancellare

inesorabilmente alcuni elementi, richiedendone l'approvazione.

```
- (void)
deleteElem:(id)sender
{
    NSAlert *alert = [[[NSAlert alloc] init] autorelease];
    [alert addButtonWithTitle:@"OK"];
    [alert addButtonWithTitle:@"Cancel"];
    [alert setMessageText:@"Delete the selected elements?"];
    [alert setInformativeText:@"Deleted elements cannot be restored."];
    [alert setAlertStyle:NSWarningAlertStyle];
    [alert beginSheetModalForWindow:[winCtl window]
            modalDelegate:self
            didEndSelector:@selector(alertDidEnd:returnCode:contextInfo:)
            contextInfo:nil];
}
```

Non c'è nulla di particolare qui, si tratta del codice standard per gestire un normale Alert. Se l'utente non si è ancora stancato di tutta questa trafila, e dice di proseguire con la cancellazione, tocca al metodo indicato in precedenza tramite la direttiva `@selector`:

```
- (void)
alertDidEnd:(NSAlert *)alert
    returnCode:(int)returnCode
    contextInfo:(void *)contextInfo
{
    if (returnCode == NSAlertFirstButtonReturn)
        [ self deleteSelectedElements ];
}
```

Finalmente ci siamo: è chiamato il metodo `deleteSelectedElements`, che ci libera dalla presenza degli elementi correntemente selezionati:

```
- (void)
deleteSelectedElements
{
    int        i , numElem ;
    CCE_BasicForm * elem ;
    // ciclo su tutti gli elementi del gruppo
    numElem = [ [ theElements elemArray] count ];
    for ( i = (numElem-1) ; i >= 0 ; i -- )
    {
        elem = [ [ theElements elemArray] objectAtIndex: i ];
        if ( [ elem cceIsSelected ] )
        {
            [ theElements remElemAtIndex: i ];
        }
    }
    [ self setNeedsDisplay: YES ];
}
```

Scrivere codice ad oggetti è come l'arte di procrastinare. Il lavoro sporco è svolto ancora una volta altrove, e precisamente dal metodo `remElemAtIndex`, proprio della classe `CCE_ElemGroup`.

```
- (void) remElemAtIndex: (int) i
{
```

```
    // eseguo la cancellazione
    [ elemArray removeObjectAtIndex: i ];
    [ ownerView setNeedsDisplay: YES ];
}
```

In un modo o nell'altro, sono alla fine. Al termine di questa catena di metodi, si arriva all'eliminazione degli elementi della copertina. Ottimo. Ma, come il dialogo avvertiva, non c'è modo di tornare indietro. O meglio, per tornare indietro, occorre realizzare un meccanismo di Annullamento e Ripristino, in una sola parola, *Undo*.

ANNULLA E RIPRISTINA

Il meccanismo di Undo è uno dei motivi per cui si ama un computer. Scrivi una lettera, sbagli qualcosa, usi Undo e torni indietro. Sono finiti i tempi in cui si potevano commettere danni irreparabili con poco: adesso c'è sempre la possibilità di tornare sui propri passi (fosse così anche nella vita reale...). Di più, all'inizio si poteva tornare indietro di un solo passo, ma poi sempre più programmi realizzano un Undo multi-livello, attraverso il quale tornare indietro di più passi, molti, infiniti. Che bello. C'è un però. Però i programmatori dovevano diventare matti per realizzare tutto ciò; ad ogni possibile operazione dell'utente, il programmatore paranoico si deve chiedere se si può tornare indietro, e se sì, come. Ed una volta capito il come, prima di fare qualcosa, bisogna salvare lo stato corrente per essere sicuri di poterlo ripristinare. Insomma, uno sforzo titanico. Ora, con Cocoa, non è che non si debba essere paranoici. Prima che eseguire qualsiasi cambiamento, occorre sempre chiedersi se vale la pena di tornare indietro, e come. Però Cocoa mette a disposizione alcuni strumenti che semplificano la vita del programmatore (in ciò, va detto, aiutato dalla natura ad oggetti del programma).

In effetti, ogni modifica dell'utente può essere ricondotta alla modifica di uno stato di un oggetto. Quindi, prima di eseguire questa modifica, basta conservare in luogo fresco ed asciutto l'oggetto nel suo stato corrente. Meglio ancora, si conserva da qualche parte una serie di istruzioni che permettono di riportare l'oggetto allo stato corrente. Se è una macchinina-oggetto è di colore rosso, ed il bambino-utente la vuole gialla, per realizzare il meccanismo di Undo è sufficiente ricordare l'oggetto (o meglio, sapere qual è l'oggetto interessato), il metodo di colorazione, ed il parametro del metodo (rosso). Se una volta colorata di giallo, il bambino non gradisce, basta ripigliare l'oggetto, ed invocare il metodo (di colorazione) con il parametro conservato (rosso). Facile.

Bene, vediamo cosa Cocoa mette a disposizione. Attraverso la classe `NSUndoManager` è disponibile una zona dove conservare la triade oggetto-metodo-parametri. Supponiamo di voler realizzare il meccanismo di Undo per la cancellazione di un elemento: devo riscrivere il metodo `remElemAtIndex` per memorizzare le informazioni richieste.

```
- (void) remElemAtIndex: (int) i
{
    // gestione dello undo per aggiunta/rimozione elementi
      [[[self undoManager] prepareWithInvocationTarget: self] addElem: [
elemArray objectAtIndex: i ] atIndex: i ];
    // eseguo la cancellazione
    [ elemArray removeObjectAtIndex: i ];
    [ ownerView setNeedsDisplay: YES ];
}
```

Ho aggiunto una riga, piuttosto densa. Per prima cosa, ricavo una istanza della classe `NSUndoManager`. Poiché oggetti di tipo Document o View già la possiedono, mi limito a recuperarne una già disponibile (la cosa è un po' più complicata del previsto, ma lo spiego dopo). Poi predispongo le cose: dico di preparare un blocco di Undo che si riferisce all'oggetto `self` (nel caso, il vettore `theElements` che contiene tutti gli elementi della CoverView), e che per rimediare ad una cancellazione di un elemento, occorre utilizzare il metodo `addElem:atIndex:` (in altre parole, un metodo per aggiungere un elemento), al quale fornire come parametri l'elemento stesso in via di cancellazione e la sua posizione all'interno del vettore. Se l'utente, dopo la cancellazione, cambia idea e sceglie la voce di menù *Undo*, Cocoa esegue per noi qualcosa di simile alla seguente istruzione:

```
[ theElements addElem: <elemento> atIndex: <indice, i> ];
```

riportando la situazione al punto di prima.

Non ho ancora descritto iL metodo `addElem:atIndex:`, ma non credo vi spaventerà; del resto, è simile al vecchio `addElem:`, che aggiungeva brutalmente l'elemento all'inizio del vettore:

```
- (void) addElem: (CCE_BasicForm *) elem atIndex: (int) i
{
    // gestione dello undo per aggiunta/rimozione elementi
     [[[self undoManager] prepareWithInvocationTarget:self] remElemAtIndex:
i ];
    // inserisco davanti a tutto
    [ elemArray insertObject: elem atIndex:i];
    [ ownerView setNeedsDisplay: YES ];
}
```

Utilizzare il vecchio metodo, senza la possibilità di indicare la posizione di inserimento, era certamente possibile, ma avrebbe snaturato l'ordine di rappresentazione degli elementi, provocando una non completa uguaglianza tra lo stato previsto e quello ottenuto.

Qui si trova anche l'altra faccia del meccanismo di Undo, il *Redo*. Il nostro indeciso utente può alla fine decidere che in effetti l'elemento va veramente eliminato dalla faccia della terra; piuttosto che cancellarlo un'altra volta, può utilizzare il comando di menu *Redo* o *Ripristina* (anche questo diventato indispensabile nelle moderne applicazioni). In effetti, il contrario di aggiungere un elemento, è di rimuoverlo; per farlo, si usa ancora il metodo `remElemAtIndex`. Ma come fa Cocoa a capire che questa volta non si tratta di una operazione di Undo ma di Redo? Per capirlo, bisogna parlare di *pile* (intendendo ammassi di cose, dove le cose si buttano una sopra l'altra, e si tolgono a partire dall'ultima che è stata aggiunta).

PILE DI INFINITI RIPENSAMENTI

Le moderne applicazioni hanno multipli livelli di Undo e di Redo, ovvero, è possibile annullare una sequenza di operazioni, o ripristinare tutte o sola alcune delle operazioni annullate. Per capire come funziona questo fare e disfare a mo' di tela di Penelope, faccio il seguente modello concettuale. Ogni operazione consiste in una tripletta di

informazioni: l'indicazione di un oggetto, di un metodo da eseguire su questo oggetto, e dai parametri del metodo. Ogni operazione possiede (idealmente) una operazione complementare, tale per cui eseguendo in successione una operazione ela sua complementare, non si hanno modifiche allo stato generale dell'applicazione.

Un'applicazione con interfaccia utente possiede il famoso *loop degli eventi*, ovvero ogni applicazione che deve interagire con un utente generalmente è lì che fa nulla, aspettando qualche comando da parte dell'utente. Come il comando arriva, sono eseguite una serie di operazioni (che producono determinati risultati), che terminano con l'applicazione di nuovo in attesa di comandi da parte dell'utente. Un ciclo (o loop) degli eventi è ciò che accade tra un comando ed il successivo. Man mano che l'utente gioca con una applicazione, esegue dei comandi che si traducono in una serie di operazioni. Alcune di queste operazioni sono reversibili, e quindi un buon programmatore ha inserito istruzioni del tipo `prepareWithInvocationTarget:` con opportuna parametrizzazione. Cocoa mette via ordinatamente le corrispondenti triplette oggetto-metodo-parametri, inserendole sopra una pila. Questa è la pila degli Undo, e ad ogni ciclo degli eventi alla pila si aggiungono un po' di triplette. C'è anche un'altra pila, la pila dei Redo, che al momento è vuota.

Arriva ciò che tutti stiamo aspettando, ovvero l'utente che esegue un Undo. Allora il programma piglia il blocco di triplette che si trova in cima alla pila, corrispondenti alle operazioni eseguite nell'ultimo ciclo degli eventi, ed esegue le operazioni indicate. Se il programmatore non ha fatto sciocchezze, la sequenza delle istruzioni dovrebbe riportare l'applicazione nello stato precedente all'ultimo ciclo degli eventi, realizzando in tutto e per tutto un comando di Undo.

Ora, eseguendo le operazioni descritte dalle triplette, Cocoa può imbattersi (e generalmente lo fa) in istruzioni del tipo `prepareWithInvocationTarget:`. Ma queste istruzioni specificano operazioni che dovrebbero essere eseguite per riportare l'applicazione in uno stato prima dell'operazione di Undo: Cocoa piglia quindi il blocco delle triplette raccolte e le inserisce all'interno della pila dei Redo. In pratica, in seguito ad un comando di Undo, Cocoa piglia le triplette dalla pila degli Undo, le esegue, riceve nuove triplette (che se il programmatore è onesto, derivano dalle operazioni complementari di quelle appena eseguite) e le inserisce nella pila dei Redo.

Cosa succede se adesso l'utente comanda Redo? Cocoa piglia le triplette dalla pila dei Redo, ed esegue le operazioni indicate. Così facendo, si imbatte in nuove triplette, da conservare nella pila degli Undo (ancora una volta, se il programmatore è stato onesto, dovrebbe ritrovare tali e quali le triplette che aveva eliminato dalla pila degli Undo all'ultima esecuzione del comando corrispondente).

C'è inoltre da tenere presente che le pile non contengono solo un blocco di triplette alla volta. Cocoa fornisce di default una pila infinita per Undo e Redo: si può teoricamente conservare nella pila degli Undo (ed anche in quella dei Redo) un numero infinito di blocchi di triplette, in modo da tornare indefinitamente sui propri passi. In pratica la grandezza delle pile è limitata dalla memoria disponibile ma soprattutto dalla natura delle applicazioni, per le quali ha senso mantenere un numero sensato di blocchi sulla pila.

UNDO NELLA PRATICA

Le operazioni che al momento considero reversibili non sono poi tantissime: aggiunta e cancellazione di un elemento; spostamento di un elemento; ridimensionamento di un elemento. Ho già descritto come gestire la cancellazione e il ripristino di un elemento. C'è una piccola cosa da notare: il codice effettivo di `addElem:` è il seguente:

```
- (void) addElem: (CCE_BasicForm *) elem atIndex: (int) i
{
    // gestione dello undo per aggiunta/rimozione elementi
    [[self undoManager] setActionName: @"ElemGroup -> addElem"];
     [[[self undoManager] prepareWithInvocationTarget:self] remElemAtIndex:
i ];
    // inserisco davanti a tutto
    [ elemArray insertObject: elem atIndex:i];
    [ ownerView setNeedsDisplay: YES ];
}
```

Ho aggiunto una istruzione (non guardate la stringa, è per il mio debugging) per completare acconciamente la voce di menu. Ad esempio, mentre sto scrivendo questo testo con TextEdit, la voce di menù è in effetti *Annulla Inserimento* piuttosto che un semplice *Annulla*, proprio per evidenziare che l'operazione che sarebbe annullata è l'inserimento di testo. Se impartissi il comando di *Annulla*, il comando di *Ripristino* diverrebbe attivo, ma con la dicitura *Ripristina Inserimento*. Le stringhe "Annulla" e "Ripristina" sono messe da Cocoa, con il metodo `setActionName:` il programmatore le completa.

Per la gestione di Undo/Redo sugli spostamenti, occorre modificare i metodi `moveMeIfSelectBy:`

```
- (void)
moveMeIfSelectBy: (NSPoint) distance
{
    if ( (! cceIsLocked) && cceIsSelected )
    {
        // gestione dello undo del movimento
                    [[self undoManager] setActionName: @"BasicForm ->
moveMeIfSelectBy"];
             [[[self undoManager] prepareWithInvocationTarget: [ownerView
theElements]]
            moveElem: self by: NSMakePoint(-distance.x, -distance.y)];
        // effettuo il movimento vero e proprio
                [ self setElemLimit: NSOffsetRect(elemLimit, distance.x,
distance.y)];
               [ self buildMeWithStartPt: NSMakePoint(elemLimit.origin.x,
elemLimit.origin.y)
                                  andPt: NSMakePoint( elemLimit.origin.x +
elemLimit.size.width,
                                                      elemLimit.origin.y +
elemLimit.size.height) ];
    }
}
```

Ogni realizzazione di questo metodo richiede le due istruzioni per la gestione dello Undo, una per impostare la stringa, l'altra per rimediare allo spostamento. Ho costruito un nuovo metodo `moveElem:by:`, che riceve come parametro lo spostamento opposto a quello richiesto.

Ugualmente, all'interno del metodo `handleResize:...`, prima di ogni altra operazione, ho aggiunto le due istruzioni:

```
        // mi preparo lo undo
    [[self undoManager] setActionName: @"BasicForm -> handleResize"];
          [[[self   undoManager]   prepareWithInvocationTarget:   [ownerView
theElements]]
            resizeElem: self withStartPt: NSMakePoint(elemLimit.origin.x,
elemLimit.origin.y)
           andPt: NSMakePoint( elemLimit.origin.x + elemLimit.size.width,
                                                          elemLimit.origin.y  +
elemLimit.size.height)];
```

In caso di Undo, verrà eseguito il (nuovo) metodo `resizeElem:withStartPt:andPt:`, che ha ricevuto come parametri le dimensioni correnti dell'oggetto, prima del ridimensionamento.

Ci sono due avvertenze: la prima è che, come al solito, la classe `CCE_Line`, costruendo le linee in modo diverso da quanto accade con i rettangoli ed i cerchi, ha una versione leggermente diversa di queste istruzioni (nulla di trascendente, tuttavia). La seconda avvertenza è in realtà la spiegazione dei nuovi metodi della classe `CCE_ElemGroup`:

```
- (void)
moveElem: (CCE_BasicForm*) elem by: (NSPoint) distance
{
    [ elem moveMeBy: distance ];
}

- (void)
resizeElem:  (CCE_BasicForm*)  elem  withStartPt:  (NSPoint)  stPt  andPt:
(NSPoint) endPt
{
    [ elem resizeMeWith: stPt andPt: endPt ];
}
```

La domanda che mi pongo (precedendo o voi miei venticinque lettori) riguarda il motivo per cui ho voluto usare `theElements` come target delle operazioni di Undo, piuttosto che direttamente gli oggetti coinvolti. Mi spiego meglio: perché il codice seguente non funziona come voglio?

```
- (void)
moveMeIfSelectBy: (NSPoint) distance
{
    if ( (! cceIsLocked) && cceIsSelected )
    {
        // gestione dello undo del movimento
                    [[self   undoManager]   setActionName:   @"BasicForm   ->
moveMeIfSelectBy"];
        [[[self undoManager] prepareWithInvocationTarget: self]
            moveMeBy: NSMakePoint(-distance.x, -distance.y)];
        // effettuo il movimento vero e proprio
               [  self  setElemLimit:  NSOffsetRect(elemLimit,  distance.x,
distance.y)];
              [  self  buildMeWithStartPt:  NSMakePoint(elemLimit.origin.x,
elemLimit.origin.y)
                                  andPt:  NSMakePoint(  elemLimit.origin.x  +
elemLimit.size.width,
```

```
                                                                elemLimit.origin.y  +
elemLimit.size.height) ];
    }
}
```

Questa seconda versione è certamente più efficiente della prima, in quanto evita un passaggio. Però mi rende confusa la gestione multi-livello degli Undo e dei Redo, per cui talvolta rimangono nelle pile degli Undo e dei Redo dei blocchi di operazioni che sembrano non avere alcun effetto. Confesso che la situazione mi è poco chiara, e ho già speso abbastanza tempo cercando di capire senza concludere nulla. Riferendo invece ogni operazione di Undo all'oggetto `theElements`, le cose cominciano a funzionare.

Ho un ulteriore paio di commenti sul metodo `resizeMeWith`; non è altro che `buildMeWithStartPt` con due istruzioni per la gestione dello Undo in testa. Ho preferito un metodo separato per il ridimensionamento dove esplicitare le due istruzioni per lo Undo, piuttosto che inserirle direttamente all'interno di `buildMeWithStartPt`. Il fatto è che il metodo `buildMeWithStartPt` è chiamato frequentemente per svolgere operazioni continue (spostamenti e ridimensionamenti), e tutti questi messaggi sarebbero stati inutili (tranne l'ultimo) ed anzi avrebbero riempito le pile. In secondo luogo, mi rendo conto che le operazioni di Undo sono chiamate due volte con la stessa parametrizzazione. La prima volta all'interno del metodo `handleResize:...`, e la seconda qui. Il motivo è che questo stesso metodo è ovviamente chiamato anche in caso di Redo (senza passare per `handleResize:...`); senza la coppia in `handleResize:` perdo lo Undo, senza questa coppia perdo il Redo.

```
- (void)
resizeMeWith: (NSPoint) stPt andPt: (NSPoint) endPt
{
    // gestione dello undo del movimento
    [[self undoManager] setActionName: @"BasicForm -> resizeMeWith"];
          [[[self   undoManager]   prepareWithInvocationTarget:   [ownerView
theElements]]
             resizeElem: self withStartPt: NSMakePoint(elemLimit.origin.x,
elemLimit.origin.y)
            andPt: NSMakePoint( elemLimit.origin.x + elemLimit.size.width,
                                                                elemLimit.origin.y  +
elemLimit.size.height)];
    // metto a posto il rettangolo che lo contiene
    [self buildMeWithStartPt: stPt andPt: endPt ];
}
```

TANTE PICCOLE COSE

Ci sono tante altre piccole cose per completare la descrizione delle modifiche al codice.

Ho aggiunto due variabili d'istanza (e metodi accessor) alla classe `CCE_BasicForm`. La prima è un riferimento alla CoverView in cui l'elemento grafico è contenuto. L'uso di questa variabile comporta alcune operazioni in più in sede di costruzione dell'oggetto, che si ripagano in un accesso più veloce alla CoverView quando si rende necessario. Ad esempio, il metodo `undoManager`, che serve a recuperare l'istanza condivisa della classe `NSUndoManager` per la gestione di Undo e Redo si scrive facilmente:

```
- (NSUndoManager *)undoManager
{
    return [[[ownerView winCtl] document] undoManager];
}
```

Per lo stesso motivo sono cambiati i metodi come `handleCreation:` che in precedenza richiedevano la view di riferimento come parametro.

L'altra variabile d'istanza è un numero intero cui assegno un valore crescente per ogni elemento; mi serve come identificatore dell'oggetto quando faccio del debugging (ad esempio, so che la linea di codice 1 è quella disegnata per prima, eccetera). Tutto ciò comporta una modifica a tutte le funzioni `printMeWithLevel`.

Ho spostato l'inviocazione al metodo `setNeedsDisplay` il più in basso possibile: in precedenza era la CoverView che forzava il ridisegno della vista perché presumeva che fosse qualcosa nel contenuto della vista. Per quanto possibile, ho spostato questa istruzione all'interno degli elementi, essendo loro stessi i migliori conoscitori della necessità o meno di un ridisegno. In particolare ciò ha richiesto una nuova versione del metodo accessor `setCceIsSelected`, che è diventato:

```
- (void)setCceIsSelected:(BOOL)newCceIsSelected {
    if (cceIsSelected != newCceIsSelected) {
        cceIsSelected = newCceIsSelected;
        [ [ self ownerView ] setNeedsDisplay: YES ];
    }
}
```

Se lo stato di selezionato di un elemento è stato modificato, certo ci sarà da ridisegnare qualcosa (gli handle, ad esempio).

Ho cambiato il comportamento della palette che contiene gli strumenti per il disegno della copertina. La versione precedente manteneva selezionato l'elemento anche dopo il completamento del tracciamento dell'elemento. Ho preferito fare in modo che, al termine della creazione di un nuovo elemento, lo strumento selezionato tornasse ad essere la freccia. Allo scopo, c'è un metodo della classe `CoverPalWinCtl` dall'affascinante nome di `resetTool`:

```
- (void)
resetTool
{
    [toolMatrix selectCellWithTag: 0] ;
     currCCE = nil ;
}
```

Il metodo è invocato da `mouseDown:` della classe `CoverView` al termine della creazione di un elemento.

Tutte le istruzioni di disegno della base della copertina, prima raccolte all'interno del metodo `initWithFrame` di `CoverView` sono ora raccolte in un metodo a parte. Saranno quanto prima sostituite da qualcos'altro.

Infine, per evitare troppa confusione nella selezione e nello spostamento degli oggetti, all'interno del metodo `handleMoveAll` di `CoverView` verifico se il nuovo punto del

mouse è realmente nuovo, cioé se il puntatore del mouse si è spostato. Solo in tal caso scateno la proceudra di spostamento degli elementi selezionati.

Capitolo 049 - Scrivere in una view

Riprendo in mano la classe `CCE_Text` per disegnare stringhe di testo all'interno di una view.

Sorgenti: documentazione Apple

Prima stesura: 6 agosto 2004

Così fan tutti

Quando si tratta di disegnare del testo all'interno di una finestra, normalmente si usa una variabile della classe `NSTextView`, che fornisce già pronte tutta una serie di funzionalità adatte alla maggior parte dei casi. Tuttavia, nel mio caso, un po' per snobismo, un po' per imparare meglio come funzionano le cose, provo a farne a meno. L'idea è di migliorare la classe `CCE_Text` in modo che sia in grado di rappresentare all'interno della CoverView testo di vario tipo. In particolare mi interessa riportare l'elenco dei file di un volume, come se fosse una `NSOutlineView` (ma con alcune interessanti differenze).

Per prima cosa, mi rinfresco le idee con un po' di teoria. L'architettura preposta alla rappresentazione del testo all'interno di Cocoa è composta da alcune classi, inquadrate all'interno del paradigma MVC (*Model*, *View*, *Controller*). La classe modello dovrebbe essere quella che contiene il modello dei dati: nel caso di testo, deve contenere appunto il testo, secondo una qualche rappresentazione (che si spera ricca, flessibile, facile da usare, eccetera). Questa classe si chiama `NSTextStorage`, ed è una sottoclasse di `NSMutableAttributedString`, a suo volta sottoclasse di `NSAttributedString` (che non è una sottoclasse di `NSString`!). `NSAttributedString` (per stringhe definite staticamente) e `NSMutableAttributedString` (per stringhe capaci di cambiare nel tempo) sono classi che gestiscono stringhe dotate di attributi; gli attributi sono il font, il colore, lo stile, il kerning, eccetera, ovvero tutti quegli attributi tipografici che siamo abituati ad utilizzare quando scriviamo del testo in bella copia. Le classi mettono a disposizione metodi per costruire queste stringhe a partire da normali stringhe, da file RFT e HTML e (a partire da Mac OS X 10.3) perfino documenti di Microsoft Word. Forniscono metodi per la manipolazione della stringa, degli attributi e di varia utilità. come esempio fornisco il gruppo di istruzioni seguente:

```
temp = [ [ NSMutableAttributedString alloc] initWithString: @"riga
uno\nriga due\nriga tre\nriga quattro\nriga cinque\nriga sei" ];
locFont = [ NSFont fontWithName: @"Arial" size: 12 ] ;
locFont = [ [NSFontManager sharedFontManager] convertFont: locFont
toHaveTrait: NSBoldFontMask];
[ temp addAttribute: NSFontAttributeName value: locFont range:
NSMakeRange(0, [ temp length] ) ];
```

Costruisco in primo luogo una `NSMutableAttributedString` partendo da una semplice stringa; costruisco poi una variabile di tipo `NSFont`, tale da specificare un font

Arial, di dimensione 12, cui aggiungo la specifica (*tratto*) di Grassetto; attribuisco infine questo font a tutta la stringa. Il metodo `addAttribute:value:range:` aggiunge infatti l'attributo di font specificato mediante il valore al range dei caratteri che va da zero alla intera lunghezza della stringa. Specificando un diverso range, potevo giocare sulla rappresentazione della stringa in vari modi divertenti.

Tornando alla classe `NSTextStorage`, questa classe aggiunge alla capacità di modellare il testo anche il collegamento con le sezioni View e Controller del paradigma.

La classe `NSLayoutManager` funziona appunto da classe Controller, che governa l'interazione tra la classe modello/sorgente del testo e le classi preposte alla visualizzazione di questi testi. La classe è responsabile della trasformazione dei caratteri in glifi; un carattere è la rappresentazione astratta di un carattere, mentre un glifo è la rappresentazione grafica di questo carattere. La lettera *p* è un carattere, il simbolo grafico '*p*' è un glifo; ovviamente, ad un carattere possono corrispondere diversi glifi, a seconda della rappresentazione (font, dimensione, eccetera).

La classe `NSTextContainer` regola infine la parte View del paradigma, e stabilisce come i caratteri (no!, i glifi) sono rappresentati a video. Stabilisce quindi la zona dello schermo dove andare a disegnare i vari glifi. Normalmente, questa classe è associata ad una NSTextView, con la quale collabora a rappresentare sullo schermo il testo.

È interessante notare la catena d'uso di queste classi. Per un dato testo (ad un esempio, una noiosa relazione sullo stato aziendale di venti pagine) esiste una sola classe `NSTextStorage`, che contiene il testo nudo e i vari attributi al testo (font, dimensioni, colori, eccetera). C'è poi in genere un solo `NSLayoutManager` che regola la visualizzazione; collegate a questa classe ci sono in genere più istanze di `NSTextContainer` che determinare le zone di rappresentazione del testo (le varie pagine, o le varie colonne della relazione). Infine, `NSTextContainer` si appoggia qualche altra classe per la visualizzazione vera e propria.

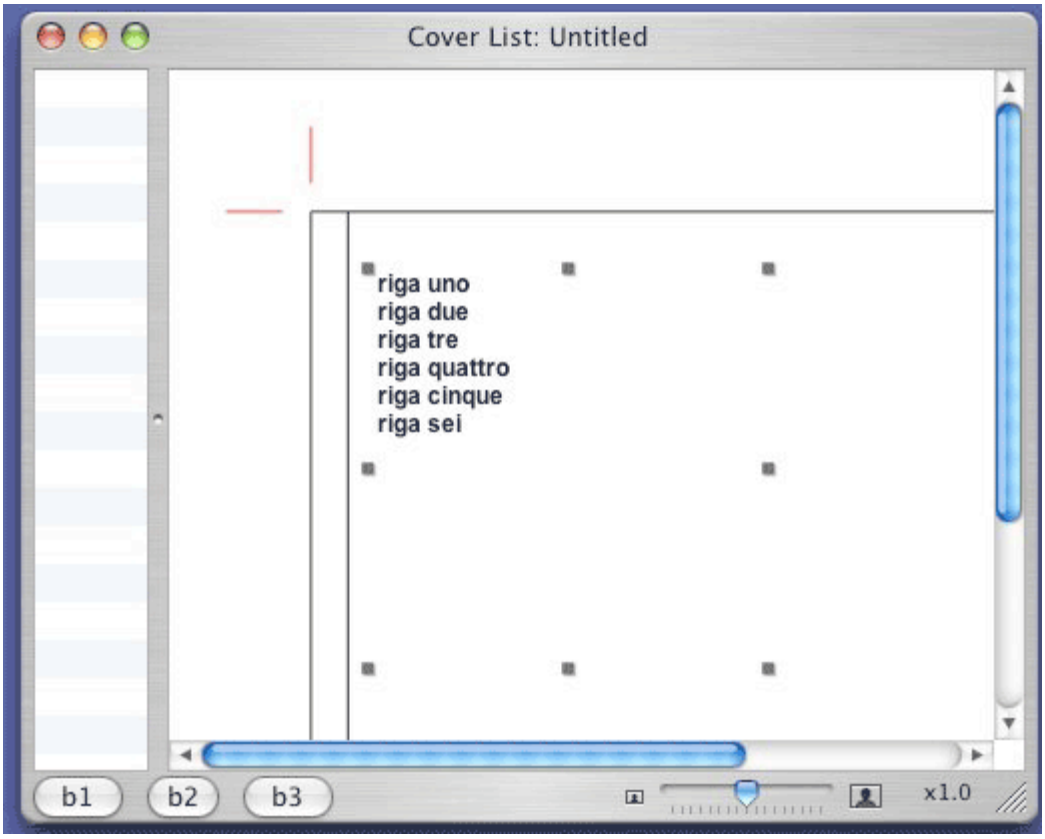
LA CLASSE CCE_TEXT

Avevo già scritto la classe `CCE_Text` in grado di gestire delle stringhe di testo; in effetti, la classe è già a buon punto, anche se la maggior parte delle istruzioni erano state scritte per mera copia da un esempio e non le avevo del tutto capite. Ne approfitto per ripulirne e cambiare qualcosa.

In primo luogo, modifico il metodo preposto all'inizializzazione.

```
- (id) initWithAttributedString: (NSAttributedString *) theText
        inRect: (NSRect) aRect
        andRotation: (float) angle
{
    self = [ super init ];
    if ( self )
    {
        NSTextStorage * loctextStorage ;
        NSLayoutManager *layoutManager;
        NSTextContainer *textContainer;
        // layout gestisce come il testo e' rappresentato
         loctextStorage = [[[NSTextStorage alloc] initWithAttributedString:
```

```
theText] autorelease ];
        layoutManager = [[[NSLayoutManager alloc] init] autorelease ];
        [loctextStorage addLayoutManager:layoutManager];
        // indica dove il testo e' rappresentato
        textContainer = [[[NSTextContainer alloc] init] autorelease ];
        // dovrei specificare un NSSize per il textcontainer
        // lo faro' piu' pardi...
        [layoutManager addTextContainer:textContainer];
        // assegno il textStorage
        [ self setTextStorage: loctextStorage ];
        // per ora me ne faccio nulla
        rotAngle = angle ;
        theTextCont = textContainer ;
        [ self setElemLimit: aRect ];
    }
    return ( self );
}
```

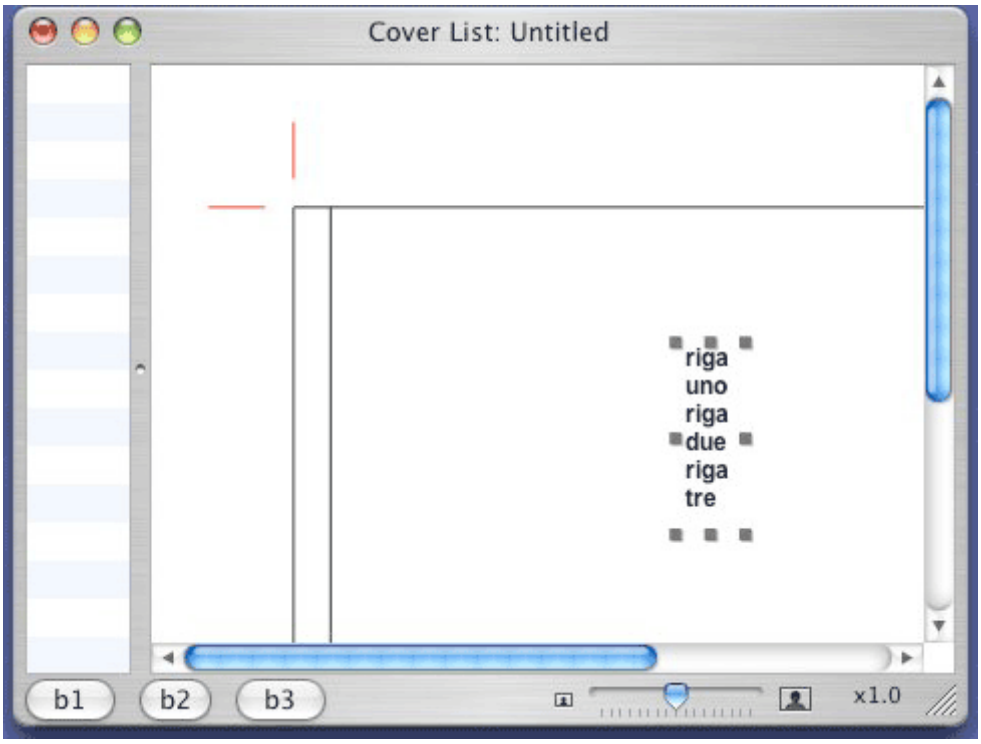


Parto dall'idea che la classe accetti come parametro di inizializzazione una `NSAttributedString`, costruisco in successione una `NSTextStorage`, una `NSLayoutManager` ed una `NSTextContainer` (una sola; per ora avrò tutto il testo racchiuso all'interno di un unico rettangolo). Ciascuna delle tre classi nasce `autorelease`, perché poi sono *retained* nel momento in cui sono associate alla classe collegata (`NSLayoutManager` da `NSTextStorage`, `NSTextContainer` da `NSLayoutManager`). Infine, conservo (e faccio retain) la classe `NSTextStorage` all'interno di una variabile d'istanza della classe `CCE_Text`. Per comodità (serve accedervi spesso per vari motivi) convervo anche un riferimento alla classe `NSTextContainer` (faccio una assegnazione diretta ad una variabile d'istanza, e quindi non aggiungo retain).

C'è solamente una importante istruzione che qui non è presente, in quanto nascosta all'interno del metodo `setElemLimit`, che ho riscritto per la classe `CCE_Text`.

Per la corretta rappresentazione del testo occorre infatti individuare il rettangolo all'interno del quale il testo è rappresentato. Ecco quindi la varsione specifica del metodo:

```
- (void)setElemLimit:(NSRect)newElemLimit {
elemLimit = newElemLimit;
        [ self buildHdlList ];
        [ theTextCont setContainerSize: elemLimit.size ];
}
```



Ora, grazie alle mie superbe capacità di programmatore (sto scherzando), l'attribuzione `setContainerSize:` è l'unica istruzione che mancava alla classe `CCE_Text` per partecipare con successo a tutte le operazioni di manipolazione del testo: selezione, spostamento, ridimensionamento. Provare per credere.

Inoltre, per semplificare (non che quello presente fosse sbagliato, anzi) il metodo di disegno, ho scritto:

```
- (void) drawElement: (NSRect) inRect
{
    // recupero il primo layout
      NSLayoutManager *layoutManager = [ [ textStorage layoutManagers ]
objectAtIndex: 0] ;
    // recupero il range del testo
        NSRange glyphRange = [layoutManager glyphRangeForTextContainer:
theTextCont ];
    // disegno del testo
          [ layoutManager drawGlyphsForGlyphRange:glyphRange atPoint:
elemLimit.origin ];
    [ self drawHandles ] ;
```

```
}
```

Qui recupero il `NSLayoutManager`, e faccio calcolare quanti e quali caratteri si possono rappresentare all'interno del `NSTextContainer` corrente. Ottengo così la variabile `glyphRange`, che dice (è un range, quindi posizione di partenza e numero di elementi) quali e quanti elementi sono rappresentabili. Quindi, visto che non devo fare altro, passo direttamente a disegnare i glifi a partire dal punto in alta a sinistra del rettangolo.

Rimane da vedere un esempio d'uso di questa classe, ad esempio attraverso il frammento che trovate all'interno del metodo `insertBasicDraw` della classe `CoverView`:

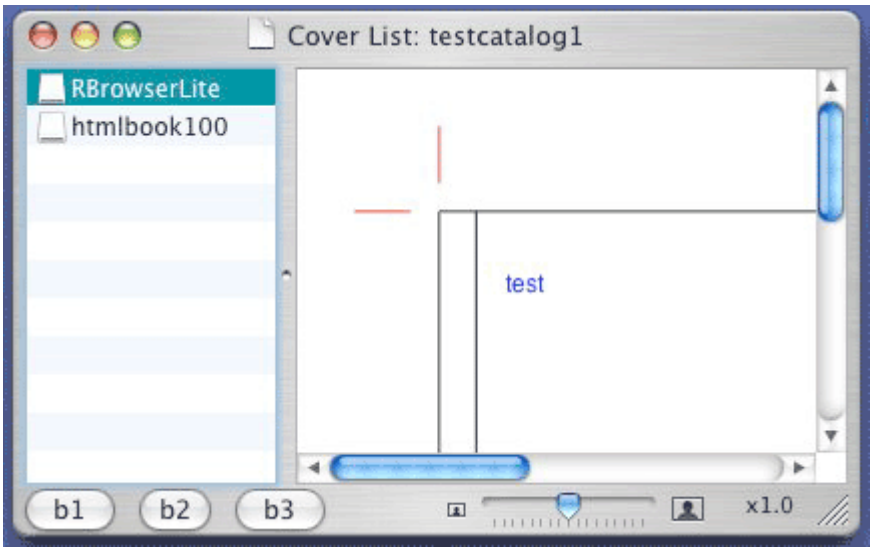
```
CCE_Text         * tmpText ;
NSFont             * locFont ;
NSMutableAttributedString * temp ;

temp = [ [ NSMutableAttributedString alloc] initWithString: @"riga
uno\nriga due\nriga tre\nriga quattro\nriga cinque\nriga sei" ];
locFont = [ NSFont fontWithName: @"Arial" size: 12 ] ;
locFont = [ [NSFontManager sharedFontManager] convertFont: locFont
toHaveTrait: NSBoldFontMask];
[ temp addAttribute: NSFontAttributeName value: locFont range:
NSMakeRange(0, [ temp length] ) ];
// calcolo il punto
x1 = STARTPOINT_X + CUTSIGN_DIM + CUTSIGN_DIST + CUTSIGN_DIM ;
y1 = STARTPOINT_Y + CUTSIGN_DIM + CUTSIGN_DIST + CUTSIGN_DIM ;
tmpText = [[ CCE_Text alloc] initWithAttributedString: temp
            inRect: NSMakeRect(CM2PX(x1), CM2PX(y1), 200, 200) andRotation:
0.0 ];
[ tmpText setOwnerView: self ];
```

Le prime quattro istruzioni le ho già spiegate in precedenza, mentre le successive due sono piuttosto banali; finalmente c'è la costruzione di una istanza di `CCE_Text`, passando come parametri la `NSAttributedString` (si, lo so, è in realtà `NSMutableAttributedString`, ma funziona lo stesso) ed il rettangolo di iniziale rappresentazione. L'angolo di rotazione è al momento disattivo (non perché non funzioni, ma perché manda in malora la selezione e lo spostamento degli oggetti... penso che riprenderò la questione quanto prima).

LA LISTA DEI FILE

Ora che so come scrivere del testo all'interno della CoverView, voglio utilizzare la classe `CCE_Text` per riportare la lista dei file, come se fosse una `NSOutlineView`. Al contrario della lista, il testo non è manipolabile, e sono presenti solo le voci che, nella finestra di catalogo, riportano il box di spunta `is2Print` a Vero (la cosa era stata predisposta nel capitolo 34).

Per fare questo, ho bisogno di un meccanismo per costruire una `NSMutableAttributedString` contenente tutta la lista di interesse, e di una strategia di aggiornamento del contenuto della finestra quando l'utente cambia la selezione del volume. In realtà il tutto confluisce nella modifica del metodo `tableViewSelectionDidChange` della classe sorgente di dati `CatDataSrc`.



Come è già stato spiegato, la lista dai volumi della finestra della copertina (un oggetto della classe NSTableView) utilizza la classe sorgente dei dati `CatDataSrc` per recuperare i dati da visualizzare. Utilizza poi la stessa classe (ma solo per comodità di localizzazione del codice) come classe delegata al trattamento delle operazioni scatenate dall'utente al cambio di selezione del volume.

Riscrivo quindi il metodo `tableViewSelectionDidChange:` per tenere conto del fatto che voglio aggiornare il contenuto della CoverView. Lo discuto in due momenti:

```
- (void)
tableViewSelectionDidChange:(NSNotification *)aNotification
{
    NSTextStorage * newFileList ;
    CCE_Text        * wxw ;
    int               elNum, jj, nn ;
    NSTableView       * ov = (NSTableView*)[ aNotification object ] ;
    // e' cambiata la selezione
    NSRange           myrange ;

    currentVol = [ ov selectedRow ] ;
    newFileList = [[NSTextStorage alloc] initWithString:@"test"];
    ...
    wxw = [ [[refDoc coverWin] cvrView] fileList];
    // imposto gli attributi per tutto il range
    myrange = NSMakeRange(0, [ newFileList length] ) ;
       [  newFileList  addAttribute:  NSForegroundColorAttributeName  value:
[NSColor blueColor] range: myrange ];
    [[ wxw textStorage] setAttributedString: newFileList ] ;
    [ [[refDoc coverWin] cvrView] setNeedsDisplay: YES ];
}
```

All'interno della classe `CoverView` ho definito una nuova variabile d'istanza, `fileList`, che punta ad una specifica istanza di un oggetto della classe `CCE_Text`, quella destinata appunto a contenere la lista dei file del volume. È utilizzata per poter rimpiazzare il contenuto corrente con un nuovo contenuto (al momento, la sola scritta "test", in un bel colore blu). A parte il lungo giro di messaggi necessario per raggiungere l'oggetto `CCE_Text`, non ci sono cose particolari da notare.

La parte più interessante è invece come ricavare la lista dei file. Ora, non esiste un

meccanismo esplicito per esaminare il contenuto della classe sorgente di dati. La classe `CatDataSrc` infatti realizza una serie di metodi che sono sfruttati da oggetti `NSOutlineView` per estrarre i dati. Il procedimento di estrazione è trasparente per il mio codice, ed è interamente gestito da Cocoa. Se quindi volessi recuperare la lista dei file presenti all'interno della sorgenti dati, dovrei scrivere nuovi metodi che gestiscano e regolino questo processo. In realtà, posso ricorrere ad un trucco interessante: simulo la presenza di una `NSOutlineView` ed utilizzo i metodi standard di interrogazione standard della sorgente dati.

A suo tempo, avevo fatto in modo che la stessa classe `CatDataSrc` potesse funzionare sia per la `NSOutlineView` della finestra con la lista dei volumi che per una `NSOutlineView` che volevo proditoriamente utilizzare all'interno della CoverView. Bastava distinguere il richiedente tramite l'attribuzione di un apposito tag alla `NSOutlineView`. Qui ricorro ad un trucco simile: decido che se il parametro NSOutlineView (indicante la tabella richiedente) è vuoto (*nil*), allora, procedo tranquillo come se la richiesta provenisse dalla `NSOutlineview` interna alla CoverView (che poi in realtà non esiste più, visto che la sto proprio rimpiazzando con un `CCE_Text`).

Per fare questo, rimpiazzo in ciascun metodo `outlineView:qualcosa:` una istruzione di controllo sul tag; è in pratica la prima istruzione di ciascun metodo:

```
- (int)
outlineView: (NSOutlineView *)outlineView
    numberOfChildrenOfItem:    (id)item
{
    VolInfo * localItem ;
    // se sto recuperando dati per la CoverCtlWin
          if   (   (outlineView   ==   nil)   ||   ([   outlineView   tag]   ==
TAG_CDCATDOC_FILELIST) )
    {
        if ( currentVol == -1 )
            return ( 0 ) ;
        if (item == nil)
            localItem = [ startPoint objectAtIndex: currentVol ];
        else localItem = item ;
    }
    else if ( [ outlineView tag] == TAG_CDCATDOC_FULLLIST )
    {
        /* eccetera */
    }
    /* eccetera */
}
```

Una volta messi a posto questi metodi, sono pronto per riempire gli spazi che prima avevo lasciato vuoti.

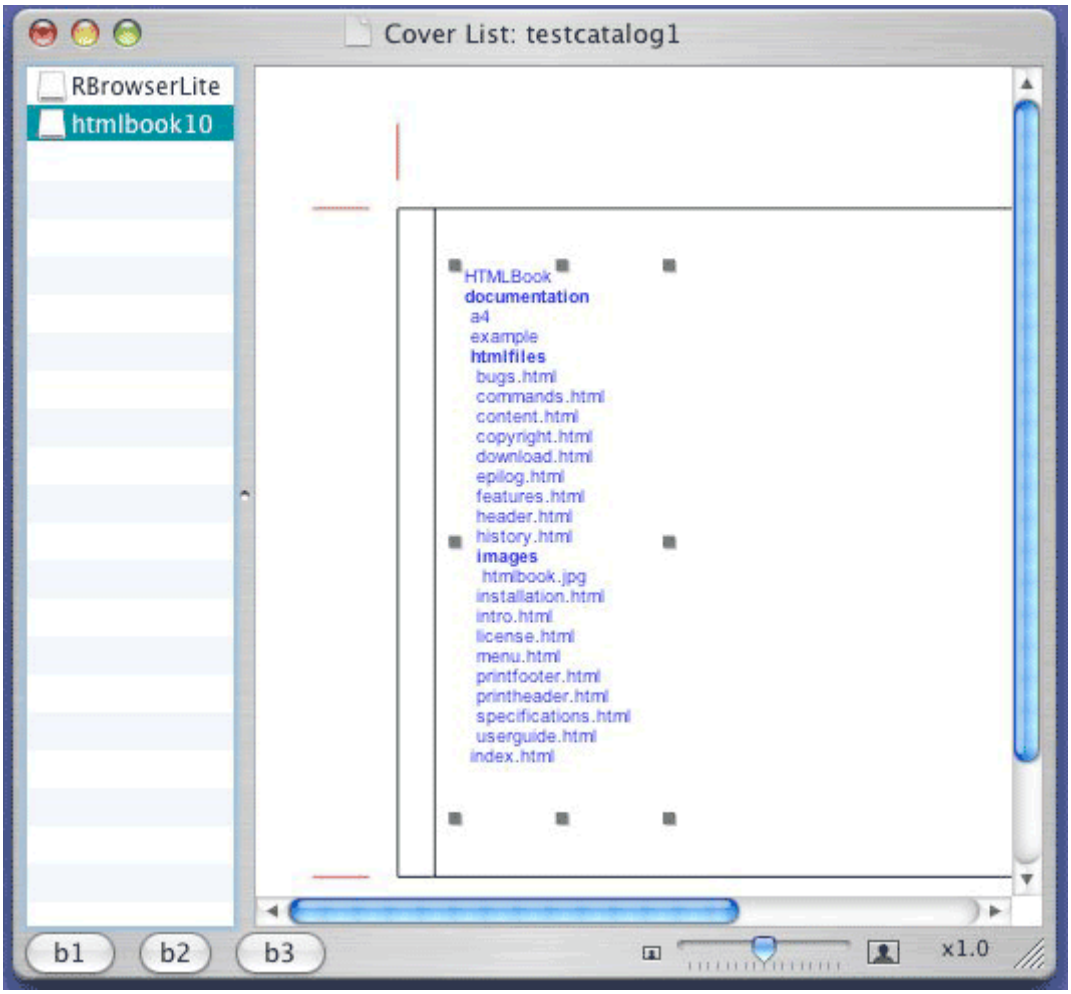
```
- (void)
tableViewSelectionDidChange:(NSNotification *)aNotification
{
    ...
    currentVol = [ ov selectedRow ] ;
    newFileList = [[NSTextStorage alloc] initWithString:@""];
    elNum = [ self outlineView: nil numberOfChildrenOfItem: nil ];
    for ( jj = 0 ; jj < elNum; jj ++ )
    {
        FileStruct * elem ;
```

```
        elem = [ self outlineView:     nil child: jj ofItem: nil ];
        nn = [ self outlineView: nil numberOfChildrenOfItem: elem ];
        // se non e' da stampare, salto
        if ( ! [ elem is2Print ] ) continue ;
        if (nn > 0 )
            addStringExpandItem (newFileList, self, elem, nn, 0 );
        else
        {
            terminalWithLevel( newFileList, [ elem fileName], 0, NO );
        }
    }
    wxw = [ [[refDoc coverWin] cvrView] fileList];
    ...
}
```

La prima nuova istruzione:

```
elNum = [ self outlineView: nil numberOfChildrenOfItem: nil ];
```

chiede in pratica quanti elementi di primo livello si trovano all'interno del volume selezionato. Entrando nel metodo `outlineView:numberOfChildrenOfItem:` con tutti i parametri a nil, questo restituisce alla fine il numero degli elementi selezionati.



Proseguendo nella lista delle istruzioni, eseguo un ciclo per tutti gli elementi di primo livello; li estraggo uno ad uno col metodo `outlineView:child:ofItem:` e poi verifico se hanno a loro volta figli (cioé, se sono in realtà cartelle o bundle o quello che è).

Tuttavia, l'esplorazione (nel caso di cartelle) o l'aggiunta alla lista (file standard) è eseguita solamente se il campo `is2Print` dell'elemento sotto esame è Vero; in caso contrario, si passa all'elemento successivo.

L'esplorazione e l'aggiunta alla lista procedono ricorsivamente, tramite aggiunte successive alla stringa `newFileList` di apposite aggiunte. Se ad esempio si sta considerando un file normale, è eseguita la procedura `terminalWithLevel`. Questa si limita ad aggiungere il nome del file alla fine della stringa:

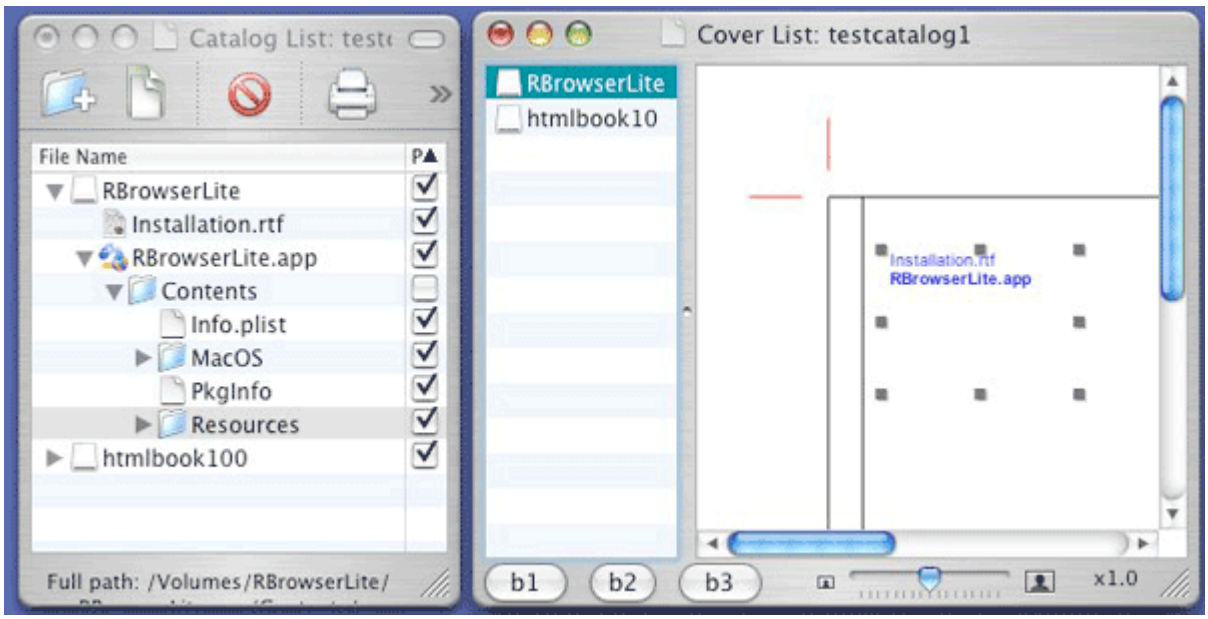
```
void
terminalWithLevel ( NSTextStorage* str, NSString *fname, int lev, BOOL
boldface )
{
    NSString    * xxx ;
    NSMutableAttributedString * temp ;
    NSFont * locFont ;

     xxx = [ NSString stringWithFormat: @"%@%@\n", prefixLevString( lev ),
fname ] ;
    temp = [ [ NSMutableAttributedString alloc] initWithString: xxx ];
    locFont = [ NSFont fontWithName: @"Arial" size: 9 ] ;
    if ( boldface )
         locFont = [ [NSFontManager sharedFontManager] convertFont: locFont
toHaveTrait: NSBoldFontMask];
        [ temp addAttribute: NSFontAttributeName value: locFont range:
NSMakeRange(0, [ temp length] ) ];
    [ str appendAttributedString: temp ] ;
}
```

Come già accaduto in altri casi, utilizzo un meccanismo di indentazione per evidenziare la profondità del file all'interno della gerarchia delle cartelle. La funzione `prefixLevString` produce una stringa di `lev` caratteri (spazi, nel caso, ma potrebbero essere tabulazioni o altro) che precedono il nome del file vero e proprio. Uno dei parametri della funzione permette di assegnare alla stringa il tratto grassetto prima di aggiungerla in coda alla lista.

Il meccanismo ricorsivo che esplora tutto l'albero delle cartelle è nascosto all'interno della seguente procedura:

```
void
addStringExpandItem ( NSTextStorage* str, CatDataSrc * myDataSrc,
FileStruct * elem, int nn, int lev )
{
    int            jj, nnx ;

    terminalWithLevel( str, [ elem fileName], lev, YES );
    for ( jj = 0 ; jj < nn; jj ++ )
    {
        FileStruct * elemx ;
        elemx = [ myDataSrc outlineView: nil child: jj ofItem: elem ];
        nnx = [ myDataSrc outlineView: nil numberOfChildrenOfItem: elemx ];
        if ( ! [ elemx is2Print ] ) continue ;
        if ( nnx > 0 )
        {
            addStringExpandItem ( str, myDataSrc, elemx, nnx, lev+1 );
        }
        else
        {
```

```
            terminalWithLevel( str, [ elemx fileName], lev+1, NO );
        }
    }
}
```



Per prima cosa, aggiungo il nome della cartella (perché sono sicuro che questa procedura è chiamata avendo come argomento una cartella), ma la scrivo in carattere grassetto per evidenziare la cosa. Poi, procedo ricorsivamente ad esaminare il contenuto della cartella, aggiungendo direttamente il nome del file nel caso di vero file, o chiamando ricorsivamente la procedura nel caso di ulteriori cartelle.

Avrei potuto certamente fare di meglio, ma per il momento mi fermo qui. Ah, pregasi notare come il meccanismo di *Undo* funziona anche per questi campi di testo, in caso di spostamento e ridimensionamento.

Capitolo 050 - Il costruttore diventa ispettore

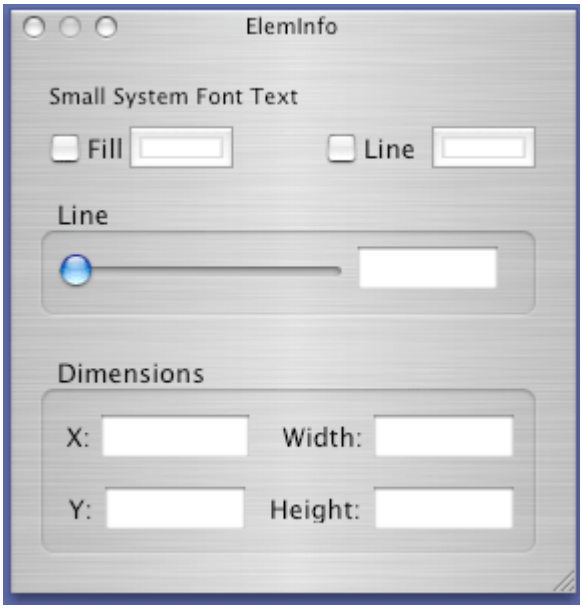
L'inserimento di nuovi elementi all'interno di una copertina avviene attraverso il mouse. Non è quindi più utile la finestra `ObjMaker`; in questo capitolo la sostituisco con una finestra che riporta le caratteristiche dell'elemento selezionato; attraverso la quale, è possibile modificare alcune caratteristiche dell'elemento.

Sorgenti: documentazione Apple

Prima stesura: 9 agosto 2004

Morte di un costruttore

La finestra `ObjMaker` non è più utile. Distruggo quindi il file nib e i file .h e .m relativi e sostituisco il tutto con una nuova finestra, invero piuttosto simile, di nome `ElemInfo` (confesso, ho solo cambiato il nome ai file). Scopo di questa finestra è di presentare in bella copia gli attributi dell'elemento selezionato (colore di riempimento, dimensioni, eccetera) e di modificarli; la modifica degli attributi di questo elemento dovrebbe riflettersi in tempo reale sull'elemento stesso, in modo da poter ritoccare elementi che al momento della creazione col mouse non erano riusciti molto bene.

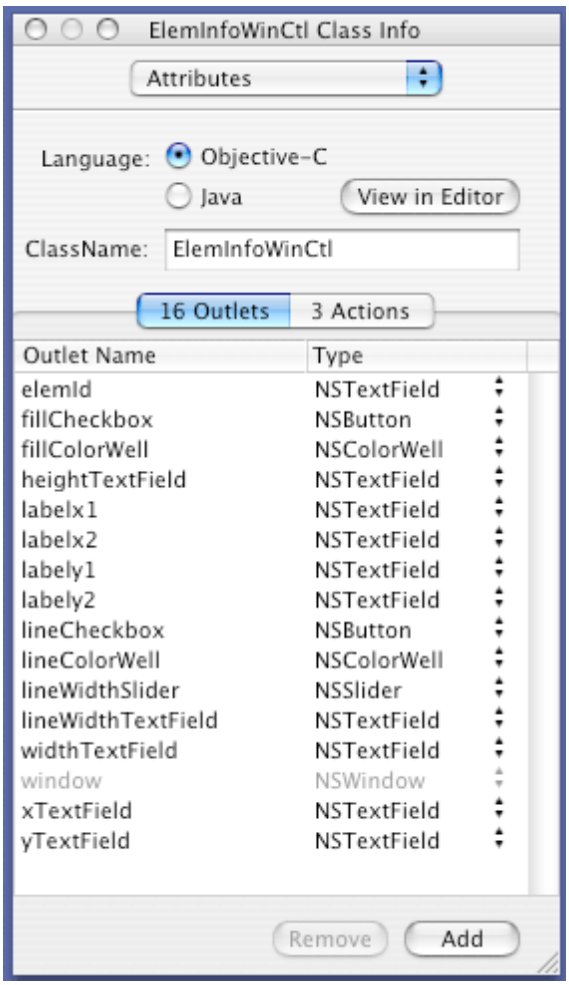


Con *Interface Builder* modifico la finestra in modo opportuno. Scompaiono il menu di scelta del tipo di elemento ed il pulsante di creazione, sostituiti da una semplice etichetta (riprenderà una breve descrizione dell'elemento selezionato). Una connessione degna di nota è quella che mette in relazione ogni singolo controllo della finestra (pulsanti, dispositivi di scelta del colore, ed anche i campi di testo) con il *File's Owner* attraverso l'azione `changedElem:`. In questo modo sarà possibile rispondere immediatamente ad ogni cambiamento del valore del controllo.

Una storia un po' più lunga è legata allo slider ed al campo di testo collegati, entrambi dedicati alla specifica dello spessore della linea. Ora, per poter collegare strettamente il campo testo e lo slider (in modo che una modifica sull'uno si rifletta nell'aggiornamento del valore dell'altro), ognuno è destinatario (*target*) di una *action* dell'altro. Per fare in modo che la modifica sia riconosciuta anche all'esterno (il meccanismo target/action può essere applicato una sola volta), ho scollegato slider e campo testo, e li ho invece collegati allo stesso metodo `slideChange:` del *File's Owner*. In altre parole, realizzo tramite mio codice il collegamento tra slider e campo testo, in modo da inserire a piacere altre istruzioni utili.

```
- (IBAction)
slideChange:(id)sender
{
    // recupero il valore corrente
    float newVal = [ sender floatValue ];
    // limite massimo
    if ( newVal > 10 ) newVal = 10 ;
    // imposto il valore per entrambi
    [ lineWidthTextField setFloatValue: newVal ] ;
    [ lineWidthSlider setFloatValue: newVal ] ;
    // dico che e' cambiato il valore di un attributo
    [ self changedElem: sender ];
}
```

Nel caso, l'istruzione aggiunta è proprio l'invocazione del metodo `changedElem:`, come se il tutto provenisse dal controllo interessato.



Una seconda operazione degna di nota è la presenza di nuovi outlet per collegare il *File's Owner* alle quattro etichette del box *Dimensions*. Infatti, quando l'elemento selezionato è una linea, non ha senso individuare le grandezze rappresentate (il punto di partenza e quello di fine del segmento) con le stesse etichette di un rettangolo (posizione e dimensioni).

A questo punto torno in *XCode* e scrivo (aggiorno) i metodi per la costruzione della finestra.

```
- (void)
windowDidLoad
{
    NSWindowController *controller ;

    [super windowDidLoad];
    // predisposizione dei vari elementi della finestra
    [fillCheckbox setState:NSOffState];
    ...
    [heightTextField setEnabled:YES];
    // aggiungo due osservatori quando cambia la finestra principale
    // in modo che possa aggiustare theSelectedElem
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowChanged:)
        name:NSWindowDidBecomeMainNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowResigned:)
        name:NSWindowDidResignMainNotification object:nil];
    // osservatore sul cambiamento della selezioen corrente
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(changedSelection:)
        name:@"CoverViewChangedSelection" object:nil];
    // pero' devo mettere a posto la prima volta
    // recupero la finestra di fronte a tutti
    controller = [ [NSApp mainWindow] windowController];
    // se e' del tipo giusto, aggiorno di conseguenza
    if (controller && [controller isKindOfClass:[CoverWinCtl class]])
            [self setMainWindow: [(CoverWinCtl *)controller cvrView]];
    else    [self setMainWindow:nil];
}
```

AGGIORNAMENTO DELLE INFORMAZIONI

A parte l'aggiustamento dei vari elementi dell'interfaccia, ci sono i tre abbonamenti a varie notifiche. Le prime due sono ordinaria amministrazione (ad esempio, nel capitolo 22): la finestra è interessata agli eventi di evidenziazione della finestra principale per aggiornare di conseguenza le informazioni mostrate al proprio interno. La terza notifica è invece meno standard: dico che la finestra è interessa a notifiche di nome `CoverViewChangedSelection`, in corrispondenza delle quali conviene eseguire il metodo `changedSelection:`. Non si tratta di una notifica standard di Cocoa, ma di una notifica che genererò direttamente all'interno del mio codice quando sono più o meno convinto che sia cambiata la selezione corrente all'interno di una copertina.

Subito dopo, le istruzioni seguenti si preoccupano di aggiustare l'aspetto della finestra la prima volta che appare, andando ad aggiustare direttamente le informazioni richieste.

Infatti, i tre metodi conseguenti alle notifiche si basano tutti sullo stesso metodo `setMainwindow::`

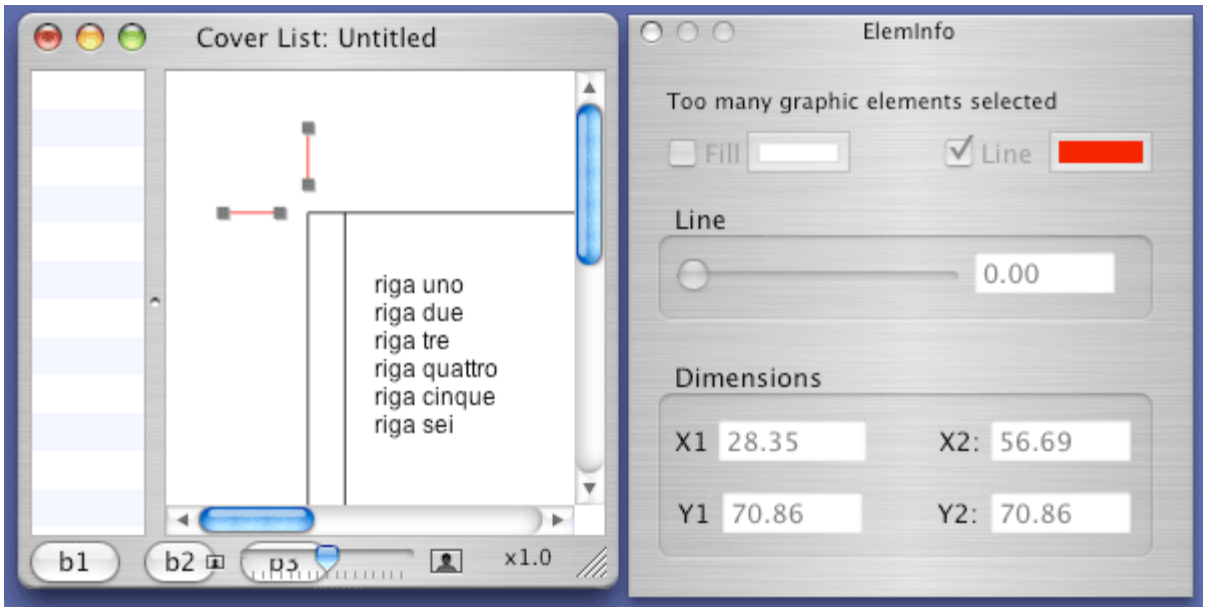
```
- (void)
mainWindowChanged: (NSNotification *)notification
{
            NSWindowController   *controller   =   [[notification   object]
windowController];
    if (controller && [controller isKindOfClass:[CoverWinCtl class]])
    {
        [self setMainWindow: [(CoverWinCtl *)controller cvrView]];
    }
    else    [self setMainWindow:nil];
}

- (void)
mainWindowResigned: (NSNotification *)notification
{
    [self setMainWindow:nil];
}

- (void)
changedSelection: (NSNotification *)notification
{
    id notifObj = [notification object] ;
    if ( [notifObj isKindOfClass:[CoverView class]])
    {
        [self setMainWindow: (CoverView *) notifObj ];
    }
}
```

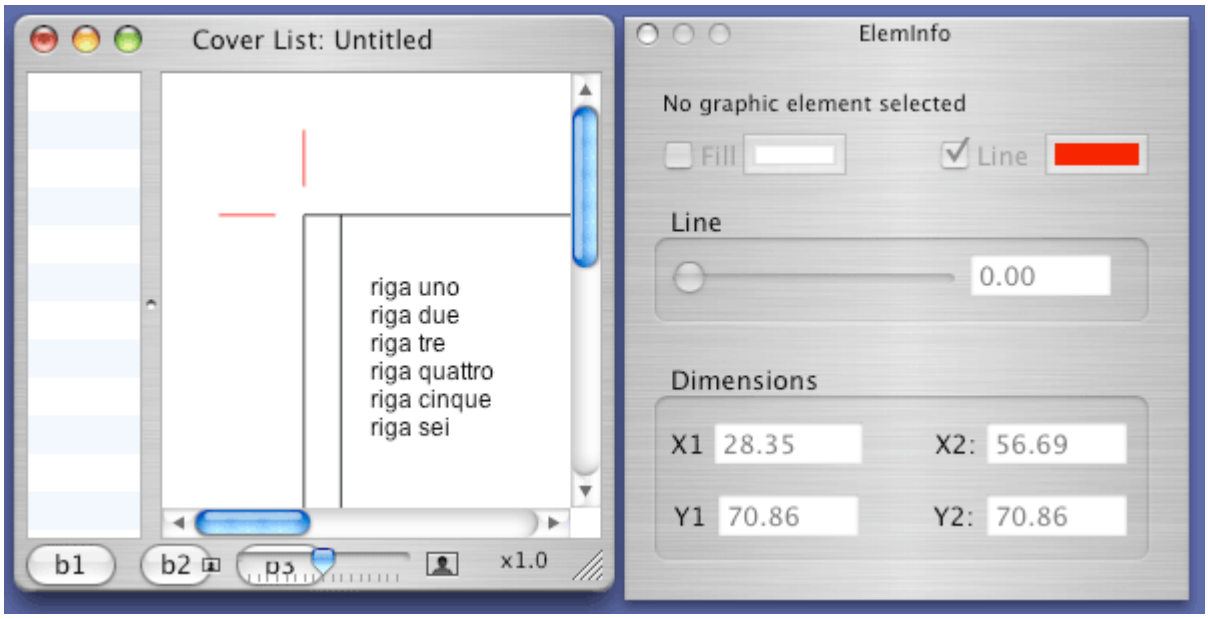
In tutti e tre i casi, se la finestra in primo piano dipende da una `CoverWinCtl`, allora siamo a cavallo: chiamo `setMainwindow:` passandogli come argomento l'oggetto della classe `CoverView` dove si trovano gli elementi d'interesse. Negli altri casi, chiamo il metodo con argomento nil.

Infatti il metodo `setMainWindow:` lavora come segue:

```
- (void)
setMainWindow: (CoverView *) coverView
{
    NSMutableArray * selArr ;
    int numElem ;
    // tutti gli elementi sono disabilitati
    [fillCheckbox setEnabled:NO];
    ...
    [heightTextField setEnabled:NO];
    theSelectedElem = nil;
    // e se la fienstra davanti non e' giusta, ho finito
    if ( coverView == nil )
        return ;
    // se arrivo qui, la finestra davanti ' di una copertina
    // guardo se ci sono elementi selezionati
    selArr = [ [ coverView theElements] getSelectedGraphics ];
    numElem = [ selArr count ];
    // se non ce ne sono...
    if ( numElem == 0 )
    {
        // dico che se ne fa nulla
        [ elemId setStringValue: @"No graphic element selected"];
```

```
    }
    // ... o ce ne sono troppi
    else if ( numElem > 1 )
    {
        // dico che se ne fa nulla
                [ elemId setStringValue: @"Too many graphic elements
selected"];
    }
    else
    {
        // c'e' un solo elemento, imposto il puntatore
        theSelectedElem = [ selArr objectAtIndex: 0 ] ;
        // e aggiorno la finestra di conseguenza
        [ self updateWindowInfo ];
    }
}
```

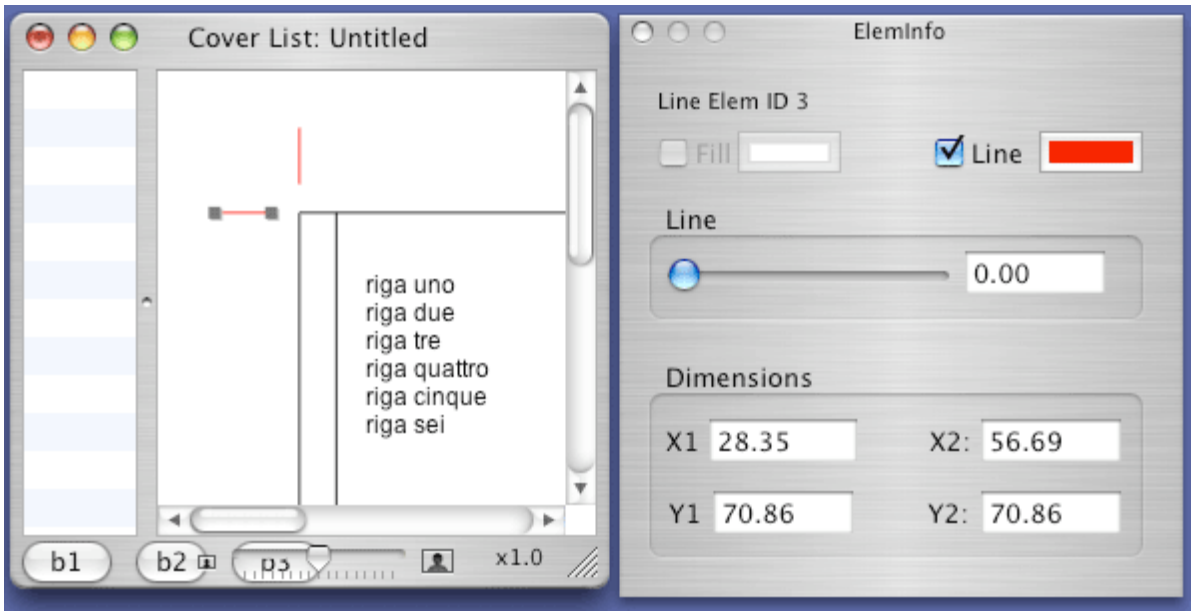




Per prima cosa, pone la finestra in uno stato di default (tutto disabilitato); poi controlla se c'è un elemento selezionato. Sicuramente l'elemento non c'è se la finestra di fronte

non è corretta (parametro di valore `nil`); se invece la `CoverView` è effettiva, verifica quanti sono gli elementi selezionati. Se non ce ne sono, o ce n'è più d'uno, si dichiara incapace di trattare la cosa, e riporta un messaggio appropriato. Se invece, e finalmente, c'è un solo elemento selezionato, passa ad aggiornare il contenuto della finestra.

```
- (void)
updateWindowInfo
{
    NSRect locR = [theSelectedElem elemLimit] ;
    // attivo tutti gli elementi della finestra
    [fillCheckbox setEnabled:YES];
    ...
    [heightTextField setEnabled:YES];
    // predispongo gli elementi che corrispondono agli attributi comuni
    [fillCheckbox setState: [ theSelectedElem cceIsFilled] ];
    [fillColorWell setColor: [ theSelectedElem cceFillColor] ];
    [lineCheckbox setState: [ theSelectedElem cceIsStroked] ];
    [lineColorWell setColor:[ theSelectedElem cceLineColor] ];
    [lineWidthSlider setFloatValue: [ theSelectedElem cceLineWidth] ];
    [lineWidthTextField setFloatValue: [ theSelectedElem cceLineWidth] ];
    // la linea va trattata in maniera particolare
    if ( [theSelectedElem isKindOfClass:[CCE_Line class]])
    {
        // recupero i due punti di inizio e fine
        NSPoint locP = [(CCE_Line*)theSelectedElem startPoint] ;
           [xTextField setStringValue: [ NSString stringWithFormat: @"%f",
locP.x ] ];
           [yTextField setStringValue: [ NSString stringWithFormat: @"%f",
locP.y ] ];
        locP = [(CCE_Line*)theSelectedElem endPoint] ;
         [widthTextField setStringValue: [ NSString stringWithFormat: @"%f",
locP.x ] ];
            [heightTextField setStringValue: [ NSString stringWithFormat:
@"%f", locP.y ] ];
        [ elemId setStringValue: [ theSelectedElem describeMe ] ];
        // aggiusto le etichette coi nomi corretti
        [labelx1 setStringValue: @"X1:" ];
        [labelx2 setStringValue: @"X2:" ];
        [labely1 setStringValue: @"Y1:" ];
        [labely2 setStringValue: @"Y2:" ];
        // una linea non puo' essere riempita di colore
        [fillCheckbox setEnabled: NO];
        [fillColorWell setEnabled: NO];
        return ;
    }
    // se arrivo qui, sono sicuro che non e' una linea
        [xTextField setStringValue: [ NSString stringWithFormat: @"%f",
locR.origin.x ] ];
        [yTextField setStringValue: [ NSString stringWithFormat: @"%f",
locR.origin.y ] ];
      [widthTextField setStringValue: [ NSString stringWithFormat: @"%f",
locR.size.width ] ];
     [heightTextField setStringValue: [ NSString stringWithFormat: @"%f",
locR.size.height ] ];
    // predispongo le etichette standard
    [labelx1 setStringValue: @"X:" ];
    [labely1 setStringValue: @"Y:" ];
    [labelx2 setStringValue: @"Width:" ];
    [labely2 setStringValue: @"Height:" ];
    // preparo una scritta a seconda dell'elemento
    [ elemId setStringValue: [ theSelectedElem describeMe ] ];
}
```



Per prima cosa, attivo tutti gli elementi dell'interfaccia, e poi aggiorno il valore di ciascuno con i valori che ricavo dalle variabili d'istanza dell'elemento stesso. Devo come al solito distinguere il comportamento degli elementi `CCE_Line` dagli altri; la differenza si trova nel significato dei quattro campi relativi alle dimensioni, che nel caso di una linea sono le coordinate dei due punti estremi del segmento; negli altri casi si tratta invece del punto origine e delle dimensioni (larghezza ed altezza) del rettangolo che racchiude l'elemento.

Per produrre poi una stringa in grado di descrivere l'elemento, ricorro come sempre al polimorfismo, realizzando per ciascuna classe di elementi il metodo `describeMe:`. Ad esempio, per un cerchio il metodo è:

```
- (NSString*)
describeMe
{
      return ( [ NSString stringWithFormat:@"Circle Elem ID %d", [ self
objID] ] );
}
```

Ho così trovato anche un uso interessante per la variabile d'istanza `objId` che avevo introdotto ai soli scopi di debugging.

C'è un'ultima notazione prima che tutto quanto funzioni; occorre aggiungere l'invio della notifica `CoverViewChangedSelection` nel punto più opportuno. Non ho trovato di meglio che farlo all'interno del metodo `mouseDown:` della classe `CoverView`, al termine di tutte le operazioni, attraverso l'istruzione

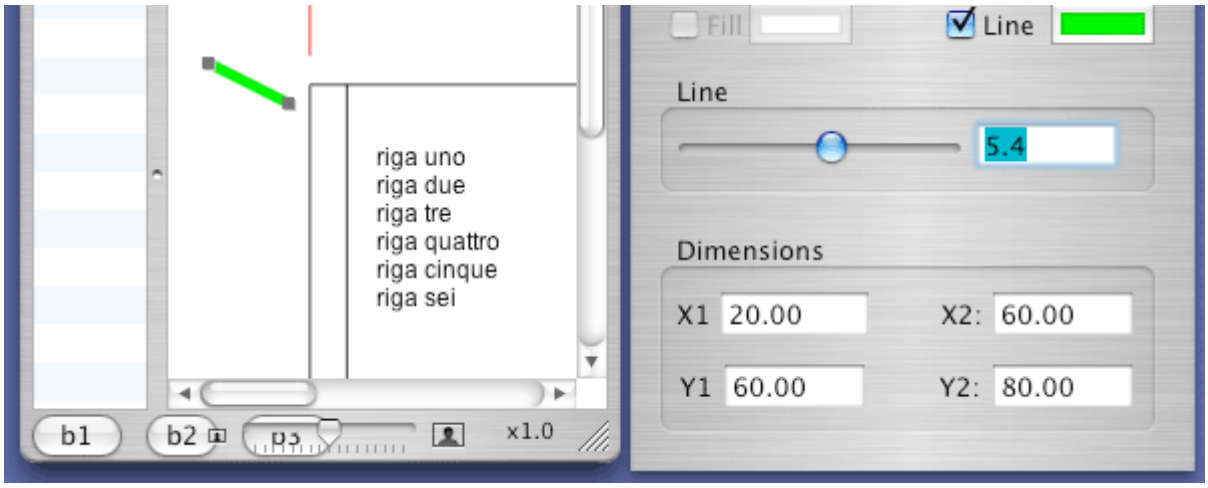
```
    [ [NSNotificationCenter defaultCenter]
        postNotificationName:@"CoverViewChangedSelection"
        object:self];
```

Infatti, alla conclusione di questo metodo, o è stato creato (e quindi selezionato) un nuovo elemento, o comunque è avvenuto un clic (o trascinamento), il quale, a meno di casi particolari, ha cambiato la selezione corrente.

CAMBIO DEI PARAMETRI

Oltre che alla visualizzazione del valore dei parametri, la finestra può essere utilizzata per modificarne il valore. Operando su uno qualsiasi dei controlli (due pulsanti di spunta, due `NSColorWheel`, lo slider ed il campo testo associato, e i quattro campi testo con le coordinate), al termine della modifica è automaticamente chiamato il metodo `changedElem:`. Quando, all'interno di *Interface Builder*, ho modificato la finestra, mi sono preoccupato di attribuire ad ogni controllo un tag univoco. In questo modo il metodo può discriminare facilmente chi ha causato la chiamata.

```
- (IBAction)
changedElem:(id)sender
{
    float p1, p2, p3, p4 ;
    // a seconda dell'attributo modificato
    switch ( [ sender tag ] ) {
    case 10 :     // stato di riempimento
        [ theSelectedElem setCceIsFilled: [ fillCheckbox state ]];
        break ;
    case 11 :     // colore di riempimento
        [ theSelectedElem setCceFillColor: [ fillColorWell color ]];
        break ;
    case 12 :     // stato di disegno linea/contorno
        [ theSelectedElem setCceIsStroked: [ lineCheckbox state ]];
        break ;
    case 13 :     // colore della linea
        [ theSelectedElem setCceLineColor: [ lineColorWell color ]];
        break ;
    case 14 :     // dimensione della linea
    case 15 :
        [ theSelectedElem setCceLineWidth: [ lineWidthSlider floatValue ]];
        break ;
    case 16 :     // coordinate limite dell'elemento
    case 17 :
    case 18 :
    case 19 :
        // recupero i quattro valori
        p1 = [xTextField floatValue ];
        p2 = [yTextField floatValue ];
        p3 = [widthTextField floatValue ];
        p4 = [heightTextField floatValue ];
        // la linea ha il solito trattamento particolare
        if ( [theSelectedElem isKindOfClass:[CCE_Line class]])
                    [theSelectedElem  resizeMeWith:  NSMakePoint(p1,p2)  andPt:
NSMakePoint(p3,p4) ];
        else
                    [theSelectedElem  resizeMeWith:  NSMakePoint(p1,p2)  andPt:
NSMakePoint(p1+p3,p2+p4) ];
        break ;
    default:
        NSLog( @"changed %@", sender );
    }
    // forzo un ridisegno
    [ [theSelectedElem ownerView] setNeedsDisplay: YES ];
}
```



In corrispondenza di ogni chiamata sono eseguite le istruzioni corrispondenti alla modifica richiesta. Se ad esempio si è verificata un'azione sullo slider (caso 14 o 15), è modificata la variabile d'istanza `cceLineWidth` dell'elemento selezionato; così, similmente, negli altri casi. Un po' più complicata (non di tanto) la modifica di una delle coordinate, in cui si deve distinguere tra `CCE_Line` e gli altri elementi. è chiamato il metodo `resizeMeWith:andPt:` perché questo metodo ha incorporato il meccanismo di undo.

LE ALTRE MODIFICHE

Nel corso delle mie operazioni, ho effettuato altre modifiche.

Ad esempio, ho dovuto aggiungere i metodi accessor per le variabili della classe `CCE_Line`, in quando per la visualizzazione delle coordinate era necessario accedervi.

Ho provato poi a giocare un po' con la modifica dei font dei campi di testo `CCE_Text` (attualmente limitati alla sola visualizzazione della lista degli elementi). Quando l'utente utilizza il pannello font o seleziona le voci di menu come *Grassetto*, *Corsivo*, eccetera, Cocoa invia vari messaggi in giro per la gerarchia. Il più gettonato pare essere il messaggio `changeFont:` inviato al *First Responder*. Per essere sicuro i intercettare tale messaggio, piazzo il metodo corrispondente all'interno della classe `CoverWinCtl`, alla quale sono sicuro arrivi.

```
- (void)
changeFont:(id)sender
{
    [ [ cvrView theElements ] changeFont: sender ];
}
```

Mi limito a passare la palla al gestore di tutti gli elementi all'interno della finestra. A sua volta, l'oggetto `CCE_ElemGroup` non farà altro che spazzolare tutti gli elementi della finestra, e inviare un apposito metodo ai soli elementi `CCE_Text` selezionati:

```
- (void)
changeFont:(id)sender
{
    int         i , numElem ;
```

```
    CCE_BasicForm * elem ;

    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
            if ( [ elem cceIsSelected ] && [elem isKindOfClass:[CCE_Text
class]])
             [ elem changeFont: sender ];
    }
}
```

Finalmente, l'oggetto `CCE_Text` potrà trattare il cambio di font. Lo fa nel seguente modo (seguo pedissequamente le documentazione Apple):

```
- (void)
changeFont:(id)sender
{
    // recupero il font corrente
    NSFont * curFont = [ textStorage font ];
    // lo converto nel nuovo font come stabilito da NSFontManager
    NSFont * newFont = [sender convertFont:curFont];
    // imposto il nuovo font
    [textStorage setFont:newFont];
    // ridisegno tutto
    [ ownerView setNeedsDisplay: YES ];
}
```

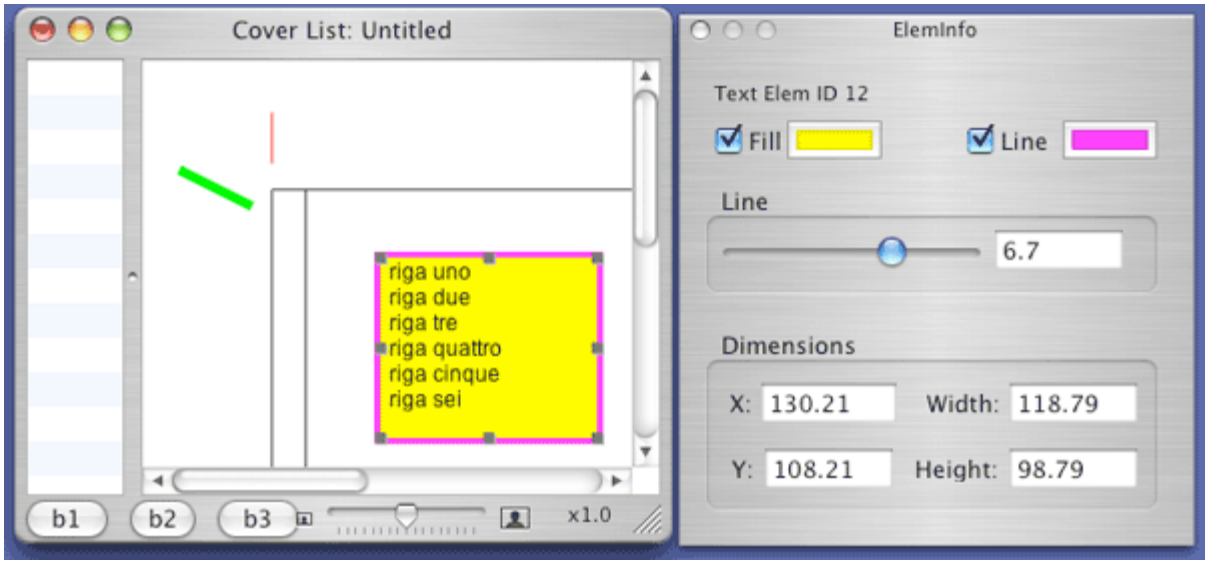
Qui bisogna tenere presente che il *sender* è sempre l'istanza (condivisa da tutta l'applicazione) della classe `NSFontManager`, che si occupa appunto di gestire il pannello dei font, e che è utilizzato per trasformare il font corrente nel nuovo font selezionato (perché trasformare? Perché `changeFont:` è invocato anche quando, ad esempio, si seleziona la voce Grassetto da menu).

Nel realizzare la precedente funzionalità, mi sono accorto di alcuni problemi nella costruzione della lista di file. Le procedure relative, infatti, imponevano sempre lo stesso tipo di carattere, rendendo quindi insensbile `CCE_Text` alla modifica del font. Ho allora cambiato il metodo come segue:

```
- (void)
tableViewSelectionDidChange:(NSNotification *)aNotification
{
    NSTextStorage * newFileList ;
    CCE_Text        * wxw ;
    int               elNum, jj, nn ;
    NSTableView       * ov = (NSTableView*)[ aNotification object ] ;

    // predispongo il volume in selezioen corrente
    currentVol = [ ov selectedRow ] ;
    // recupero se disponibile un CCE_Text per mostrare l'elenco dei file
    wxw = [ [[refDoc coverWin] cvrView] fileList];
    // se non c'e', ho nulla da fare
    if ( wxw == nil )
        return ;
    // costruisco un nuovo contenitore per il testo
    newFileList = [[NSTextStorage alloc] initWithString:@"dummy"];
     // ci metto del testo 'dummy' altrimenti l'assegnazione di font non
```

```
funziona
    [ newFileList setFont:[ [ wxw textStorage] font ] ];
    // recupero il primo livello del volume
    elNum = [ self outlineView: nil numberOfChildrenOfItem: nil ];
    // esploro il primo livello
    for ( jj = 0 ; jj < elNum; jj ++ )
    {
        FileStruct * elem ;
        // esamino tutti gli elementi uno ad uno
        elem = [ self outlineView:     nil child: jj ofItem: nil ];
        // controllo se ha figli
        nn = [ self outlineView: nil numberOfChildrenOfItem: elem ];
        // se non e' da stampare, salto
        if ( ! [ elem is2Print ] ) continue ;
        // se e' una directory, espando di un livello
        if (nn > 0 )
            addStringExpandItem (newFileList, self, elem, nn, 0 );
        else    // lo aggiungo al testo gia' predisposto
            terminalWithLevel( newFileList, [ elem fileName], 0, NO );
    }
    // elimino i primi cinque caratteri, che non mi servono piu'
    [ newFileList deleteCharactersInRange: NSMakeRange( 0, 5 )];
    // aggiorno la stringa
    [[ wxw textStorage] setAttributedString: newFileList ] ;
    // forzo un rinfresco della vocerView
    [ [[refDoc coverWin] cvrView] setNeedsDisplay: YES ];
}
```

Qui recupero dall'elemento `CCE_Text` il font corrente, e lo utilizzo per scrivere la nuova lista di file. A quanto pare, cominciare con una stringa nulla impedisce l'assegnazione del font (mi sembra ragionevole, anche se seccante). Ho allora utilizzato una stringa fittizia, che poi elimino alla fine del procedimento.



Per concludere con la classe `CCE_Text`, ho aggiunto la possibilità di disegnare (ed eventualmente riempire di colore) il rettangolo che contiene il testo; la cosa giusta da fare sarebbe di impostare il colore del font e dello sfondo tramite il pannello font, ma non ci riesco (ma prima o poi ci arrivo...); utilizzando invece la finestra `ElemInfoWinCtl` è una sciocchezza. Per il disegno vero e proprio, lascio fare alla superclasse:

```
- (void) drawElement: (NSRect) inRect
{
    // recupero il primo layout
      NSLayoutManager *layoutManager = [ [ textStorage layoutManagers ]
objectAtIndex: 0] ;
    // recupero il range del testo
        NSRange glyphRange = [layoutManager glyphRangeForTextContainer:
theTextCont ];
    // disegno del testo
    [ super drawElement: inRect ];
          [ layoutManager drawGlyphsForGlyphRange:glyphRange atPoint:
elemLimit.origin ];
}
```

Infatti, ho modificato il metodo di default della classe `CCE_BasicForm` perché disegni un rettangolo e lo riempia (in precedenza tale metodo era vuoto, e doveva essere riscritto da ogni sottoclasse):

```
- (void)
drawElement: (NSRect) inRect
{
    // se il rettangolo e' pieno
    if ( cceIsStroked )
    {
        NSBezierPath *path = [NSBezierPath bezierPathWithRect: elemLimit ];
        // imposto il colore del contorno
        [ cceLineColor set ];
        // imposto lo spesosre del contorno
        [ path setLineWidth: [ self cceLineWidth ]];
        // disegno il contorno
        [ path stroke ];
    }
    if ( cceIsFilled )
    {
        NSBezierPath *path = [NSBezierPath bezierPathWithRect: elemLimit ];
        // imposto il colore del pieno
        [ cceFillColor set ];
        // riempi tutto
        [ path fill ];
    }
}
```

Infine, l'ultima modifica degna di nota è il meccanismo di *undo* anche per gli attributi degli elementi (colori, riempimenti, eccetera). Ho quindi modificato tutti i metodi accessor della classe `CCE_BasicForm` in modo che tangano traccia delle operazioni effettuate sugli attributi. Ad esempio per lo spessore della linea:

```
- (void)setCceLineWidth:(float)newLineWidth
{
    // gestione dello undo
    [[self undoManager] setActionName: @"BasicForm -> cceLineWidth"];
          [[[self undoManager] prepareWithInvocationTarget: [ownerView
theElements]]
              changeElem: self attrib: kUndoLineWidth value: [NSNumber
numberWithFloat: cceLineWidth] ];
    // assegnazione effettiva
    cceLineWidth = newLineWidth;
}
```

In corrispondenza di una operazione di undo (o anche di redo), è invocato il metodo `changeElem:attrib:value:` che conserva la precedente operazione sotto forma di: oggetto; identificatore dell'attributo modificato (secondo un mio codice personale); valore dell'attributo (sotto forma di oggetto). Ancora una volta, per mantenere continuità nelle operazioni di undo, responsabile della faccenda è sempre l'oggetto `theElements`; di conseguenza, all'interno della classe `CCE_ElemGroup` si trova il metodo seguente:

```
- (void)
changeElem: (CCE_BasicForm*) elem attrib: (int) quale value: (id) valore
{
    switch ( quale ) {
      case kUndoLineWidth:      [ elem setCceLineWidth: [(NSNumber*)valore
floatValue ] ]; break ;
     case kUndoFillColor:      [ elem setCceFillColor: (NSColor*)valore ];
break ;
     case kUndoLineColor:      [ elem setCceLineColor: (NSColor*)valore ];
break ;
    case kUndoIsFilled:        [ elem setCceIsFilled:    [(NSNumber*)valore
boolValue ] ]; break ;
     case kUndoIsStroked:     [ elem setCceIsStroked:     [(NSNumber*)valore
boolValue ] ]; break ;
      case kUndoIsSelected:      [ elem setCceIsSelected:[(NSNumber*)valore
boolValue ] ]; break ;
    case kUndoIsLocked:        [ elem setCceIsLocked:    [(NSNumber*)valore
boolValue ] ]; break ;
    }
    [ [ self ownerView ] setNeedsDisplay: YES ];
}
```

Per avere uniformità di trattamento, l'idea è di trasformare ogni valore degli attributi in un oggetto non meglio caratterizzato. La discriminazione sul tipo avviene tramite il codice dell'attributo.

Capitolo 051 - Rotazioni e Rivoluzioni, parte 1

Capitolo sofferto; avevo cominciato ardito con l'idea di introdurre la possibilità di ruotare gli elementi grafici di un angolo a piacere. Mi sono accorto strada facendo che le cose si sarebbero complicate troppo, a meno di non disfare alcune classi. Ho allora stravolto la struttura degli elementi, rifacendo una consistente parte della classe `CCE_BasicForm` (e discendenti). Nel farlo, mi sono imbattuto in nuovi concetti, potenti ma non troppo semplici. Il testo è diviso in due capitoli per non appesantire troppo le pagine.

Sorgenti: documentazione Apple.

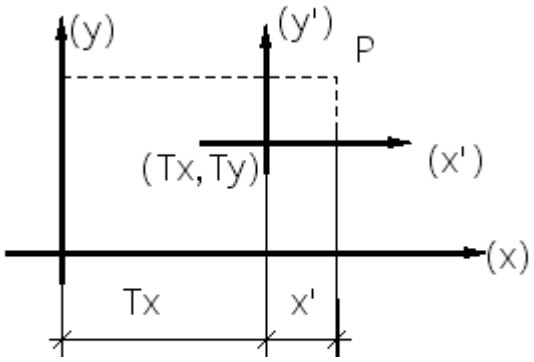
Prima stesura: 18 agosto 2004.

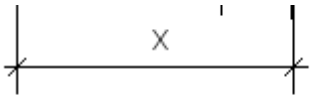
TRASFORMAZIONI AFFINI

C'è un concetto fondamentale, che deve essere chiaro a tutti coloro che producono programmi di grafica vettoriale per un computer, in mezzo tra matematica e geometria. Sto parlando della trasformazione affine, e della conseguente classe messa a disposizione da Cocoa, `NSAffineTransform`.

Per disegnare punti, linee e curve, c'è bisogno di un sistema di coordinate (cartesiane ortogonali) di riferimento. In generale, esistono diversi sistemi di riferimento *SdR*: ad esempio, c'è il SdR relativo al monitor, e c'è il SdR relativo alla finestra. Lo stesso pixel dello schermo ha una diversa rappresentazione, sotto forma di coordinate, nel SdR del monitor e nel SdR della finestra in cui si trova. Normalmente, è possibile passare da un SdR ad un altro attraverso semplici calcoli: l'insieme di questi calcoli è chiamato *Trasformazione* (affine, perché mantiene alcune proprietà geometriche, quali il parallelismo). Nel caso dei SdR citati, la trasformazione è una semplice traslazione; per passare dalle coordinate (x,y) del punto P nel SdR del monitor a quelle (x',y') nel SdR della finestra, basta conoscere le coordinate (Tx,Ty) dell'origine del SdR della finestra nel SdR del monitor. Allora i calcoli da effettuare sono:

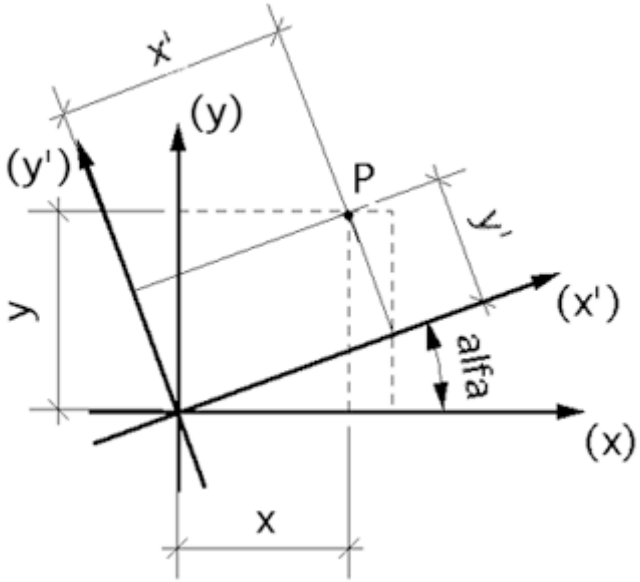
```
x = x' + Tx
y = y' + Ty
```





Questo giochino funziona se i due SdR hanno lo stesso orientamento, ovvero se l'angolo formato dai due semiassi positivi delle ascisse è nullo. Tuttavia, in molti casi, ciò non è vero: occorre tenere conto anche della possibilità di una rotazione relativa tra i due sistemi di riferimento. Per ottenere la trasformazione ci sono alcune considerazioni trigonometriche da fare, ma alla fine, se non sbaglio, per passare dalle coordinate (x,y) del punto P nel primo SdR alle coordinate (x',y') nel secondo SdR, che condivide col primo la stessa origine, ma è ruotato di un angolo alfa, si ha:

```
x = x' cos(alfa) - y' sin (alfa)
y = x' sin(alfa) + y' cos(alfa)
```



Trasformazioni più complesse della semplice traslazione e rotazione attorno all'origine si ottengono componendo tra loro queste due trasformazioni semplici. Ad esempio, una rotazione attorno ad un punto Q qualunque si ottiene con tre trasformazioni semplici: una traslazione nel punto Q, la rotazione attorno alla nuova origine Q, una nuova traslazione per tornare al punto di partenza. È estremamente importante la sequenza: le stesse trasformazioni in sequenza diversa producono risultati diversi. Ad esempio, con la sequenza precedente, se eseguo prima le due traslazioni, queste si annullano l'una con l'altra, ed il risultato della sequenza traslazione - traslazione - rotazione produce la sola rotazione nell'origine, mentre il risultato di traslazione - rotazione - traslazione produce come risultato la rotazione attorno al punto Q, ben diverso dall'origine.

MATRICI DI CLASSE

Ora, sarebbe molto comodo avere un meccanismo che permetta di rappresentare in modo semplice una trasformazione complessa, e di ottenere tale trasformazione combinando in maniera opportuna le trasformazioni semplici. Tale meccanismo esiste, ma utilizza un concetto matematico che per me è molto semplice e pratico, ma è spesso

superiore alla normale cultura matematica dispoibile: sto parlando delle matrici.

Le trasformazioni bidimensionali si possono rappresentare in maniera pratica attraverso matrici di tre elementi per tre, con la convenzione di rappresentare i punti nel piano attraverso un vettore di tre elementi. Dato un punto P=(x,y), lo si rappresenta con il vettore `[x y 1]`; la trasformazione è invece rappresentata da una matrice 3x3:

```
T = [ a d 0
      b e 0
      c f 1 ]
```

La trasformazione del punto P attraverso la trasformazione affine T produce il nuovo punto (lo stesso punto, ma con diversa rappresentazione) `P' = P T`. Attraverso l'algebra delle ed alcune semplici considerazioni, si arriva facilmente ad un insieme di regole per gestire la composizione delle trasformazioni.

Tuttavia, piuttosto che dilungarmi in queste considerazioni matematiche, illustro la classe `NSAffineTransform`, che semplifica il tutto nascondendo questi stupidi dettagli.

La classe `NSAffineTransform` è un meccanismo che rappresenta queste trasformazioni affini e permette di lavorare comodamente senza preoccuparsi più di tanto di matrici e vettori.

Ci sono ovviamente dei metodi per costruire trasformazioni, da zero (`+transform`, che produce la trasformazione identica, che lascia tutto come sta) o a partire da un'altra trasformazione (`initWithTransform`). A questo punto, ci sono metodi per costruire trasformazioni più complesse a parire dalle trasformazioni semplici: il metodo `translateXBy:yBy:` concatena una traslazione, i metodi `rotateByDegrees:` e `rotateByRadians:` eseguono rotazioni nell'origine (in gradi o radianti); ci sono metodi per comporre trasformazioni (`appendTrasform:` e `prependTransform:`) e per costruire la trasformazione inversa (`invert`). Chi fosse curioso di vedere le matrici che eseguono le operazioni di trasformazione, può usare il metodo `transformStruct` che appunto restituisce tale matrice.

Poi ci sono dei metodi per applicare queste trasformazioni: per trasformare le coordinate di un punto c'è `transformPoint:`, per trasformare dimensioni (cioé, per strutture `NSSize`) c'è `transformSize:`, ed infine per trasformare interni percorsi (istanze della classe `NSBezierPath`) c'è `transformPath:`.

Ci sono poi due importanti metodi, `set` e `concat`, per applicare le trasformazioni indicate al contesto grafico corrente. Ora, quando sono in corso operazioni di disegno all'interno di una finestra, Cocoa mantiene un contesto grafico, ovvero un insieme di variabili proprie di quell'ambiente di disegno; tra le variabili, il sistema di riferimento corrente, il colore, le dimensioni delle linee, eccetera. Se `miaTF` è una istanza della classe `NSAffineTransform`, l'istruzione `[ miaTF set ]` impone al corrente contesto grafico la trasformazione indicata. L'istruzione `[ miaTF concat ]`, invece, concatena la trasformazione indicata alla corrente in uso. Per spiegare meglio la cosa, faccio un esempio; supponiamo di voler disegnare un rettangolo, descritto dal `NSBezierPath mioPath`, ruotato di venti gradi rispetto alle coordinate della view corrente. Costruisco allora una trasformazione `miaFT` più o meno in questo modo:

```
miaTF = [ NSAffineTransform transform ];
[ miaTF rotateByDegrees: 20 ];
```

Quando, in un qualche metodo di disegno, sono nel punto di tracciatura, eseguo la seguente sequenza:

```
[ miaTF concat ];
[ mioPath stroke ] ;
```

Ho per prima cosa cambiato il sistema di riferimento, effettuando una rotazione in aggiunta alla trasformazione correte (che è quella che trasforma il SdR del monitor nel SdR della view corrente); poi ho disegnato il rettangolo; questo, di per sè, non è ruotato; ma avendo cambiato il sistema di riferimento, sarà disegnato ruotato di venti gradi.

In alternativa, avrei potuto lavorare in questo modo:

```
mioPath = [ miaTF transformPath: mioPath ];
[ mioPath stroke ];
```

Qui, invece di cambiare il SdR, ho trasformato il path ruotandolo come richiesto, e poi ho disegnato direttamente il path.

I NUOVI ELEMENTI GRAFICI

Facendo tesoro delle trasformazioni affini, ho ristrutturato le classi `CCE_BasicForm` e discendenti in maniera completamente differente rispetto al passato. Parto dal seguente principio: ogni elemento ha un proprio SdR, all'interno del quale hanno luogo le varie operazioni di disegno. In questo modo, ogni elemento possiede un punto di origine del SdR chiamato `drawPoint` ed un angolo di rotazione `rotAngle` rispetto al SdR della finestra. Ogni elemento possiede poi una dimensione (`localSize`, una struttura `NSSize`), ovvero una larghezza ed una altezza all'interno della quale si trova il disegno dell'elemento. Il disegno dell'elemento avviene sempre attraverso un elemento della classe `NSBezierPath` (tutti, a parte `CCE_ElemGroup`...). Completano le variabili d'istanza le due trasformazioni affini `localTF` e `reverTF` che permettono di passare dal SdR della finestra al SdR proprio dell'elemento. La classe `CCE_BasicForm` risulta quindi così costruita nelle sue variabili d'istanza:

```
@interface CCE_BasicForm : NSObject {
    // identificatore dell'elemento
    long            objID ;
    // punto nodale dell'elemento
    NSPoint           drawPoint ;
    // angolo di rotazione corrente
    float           rotAngle ;
    // lista degli handle, sono al massimo otto
    int               numOfHdl ;
    NSPoint           hdlList[8] ;
    // la view in cui si trova
    CoverView       * ownerView ;

    // metto da parte la tf affine
    NSAffineTransform * localTF ;
```

```
    NSAffineTransform * reverTF ;
    // dimensioni del rettangolo in coordinate locali
    NSSize            localSize ;
    // tengo il percorso per meglio disegnare
    NSBezierPath    * theDrawPath ;

    // spessore della linea di contorno
    float            cceLineWidth ;
    // colore dell'eventuale interno e del contorno
    NSColor           * cceFillColor ;
    NSColor           * cceLineColor ;
    // se devo colorare l'interno o meno
    BOOL            cceIsFilled ;
    BOOL            cceIsStroked ;
    // se l'elemento e' selezionato o meno
    BOOL            cceIsSelected ;
    // se l'lemento puo' essere manipolato o meno
    BOOL            cceIsLocked ;
}
```

Scompare il rettangolo `elemLimit`, che tanto mi aveva dato da fare nella precedente realizzazione; più avanti si vedrà come è degnamente rimpiazzato. Compaiono degli ovvi e banali nuovi metodi accessor.

L'inizializzazione dell'elemento è cambiata:

```
- (id)
initWithId: (int) ident inView: (CoverView*) cv
{
    // classica inizializzazione della superclasse
    self = [super init];
    if (self)
    {
        // inizializzazione delle variabili d'istanza comune
        // objId, e' compito del chiamante
        objID = ident ;
        // disegnato nell'origine, per il momento
        drawPoint = NSMakePoint( 0, 0 );
        // con angolo di rotazione nullo
        [ self setRotAngle: 0.0F ];
        // aggiusto le trasformazioni affini
        [ self calcLocalTranform ];
        [ self setTheDrawPath: [NSBezierPath bezierPathWithRect:
                    NSMakeRect(0.0, 0.0, 1.0, 1.0) ] ];
        // ci sono otto handle standard
        numOfHdl = 8 ;
        [ self buildHdlList ];
        // coverView, e' compito del chiamante
        ownerView = cv ;
        // spessore della linea, poca roba
        ...
    }
    return self;
}
```

Ho infatti preferito racchiudere in una unica istruzione l'inizializzazione di un elemento dotandolo di identificatore e di view in cui rappresentarlo. C'è una istruzione importante, il calcolo delle trasformazioni:

```
- (void)
```

```
calcLocalTranform
{
    // trasformazione affine per la rototraslazione
    NSAffineTransform *transform = [NSAffineTransform transform];
    // rototraslazione
    [ transform translateXBy: drawPoint.x yBy: drawPoint.y ];
    [ transform rotateByDegrees: rotAngle ];
    [ self setLocalTF: transform ];
    // gia' che ci sono, calcolo anche l'inversa
    [ transform invert ];
    [ self setReverTF: transform ];
}
```

La costruzione della trasformazione è molto semplice: si esegue la traslazione nel punto origine o poi la rotazione dell'angolo previsto; calcolo anche la trasformazione inversa, che è sempre utile.

Per vedere come questi nuovi concetti si applicano, passo a discutere la realizzazione della classe `CCE_Rect`. Cambia anche qui il metodo per l'inizializzazione.

```
- (id)
initWithId: (int) ident inView: (CoverView*) cv withRect: (NSRect) aRect
andAngle: (float) angle
{
    self = [ super initWithId: ident inView: cv ] ;
    if ( self )
    {
        // uso l'origine di aRect come punto di partenza
        [ self setRotAngle: angle ] ;
        [ self buildMeFromPt: aRect.origin withSize: aRect.size ];
    }
    return self ;
}
```

Oltre all'identificatore e alla view che lo contiene, il rettangolo è costruito a partire da un rettangolo (ovvio) e dall'angolo di rotazione. Il metodo si limita a chiamare il costruttore della superclasse, ad impostare l'angolo di rotazione con un metodo accessor, e a chiamare il nuovo metodo `buildMeFromPt:withSize:`, che sostituisce il precedente metodo per due punti.

```
- (void)
buildMeFromPt: (NSPoint) stPt withSize: (NSSize) dim
{
    // stPt sono le coordinate (view) del punto base
    // dim e' la distanza (locale) del secondo punto da stPt
    NSRect aRect ;
    // trasformo le distanze in coordinate locali
    localSize = dim ;
    // imposto il nuovo punto iniziale di disegno
    drawPoint = stPt ;
    // l'angolo di rotazione non cambia
    // aggiorno la trasformazione delle coordinate
    [ self calcLocalTranform ];
    // rettangolo che costuisce l'elemento
    aRect = NSMakeRect( 0, 0, localSize.width, localSize.height );
    // per mia convenienza conservo il percorso
    [ self setTheDrawPath: [NSBezierPath bezierPathWithRect: aRect ] ];
    // metto a posto il rettangolo che lo contiene
    [ self buildHdlList ] ;
```

```
    [ ownerView setNeedsDisplay: YES ];
}
```

Il metodo è piuttosto semplice e lineare: aggiusta le variabili interne, calcolando le nuove trasformazioni, predispone poi il percorso `theDrawPath`. Il percorso rappresenta sempre il rettangolo nel SdR dell'elemento, ed è quindi costruito a partire da un rettangolo posizionato a (0,0) e di dimensione opportuna. Scomparsa la variabile d'istanza `elemLimit`, il cui metodo accessor forzava il calcolo degli handle, occorre esplicitamente calcolare le coordinate degli handle. Anche in questo caso, gli handle possiedono corrdinate nel SdR dell'elemento, per cui il metodo corrispondente diventa:

```
- (void)
buildHdlList
{
    // gli handle sono genericamente ai quattro estremi del
    // rettangolo limite e nei quattro punti mediani
    hdlList[0] = NSMakePoint( 0, 0) ;
    hdlList[2] = NSMakePoint( localSize.width, 0) ;
    hdlList[4] = NSMakePoint( localSize.width, localSize.height) ;
    hdlList[6] = NSMakePoint( 0, localSize.height ) ;
    hdlList[1].x = (hdlList[0].x + hdlList[2].x ) * 0.5 ;
    hdlList[1].y = (hdlList[0].y + hdlList[2].y ) * 0.5 ;
    hdlList[3].x = (hdlList[2].x + hdlList[4].x ) * 0.5 ;
    hdlList[3].y = (hdlList[2].y + hdlList[4].y ) * 0.5 ;
    hdlList[5].x = (hdlList[4].x + hdlList[6].x ) * 0.5 ;
    hdlList[5].y = (hdlList[4].y + hdlList[6].y ) * 0.5 ;
    hdlList[7].x = (hdlList[6].x + hdlList[0].x ) * 0.5 ;
    hdlList[7].y = (hdlList[6].y + hdlList[0].y ) * 0.5 ;
}
```

E adesso, attenzione al metodo di disegno del rettangolo:

```
- (void)
drawElement: (NSRect) inRect
{
    // per salvare il contesto
    NSGraphicsContext *context = [NSGraphicsContext currentContext];
    [context saveGraphicsState];
    [ localTF concat ];
    // disegno del testo
    if ( cceIsStroked )
    {
        // imposto il colore del contorno
        [ cceLineColor set ];
        // imposto lo spessore del contorno
        [ theDrawPath setLineWidth: [ self cceLineWidth ]];
        // disegno il contorno
        [ theDrawPath stroke ];
    }
    // se il rettangolo e' pieno
    if ( cceIsFilled )
    {
        // imposto il colore del pieno
        [ cceFillColor set ];
        // riempi tutto
        [ theDrawPath fill ];
    }
    [ self drawHandles ] ;
    // ripristino del contesto
```

```
    [context restoreGraphicsState];
}
```

La prima istruzione si preoccupa di memorizzare il contesto grafico corrente della view; infatti l'operazione successiva lo modifica, cambiando il SdR concatenando la trasformazione relativa all'elemento. A questo punto, tornano le consuete operazioni di disegno, che avvengono nel SdR dell'elemento. Alla fine, sono disegnati gli handle ed è ripristinato il contesto grafico precedente (cioé, ritorna il SdR della view). Se così non fosse, il SdR utilizzato per il disegno successivo sarebbe ancora quello dell'elemento rettangolare (cosa che invece non va bene).

In effetti, tutti i metodi di disegno dei vari elementi avranno un prologo (salvataggio del contesto e impostazione del nuovo SdR) ed un epilogo (disegno degli handle e ripristino del contesto), per cui ho diviso il lavoro in due passi. La classe `CCE_BasicForm` si occupa del prologo e dell'epilogo, ed ogni elemento delle proprie operazioni specifiche; di conseguenza in `CCE_BasicForm` si trova il metodo generale di disegno:

```
- (void)
drawElement: (NSRect) inRect
{
    // per salvare il contesto
    NSGraphicsContext *context = [NSGraphicsContext currentContext];
    [context saveGraphicsState];
    [ localTF concat ];
    [ self specificDrawing: inRect ];
    [ self drawHandles ] ;
    // ripristino del contesto
    [context restoreGraphicsState];
}

- (void)
specificDrawing: (NSRect) inRect
{
    // da sovrascrivere
}
```

All'interno delle varie sottoclassi, i metodi specifici; ad esempio per il rettangolo, l'ormai consueta sequenza di istruzioni:

```
- (void)
specificDrawing: (NSRect) inRect
{
    // disegno del contorno
    if ( cceIsStroked )
    {
        ...
    }
    // se il rettangolo e' pieno
    if ( cceIsFilled )
    {
        ...
    }
}
```

LE ALTRE CLASSI DI ELEMENTI

Gli elementi `CCE_Circle` e `CCE_Text` sono stati modificati; con l'intento di semplificarli sono diventati sottoclassi di `CCE_Rect` (invece che direttamente di `CCE_BasicForm`); in effetti per i cerchi le cose sono piuttosto semplici. Per l'inizializzazione:

```
- (id)
initWithId: (int) ident inView: (CoverView*) cv withRect: (NSRect) aRect
andAngle: (float) angle
{
     self = [ super initWithId: ident inView: cv withRect: aRect andAngle:
angle ] ;
    if ( self )
    {
        // aggiusto il path perche' sia un cerchio
        [ self setTheDrawPath: [NSBezierPath bezierPathWithOvalInRect:
            NSMakeRect( 0, 0, localSize.width, localSize.height) ]];
    }
    return self ;
}
```

Si utilizza l'inizializzazione propria di un rettangolo, per poi rimpiazzare brutalmente il percorso con un ovale invece che lasciare il rettangolo standard. La stessa cosa succede per la costruzione per punto e dimensione:

```
- (void)
buildMeFromPt: (NSPoint) stPt withSize: (NSSize) dim
{
    NSRect aRect ;
    [ super buildMeFromPt: stPt withSize: dim ] ;
    aRect = NSMakeRect( 0, 0, localSize.width, localSize.height );
     [ self setTheDrawPath: [NSBezierPath bezierPathWithOvalInRect: aRect ]
];
}
```

Addirittura, il metodo di disegno non va riscritto, in quanto l'ovalità dell'elemento è racchiusa dal percorso.

Più complicata, ma solo per la gestione delle classi relative al testo, la classe `CCE_Text`. L'inizializzazione diventa:

```
- (id)
initWithId: (int) ident inView: (CoverView*) cv
     withAttributedString: (NSAttributedString *) theText inRect: (NSRect)
aRect withAngle: (float) angle
{
     self = [ super initWithId: ident inView: cv withRect: aRect andAngle:
angle] ;
    if ( self )
    {
        NSTextStorage * loctextStorage ;
        NSLayoutManager *layoutManager;
        NSTextContainer *textContainer;
        // layout gestisce come il testo e' rappresentato
         loctextStorage = [[[NSTextStorage alloc] initWithAttributedString:
theText] autorelease ];
        layoutManager = [[[NSLayoutManager alloc] init] autorelease ];
        [loctextStorage addLayoutManager:layoutManager];
        // indica dove il testo e' rappresentato
```

```
        textContainer = [[[NSTextContainer alloc] init] autorelease ];
        // dovrei specificare un NSSize per il textcontainer
        [ theTextCont setContainerSize: localSize ];
        [layoutManager addTextContainer:textContainer];
        // assegno il textStorage
        [ self setTextStorage: loctextStorage ];
        theTextCont = textContainer ;
    }
    return self ;
}
```

Il metodo per il disegno è bellissimo:

```
- (void)
specificDrawing: (NSRect) inRect
{
    NSRange glyphRange ;
    // recupero il primo layout
      NSLayoutManager *layoutManager = [ [ textStorage layoutManagers ]
objectAtIndex: 0] ;
    [ super specificDrawing: inRect ] ;
    // disegno del testo
    [ theTextCont setContainerSize: localSize ];
    // recupero il range del testo
    glyphRange = [layoutManager glyphRangeForTextContainer: theTextCont ];
          [   layoutManager   drawGlyphsForGlyphRange:glyphRange   atPoint:
NSMakePoint(0,0) ];
}
```

Si sfruttano brutalmente le capacità della superclasse `CCE_Rect` per il disegno del rettangolo di contorno ed il riempimento, e si aggiungono le istruzioni specifiche per il testo.

Anche la classe `CCE_Line` è stata ovviamente modificata. Tuttavia, le diversità rispetto alla classe `CCE_Rect` si sono ridotte, anche se la maggior parte delle modifiche appariranno più avanti. Ci sono i soliti tre metodi di inizializzazione, costruzione e disegno:

```
- (id)
initWithId: (int) ident inView: (CoverView*) cv fromPt: (NSPoint) stPt
toPt: (NSPoint) endPt
{
    self = [ super initWithId: ident inView: cv ] ;
    if ( self )
    {
        numOfHdl = 2 ;
           [ self buildMeFromPt: stPt withSize: NSMakeSize(endPt.x-stPt.x,
endPt.y-stPt.y) ];
        [ self setCceIsStroked: TRUE ];
    }
    return self ;
}
```

Anche qui, si sfrutta la superclasse ed il metodo di costruzione, sul quale occorre spendere due parole.

```
- (void)
buildMeFromPt: (NSPoint) stPt withSize: (NSSize)newSize
```

```
{
    NSBezierPath *path = [NSBezierPath bezierPath];
    // calcola la lunghezza del segmento
                float    c3    =    sqrt(    newSize.width*newSize.width    +
newSize.height*newSize.height) ;
     // calcolo l'angolo formato dal segmento col sistema di riferimento
della
    // finestra, e lo trasformo in numero tra 0 e 360
      [ self setRotAngle: (180 * atan2( newSize.height, newSize.width) /
M_PI) ] ;
    // il punto base e' il primo punto
    drawPoint = stPt ;
    [ self calcLocalTranform ];
    // la dimensioen della linea e' la sua lunghezza e zero.
    localSize = NSMakeSize( c3, 0) ;
    // costruisco il percorsoin coordinate locali
    [path moveToPoint: NSMakePoint(0,0) ];
    [path lineToPoint: NSMakePoint(c3,0) ];
    // lo assegno
    [ self setTheDrawPath: path ];
    // costruisco gli handle
    [ self buildHdlList ] ;
    [ ownerView setNeedsDisplay: YES ];
}
```

Per rappresentare un segmento, ho deciso di mantenere la stessa filosofia: spostamento in punto origine e rotazione. Di più: la linea riposa sempre sulla retta delle ascisse del SdR locale. Per costruire dunque la linea, mi occorre calcolare la sua lunghezza: è la radice quadrata della somma dei quadrati costruiti sulla variabile `NSSize`. Infatti `NSSize` contiene nei suoi campi `width` ed `height` la distanza (nel SdR della view, attenzione) tra i due punti costituenti il segmento. L'angolo di rotazione, con un po' di trigonometria, è dato dall'arcotangente del rapporto `height/width`. Per mantenere correttamente il segno e quindi il valore dell'angolo, uso la funzione `atan2(,)`, che si trova allo scopo nelle librerie matematiche standard del C. Imposto l'angolo di rotazione (dopo averlo convertito in gradi: `atan2(,)` restituisce radianti), il nuovo punto origine e calcolo le matrici di trasformazione. Poi costruisco il percorso con una linea che va dall'origine fino alla lunghezza prevista, sull'asse delle ascisse.

Devo costruire gli handle:

```
- (void)
buildHdlList
{
    // due soli handle, che sono i punti estremi
    hdlList[0] = NSMakePoint( 0, 0 ) ;
    hdlList[1] = NSMakePoint( localSize.width, 0) ;
}
```

E poi, il metodo di disegno:

```
- (void)
specificDrawing: (NSRect) inRect
{
    // imposto il colore
    [ cceLineColor set ];
    // imposto lo spessore della linea
    [ theDrawPath setLineWidth: [ self cceLineWidth ]];
```

```
    // disengo la linea
    [ theDrawPath stroke ];
}
```

Questa nuova struttura degli elementi ha un interessante impatto sul resto dei metodi di `CCE_BasicForm`; questi ne risulteranno semplificati, e con meno differenze tra i vari elementi.

CREAZIONE E MODIFICA

Il metodo per gestire la creazione di un elemento (rettangolare in prima battuta) è presente all'interno della classe `CCE_BasicForm`.

```
- (BOOL)
handleCreation: (NSEvent *)theEvent
{
    CoverView * view = [self ownerView] ;
    // recupero la coordinata locale
    NSPoint     point1 = [ view convertPoint:
        [theEvent locationInWindow] fromView:nil];
    NSPoint     point2 ;
    BOOL         shiftKeyPress ;
    NSSize        tmpSize ;
    // piglio il punto piu' vicino sulla griglia (se necessario)
    point1 = getNearestSnapPoint( point1, view);
    // inizializzo l'aspetto dell'oggetto (praticamente vuoto)
    [ self buildMeFromPt: point1 withSize: NSMakeSize(1,1) ] ;
    // continuo a ciclare in attesa di un mouseup
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
        theEvent = [[view window] nextEventMatchingMask:
            (NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
         shiftKeyPress = (([theEvent modifierFlags] & NSShiftKeyMask) ? YES
: NO);
        // recupero la coordinata della finestra
                point2  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
        point2 = getNearestSnapPoint( point2, view);
        tmpSize = NSMakeSize(point2.x-point1.x, point2.y-point1.y) ;
        if ( shiftKeyPress )
        {
            // costringo il rettangolo ad essere quadrato
            float delta1 = tmpSize.width ;
            float delta2 = tmpSize.height ;
            float    dirx = 1, diry = 1 ;
            if ( delta1 < 0 )
            {
                delta1 = - delta1 ; dirx = -1 ;
            }
            if ( delta2 < 0 )
            {
                delta2 = - delta2 ; diry = -1 ;
            }
            // piglio la dimensione piu' corta
            if ( delta1 < delta2 )
                tmpSize.height = diry * delta1 ;
            else
                tmpSize.width = dirx * delta2 ;
        }
```

```
        // aggiusto l'aspetto dell'oggetto    (vedi nota)
        [ self buildMeFromPt: point1 withSize: tmpSize ] ;
        // forzo un ridisegno della finestra
        [ view setNeedsDisplay: YES ];
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
    // arrivato qui, ho finito il tutto
    return YES;
}
```

Le modifiche rispetto alla versione precedente si trovano ovviamente nel fatto che è cambiato il metodo di costruzione, che ha adesso bisogno di una grandezza `NSSize`. C'è una particolarità da notare, ovvero l'invocazione del metodo `buildMeFromPt:withSize:`. A rigor di programmazione, questo metodo richiede come primo argomento le coordinate di un punto nel SdR della finestra e, come secondo argomento, le dimensioni (larghezza ed altezza) dell'elemento, nel SdR proprio dell'oggetto. Questa istruzione invece fornisce la variabile `tmpSize`, che esprime dimensioni nel SdR della view. Tuttavia, trattandosi di prima creazione dell'oggetto, l'angolo di rotazione è zero; ciò fa si che le dimensioni siano uguali nei due sistemi di riferimento.

La versione per `CCE_Line` è del tutto uguale, e non avrebbe bisogno di sovrascrittura se non fosse per poter tracciare correttamente linee orizzontali e verticali quando è premuto il tasto Shift:

```
...
    if ( shiftKeyPress )
    {
        float delta1 = tmpSize.width ;
        float delta2 = tmpSize.height ;
        float    dirx = 1, diry = 1 ;
        if ( delta1 < 0 )
        {
            delta1 = - delta1 ; dirx = -1 ;
        }
        if ( delta2 < 0 )
        {
            delta2 = - delta2 ; diry = -1 ;
        }
        // piglio la dimensione piu' corta
        if ( delta1 < delta2*0.2 )
            tmpSize.width = 0 ;
        else if ( delta2 < delta1*0.2 )
            tmpSize.height = 0 ;
        else if ( delta1 < delta2 )
            tmpSize.height = diry * delta1 ;
        else
            tmpSize.width = dirx * delta2 ;
    }
...
```

Nel caso della linea, tuttavia, il metodo `buildMeFromPt:withSize:` richiede come dimensioni grandezze espresse nel SdR della view. A maggior ragione, non devono essere effettuate trasformazioni di sorta.

Per il ri-dimensionamento degli elementi, è stato revisionato il metodo seguente per `CCE_BasicForm`:

```
- (BOOL)
handleResize: (NSEvent *)theEvent
    movingHandle: (short) hdlIdx
{
    NSPoint point ;
    NSSize newSize ;
    CoverView * view = [self ownerView] ;
    // indice dello handle che tengo fisso
    static short fixedHandle[8] = { 4, 4, 6, 0, 0, 0, 2, 4 };
    // indice dello handle che devo muovere (opposto)
    static short movingHandle[8] = { 0, 0, 2, 4, 4, 4, 6, 0 };
    // identificatore delle dimensioni che posso modificare:
    // con 2, solo width, con 1 solo height, con 3, w e h
    static short movingPoints[8] = { 3, 1, 3, 2, 3, 1, 3, 2 };
    short movingCoord = movingPoints[ hdlIdx ];
    NSPoint fixedPoint = hdlList[ fixedHandle[ hdlIdx ] ];
    NSPoint movingPoint = hdlList[ movingHandle[hdlIdx] ];
    // se e' locked, non se ne fa nulla
    if ( cceIsLocked ) return YES ;
    // trasformo i punti fissi e mobili nelle coordinate della finestra
    fixedPoint = [ localTF transformPoint: fixedPoint ];
    movingPoint = [ localTF transformPoint: movingPoint ];
    // ridisegno l'oggetto con un nuovo sistema di riferimento
    newSize = [ reverTF transformSize:
         NSMakeSize(movingPoint.x-fixedPoint.x, movingPoint.y-fixedPoint.y)
] ;
    [ self buildMeFromPt: fixedPoint withSize: newSize ] ;
     // ho ricostruito l'oggetto uguale a se stesso; tuttavia, potrebbe
essere
    // cambiato il punto base e la rotazione
    // mi preparo lo undo
    [[self undoManager] setActionName: @"BasicForm -> handleResize"];
          [[[self   undoManager]   prepareWithInvocationTarget:   [ownerView
theElements]]
        resizeElem: self withSize: localSize ];
    // continuo a ciclare in attesa di un mouseup
    do {
        // recupero il prossimo evento, che sia un movimento del mouse
        // oppure un mouseup (che finisce il tutto)
                                    theEvent     =     [[view     window]
nextEventMatchingMask:(NSLeftMouseDraggedMask | NSLeftMouseUpMask)];
        // recupero la coordinata della finestra
                point  =  [view  convertPoint:[theEvent  locationInWindow]
fromView:nil];
        point = getNearestSnapPoint( point, view);
        // ho una nuova dimensione dell'oggetto
        newSize = [ reverTF transformSize:
            NSMakeSize(point.x-fixedPoint.x, point.y-fixedPoint.y) ] ;
        // se il caso, lascio invariata una delle due dimensioni
        if ( movingCoord == 1)
            newSize.width = localSize.width ;
        else if ( movingCoord == 2)
            newSize.height = localSize.height ;
        // ricostruisco l'oggetto con una nuova dimensione
        [ self buildMeFromPt: fixedPoint withSize: newSize] ;
    }
    while ( [theEvent type] != NSLeftMouseUp) ;
    // arrivato qui, ho finito il tutto
    return YES;
}
```

I vettori `fixedHandle` e `movingHandle` sono cambiati, perché è cambiato l'ordine con cui sono calcolati gli handle, ma il concetto di handle fisso e handle mobile è rimasto

intatto. È invece cambiato il significato dei codici raccolti in `movingPoints`; prima permettevano modifiche nelle ascisse e/o nelle ordinate, adesso significato possibilità di modifica nella larghezza e/o altezza dell'elemento. Attenzione adesso che i punti `fixedPoint` e `movingPoint` sono espressi nel SdR dell'elemento, e vanno quindi convertiti nel SdR della view usando la trasformazione `localTF`. C'è poi una istruzione `buildMeFromPt:withSize:` di costruzione dell'elemento che può apparire superflua (in effetti, si deve ancora far nulla, per cui non è ancora cambiato nulla). In realtà, l'elemento è costruito nuovamente per cambiare (se il caso, ma lo è sette volte su otto) il SdR, incentrandolo sul punto che rimane fisso; il ridisegno serve anche per fissar le idee preparando l'istruzione di Undo; questa avrà come unico parametro la dimensione dell'elemento, contando sul fatto che le operazioni successive di dimensionamento non lo cambiano più.

Finalmente si può partire col ciclo di gestione dei movimenti del mouse. Quando si ottiene il nuovo punto `point`, si calcolano le nuove dimensioni dell'oggetto; la grandezza `newSize` va calcolata tenendo conto della trasformazione da SdR della view (in cui sono espresse le coordinate di `point` e `fixedPoint`) e quelle dell'elemento (necessarie per il successivo metodo `buildMeFromPt:withSize:`). La limitazione del dimensionamento dell'oggetto su larghezza e/o altezza è svolta in maniera molto facile, riportando se necessario la dimensione precedente al posto di quella appena calcolata.

Il metodo per `CCE_Line` è molto simile, e non lo riporto qui, mantenendo esattamente gli stessi concetti, con le opportune modifiche.

Capitolo 052 - Rotazioni e Rivoluzioni, parte 2

Seconda parte di un lungo capitolo, in cui si completa la storia della rotazioni e si cambia anche la strategia di gestione delle notifiche.

Sorgenti: documentazione Apple.

Prima stesura: 16 agosto 2004.

GLI ALTRI METODI DI CCE_BASICFORM

La classe `CCE_BasicForm` ha bisogno di molti altri metodi per la gestione degli elementi. il fatto interessante è che i metodi descritti in questo paragrafo vanno bene per tutti i tipi di elemento, e non richiedono di versioni particolari per classi specifiche (come invece succedeva in precedenza); fa sempre eccezione la classe `CCE_ElemGroup`, che ha metodi specifici per la gestione di gruppi di elementi (che non sono cambiati).

Parto dal metodo che determina se il clic del mouse è avvenuto sopra l'elemento. è cambiato profondamente:

```
- (BOOL)
checkClickOnMe:(NSPoint)point
{
    float ev = cceLineWidth + 2;
    // trasformo il punto in coordinate locali
    NSPoint locPt = [ reverTF transformPoint: point ];
    BOOL    insideX = NO, insideY = NO ;
    // qui faccio un po' di confusione, in quanto PointInRect
    // va in confusione con dimensioni negative o nulle
    if ( localSize.width >= 0 )
    {
        // se la dimensione e' positiva ...
        // controllo se la coordinata del punto e' compresa
        // tra un Zero e la lunghezza (con tolleranza...)
        if ( (-ev < locPt.x) && (locPt.x < (localSize.width + ev)) )
            insideX = YES ;
    }
    else
    {
        // se la dimensione e' negativa...
        // controllo se la coordinata del punto e' compresa tra la
        // lunghezza (negativa) e zero (con tolleranza)
        if ( ((- ev + localSize.width) < locPt.x) && (locPt.x < ev) )
            insideX = YES ;
    }
    // faccio lo stesso per l'altra dimensione
    if ( localSize.height >= 0 )
    {
        if ( (-ev < locPt.y) && (locPt.y< (localSize.height + ev)) )
            insideY = YES ;
    }
```

```
    else
    {
        if ( ((- ev + localSize.height) < locPt.y) && (locPt.y < ev) )
            insideY = YES ;
    }
    // il punto sta dentro se e' compreso in entrambe le coordinate
    return ( insideX && insideY );
}
```

La prima cosa da fare è convertire il punto di clic in coordinate locali. Poi, si tratta di capire se questo punto si trova all'interno del rettangolo che inscrive l'elemento grafico. Tuttavia, a causa del fatto che la funzione `NSPointInRect` rimane confusa da rettangoli con dimensioni nulle e dimensioni negative, ho sbrogliato la matassa facendo esplicito riferimento ai valori in esame. Ogni controllo avviene utilizzando una tolleranza sul posizionamento pari allo spessore della linea più altri due pixel, in modo che sia facilitato il clic sopra elementi di infimo spessore. Giova notare che il metodo esposto funziona anche per elementi di classe `CCE_Line`, senza bisogno di una versione specifica del meotdo e senza bisogno di calcoli particolari di distanza di un punto da una retta.

Per capire se il cli è avvenuto sopra un handle, la cosa è ancora semplice:

```
- (short)
clicIntoHandle: (NSPoint) point
{
    int i;
    // ricavo le coordinate locali del punto
    point = [ reverTF transformPoint: point ];
    for ( i = 0 ; i < numOfHdl ; i ++ )
    {
        // se il punto designato e lo handle sono abbastanza vicini
        float a1 = hdlList[i].x - point.x ;
        float a2 = hdlList[i].y - point.y ;
        if ( a1 * a1 + a2 * a2 <= 4 )
        {
            return ( i );
        }
    }
    return ( -1 ) ;
}
```

Rispetto alla versione precedente, c'è la trasformazione del punto in coordinate locali.

C'è poi il metodo che controlla se l'elemento si trova all'interno di un rettangolo, che si presume derivi da una operazione di selezione multipla da parte dell'utente. Non avevo voglia di fare cose sofisticate, ed ho quindi scritto:

```
- (void)
selectMeIfInRect: (NSRect) selRect addingToSelez: (BOOL) add2sel
{
    BOOL    inrect ;
    // calcolo il rettangolo che circoscrive il percorso dell'elemento
     NSRect limRect = [ [ localTF transformBezierPath: theDrawPath ] bounds
] ;
    // lo allargo di un pelo per tenere conto di rettangoli nulli
    // (che non fanno funzionare le cose)
    limRect = NSInsetRect(limRect, -1, -1 );
```

```
    // vedo se il rettangolo di selezione comprende l'elemento
    inrect = NSContainsRect( selRect, limRect ) ;
    // se non devo aggiungere alla selezione
    if ( ! add2sel )
        ... // come in precedenza
}
```

Il rettangolo di selezione è espresso in coordinate della view. Determino quindi il rettangolo che circoscrive il percorso, trasformando prima il percorso nel SdR della view, e poi calcolandone gli estremi, ottenendo il rettangolo `limRect`. Aggiungo una certa tolleranza al rettangolo (anche per gestire correttamente percorsi che rappresentano linee orizzontali e verticali, che danno luogo a rettangoli con una dimensione nulla), e poi verifico l'inclusione colla classica funzione `NSContainsRect`. Ancora una volta, lo stesso metodo va bene per tutte le classi derivate.

Ho poi collassato i due metodi di movimento in un solo metodo; in precedenza erano infatti presenti due diversi metodi per muovere un elemento, la cui differenza risiedeva nel fatto che in un caso il movimento era condizionato dall'essere l'elemento selezionato o meno. Ho mantenuto la distinzione fornendo un opportuno parametro al metodo:

```
- (void)
moveMeBy: (NSPoint) distance onlyIfSelected: (BOOL) selOnly
{
    // se selOnly e' Falso, muovo sicuro
    // se invece e' vero, solo se non locked e selected
    if ( (selOnly == NO) || ((! cceIsLocked) && cceIsSelected) )
    {
        // gestione dello undo del movimento
        [[self undoManager] setActionName: @"BasicForm -> moveMeBy"];
             [[[self  undoManager]  prepareWithInvocationTarget:  [ownerView
theElements]]
            moveElem: self by: NSMakePoint(-distance.x, -distance.y)];
        // effettuo il movimento vero e proprio: sposto il punto base
        drawPoint.x += distance.x ;
        drawPoint.y += distance.y ;
        // aggiorno la trasformazione affine
        [ self calcLocalTranform ];
        // ricosctruisco gli handle
        [ self buildHdlList ] ;
        // devo sicuramente ridisegnare qualcosa
        [ [ self ownerView ] setNeedsDisplay: YES ];
    }
}
```

Se il valore del parametro `selOnly` è Falso (NO), il movimento avviene comunque; se invece il parametri è Vero (YES), il movimento avviene solo se l'elemento è selezionato e non è bloccato. C'è da notare come avviene il movimento dell'elemento: è sufficiente modificare le coordinate dell'origine del SdR dell'elemento e ricalcolare le trasformazioni (e gli handle).

Infine, ci sono i metodi utilizzati per ottenere le coordinate degli estremi; in precedenza, i metodi erano molto semplici ed utilizzano a man bassa il fatto che il rettangolo `elemLimit` già forniva tutte le informazioni richiesta. Ora che `elemLimit` non esiste più, occorre calcolarlo esplicitamente. Ad esempio:

```
- (NSNumber*)
```

```
getElemTop
{
    // piglio il percorso e lo trasformo nelle coordinate della finestra
      // da qui, ricavo il rettangolo circoscritto, e poi l'estremo di
interesse
    NSRect limR = [ [ localTF transformBezierPath: theDrawPath ] bounds ] ;
    return ( [ NSNumber numberWithFloat: limR.origin.y] );
}
```

ROTAZIONI

Posso finalmente discutere come effettuare rotazioni di elementi. In effetti, dopo tutto il lavoro precedente, la cosa è piuttosto semplice. Liquido subito il metodo della classe `CCE_ElemGroup` per gestire le rotazioni degli elementi selezionati, in maniera incrementale o assoluta:

```
- (void)
rotateElemAngle: (float) angle absolute: (BOOL) absInc
{
    int        i , numElem ;
    CCE_BasicForm * elem ;

    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        if ( [ elem cceIsLocked] || ( ! [ elem cceIsSelected] ))
            continue ;
        if ( absInc )
                [ elem rotateMeAtAngle: angle ] ;
        else    [ elem rotateMeAtAngle: ([ elem rotAngle] + angle) ] ;
    }
}
```

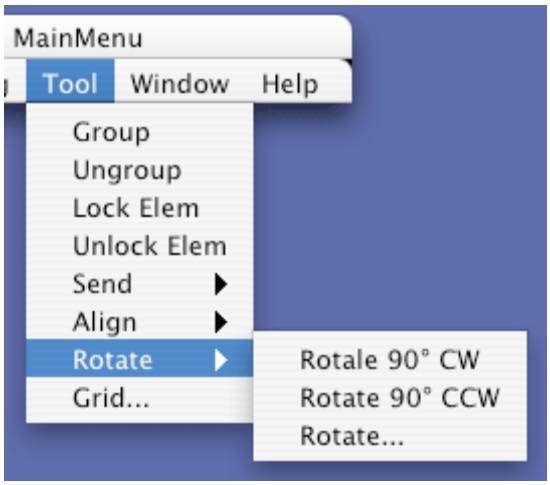
Questo è il metodo chiamato sia dalle voci di menu che richiedono incrementi di 90 gradi in senso orario o antiorario, sia dalla prossimamente descritta palette di gestione delle rotazioni. Il metodo principale è proprio della classe `CCE_BasicForm`, va bene per tutte le classi di elementi ed è così composto:

```
- (void)
rotateMeAtAngle: (float) angle
{
    NSAffineTransform * transform;
    // preparo lo undo
    [[self undoManager] setActionName: @"BasicForm -> rotateMeAtAngle"];
          [[[self   undoManager]   prepareWithInvocationTarget:   [ownerView
theElements]]
        rotateElem: self byAngle: rotAngle ];
    // parto dalla trasformazione corrente
    transform = [ [ NSAffineTransform alloc] initWithTransform: localTF ] ;
    // mi spostoal centro dell'elemento
    [ transform translateXBy: localSize.width*0.5 yBy: localSize.height*0.5
];
    // eseguo la rotazione residua sull'oggetto
    [ transform rotateByDegrees: (angle - rotAngle) ];
    // ritorno nel punto base
              [    transform    translateXBy:    -localSize.width*0.5    yBy:
```

```
-localSize.height*0.5 ];
    // bisogna aggiornare la variabili interne
    [ self setRotAngle: angle ] ;
    // ma guarda dov'e' finita l'origine...
    drawPoint = [ transform transformPoint: NSMakePoint(0, 0) ];
    // calcolo ed assegno le varie trasformazioni
    [ self setLocalTF: transform ];
    [ transform invert ];
    [ self setReverTF: transform ];
    // aggiorno gli handle
    [ self buildHdlList ] ;
    [ [ self ownerView ] setNeedsDisplay: YES ];
}
```

Dopo aver congelato la situazione corrente per il meccanismo di undo, procedo con le operazioni di rotazione. Ora, voglio che la rotazione avvenga non tanto attorno al punto origine, ma attorno al punto centrale dell'elemento. Per fare questo, devo costruire una corretta matrice di trasformazione, eseguendo le trasformazioni elementari nel giusto ordine. Parto dalla trasformazione corrente: poi, mi sposto nel centro dell'elemento (mi sposto di metà larghezza e metà altezza); in questo punto, eseguo la rotazione residua. Infine, ritorno nel punto origine, spostandomi nuovamente di metà larghezza e metà altezza. Nel compiere queste operazioni, l'origine del SdR dell'elemento si è spostata: la recupero, nelle coordinate della view, trasformando il nuovo punto origine (quello di coordinate nulle) attraverso la trasformazione corrente.

Raccontata così, sembra piuttosto semplice ed ovvia, ma la cosa ha richiesto molto lavoro (non è che le trasformazioni affini mi fossero chiare fin dall'inizio), esperimenti, rifacimenti, molti disegnini su carta per capire cosa succedendo. Però mi pare che il risultato sia semplice ed anche, nel suo piccolo, elegante.

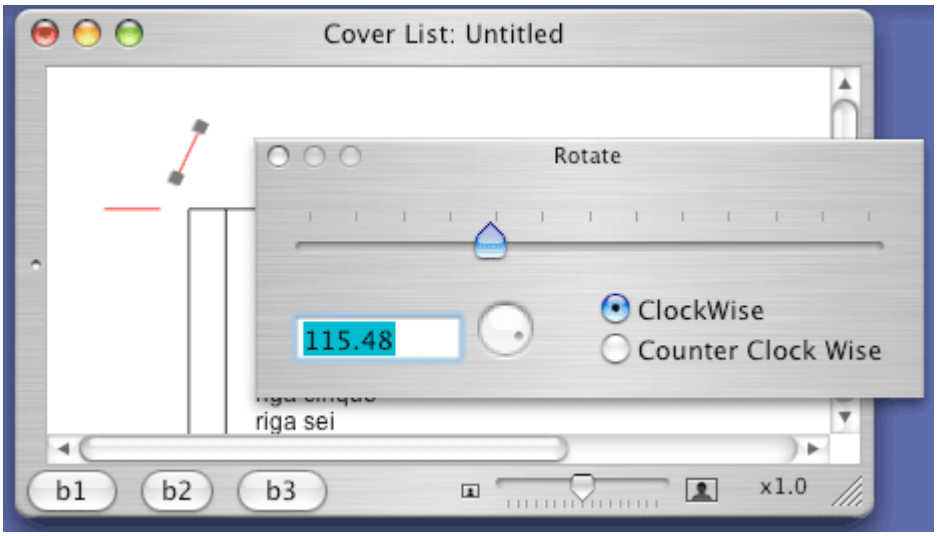


Rimane da discutere come comandare le operazioni di rotazione. Ho previsto almeno due modalità. La prima è quella classica da menu: ci sono due voci per eseguire rotazioni orarie e antiorarie di novanta gradi. Questi comandi sono incrementali, nel senso che la rotazione di novanta gradi si aggiunge all'angolo corrente. Di conseguenza il metodo che risponde a questi menu è così fatto (si trova nella classe `CoverView`):

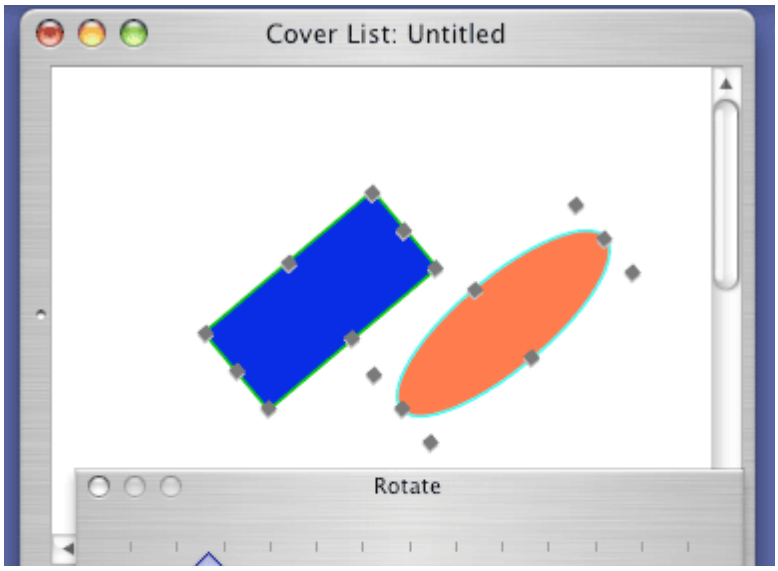
```
- (void)
rotateElem: (id) sender
{
    float        angle = ([ sender tag] == 90) ? +90.0 : -90.0 ;
```

```
    [ theElements rotateElemAngle: angle absolute: NO ] ;
    [ [NSNotificationCenter defaultCenter]
        postNotificationName:@"CoverViewChangedSelection"
        object: self ];
}
```

Ho attributo alle due voci di menu lo stesso metodo come action, e distinguo il comando effettivo attraverso i tag 90 e 91 che ho loro attribuito.



C'è poi una terza voce che apre un pannello dove è possibile comandare rotazioni arbitrarie, secondo una modalità assoluta. In questo pannello mi sono sbizzarrito con tre possibilità: si può stabilire la rotazione attraverso uno slider orizzontale, uno slider circolare ed impostando direttamente il valore attraverso un campo di testo. Non appena l'utente modifica uno di questi tre controlli, si ha subito l'effetto sugli elementi selezionati, che si dispongono secondo l'angolo indicato. Al di là di alcune imperfezioni ed idiosincrasie nell'interfaccia (non so bene come gestire il cambiamento tra rotazione oraria ed antioraria, o se gestirla del tutto, visto che l'angolo spazia in realtà in modalità assoluta tra 0 e 360 gradi; non so bene come comportarmi quando ci sono più elementi selezionati: adesso assumono tutti lo stesso angolo, cosa giusta dal punto di vista teorica, ma bruttissima a vedersi; cose del genere), la cosa sembra funzionare piuttosto bene, con effetti spettacolari.



La finestra è gestita dalla nuova classe `RotateWinCtl`, secondo le procedure ormai standard di molte finestre del genere. Una operazione su uno dei tre controlli (i due slider ed il campo testo) attiva la seguente azione:

```
- (IBAction)
updateSliders:(id)sender
{
    float value = [ sender floatValue ];

    [ crcSlider setFloatValue: value ] ;
    [ linSlider setFloatValue: value ] ;
    [ valField setFloatValue: value ] ;
    [ self execRotation: sender ];
}
```

Una generica operazione sul pannello di rotazione quindi provoca l'esecuzione del seguente metodo:

```
- (IBAction)
execRotation:(id)sender
{
    CoverView * cv = [AppDelegate getTheDrawView ] ;
    float        angle1, angle2 ;
    // non dovrebbe mai succedere...
    if ( cv == nil ) return ;

    angle1 = [ linSlider floatValue ] ;
    angle2 = ([ [ dirRadioBtn selectedCell] tag ] == 95) ? angle1 : -angle1
;
    [ [ cv theElements] rotateElemAngle: angle2 absolute: YES ] ;
    // dopo la rotazione, occorre aggiornare la selezione corrente
    // nota che l'oggetto di riferimento e' la coverView
    [ [NSNotificationCenter defaultCenter]
            postNotificationName:@"CoverViewChangedSelection"
            object: cv ];
}
```

Lasciate per il momento perdere la misteriosa istruzione che parte da `AppDelegate`; il metodo recupera l'angolo di rotazione, lo aggiusta a seconda della direzione di rotazione (ancora una volta discriminata attrverso l'attribuzione di appositi tag ai radio button), e poi comanda l'esecuzione della rotazione. Come nel metodo `rotateElem:` visto poco sopra, il metodo termina inviando una notifica; in questo modo le finestre che dipendono dalla selezione corrente per le loro funzionalità (la finestre della informazioni relative ad un elemento, ad esempio), possono aggiornare il loro contenuto di conseguenza.

NUOVA GESTIONE DELLE NOTIFICHE

L'ultimo argomento del capitolo non ha molto a che fare con le rotazioni, ma è

solamente un diverso meccanismo di gestione delle notifiche. Fino a questo momento, ogni volta che costruivo un pannello le cui funzioni dipendevano dalla finestra principale (ad esempio: informazioni di un elemento di catalogo; informazioni di un elemento grafico; il pannello di impostazione della griglia, eccetera), mi trovavo costretto a sottoscrivere almeno un paio di notifiche per gestire in modo congruente una variabile d'istanza che tenesse traccia della finestra principale; la variabile serviva per poter poi operare correttamente all'interno di questa finestra. Pigliamo ad esempio la palette per effettuare allineamenti e centraggi; avevo bisogno di almeno tre metodi per tenere aggiornata la variabile `theDrawView`:

```
- (void)
windowDidLoad
{
    [super windowDidLoad];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowChanged:)
        name:NSWindowDidBecomeMainNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowResigned:)
        name:NSWindowDidResignMainNotification object:nil];
    [self setMainWindow:[NSApp mainWindow]];
}

- (void)
mainWindowChanged: (NSNotification *)notification
{
    [self setMainWindow:[notification object]];
}

- (void)
mainWindowResigned: (NSNotification *)notification
{
    [self setMainWindow:nil];
}

- (void)
setMainWindow: (NSWindow *) mainWindow
{
    NSWindowController *controller = [mainWindow windowController];
    if (controller && [controller isKindOfClass:[CoverWinCtl class]])
        theDrawView = [(CoverWinCtl *)controller cvrView];
    else
        theDrawView = nil;
}
```

Con il proliferare di questo tipo di finestre, ad ogni cambio di finestra principale si scatenava una cascata di notifiche, che eseguono fondamentalmente le stesse operazioni (controllo che finestra si trova ora davanti alle altre, ed aggiorno di conseguenza la variabile d'istanza).

Una situazione del genere mi rode un po', per cui ho centralizzato le operazioni, facendo in modo che un unico attore nell'applicazione ricevesse le notifiche e procedesse a distribuire i compiti di aggiornamento. Contestualmente, ho messo in piedi un meccanismo per la gestione di variabili globali d'applicazione.

Ho scelto la classe `AddDelegate` come artefice del tutto. Finora, la classe era sostanzialmente utilizzata come destinatario delle voci di menu che richiedevano

apertura di finestre e pannelli. L'istanza è unica per l'applicazione ed è costruita automaticamente al caricarsi del file `MainMenu.nib`.

Alla classe ho aggiunto tre variabili d'istanza, destinate a diventare le variabili globali dell'applicazione:

```
    NSOutlineView        * theOutlineView ;
    CoverView            * theDrawView ;
    CCE_BasicForm        * theSelectedElem ;
```

Queste tre variabili contengono: un puntatore alla `NSOutlineView` quando è di fronte a tutti una finestra di catalogo, o nil altrimenti; un puntatore alla `CoverView` e all'elemento selezionato (se disponibile) quando la finestra principale è quella di disegno della copertina. La classe `AppDelegate` è l'unica classe a sottoscrivere le notifiche. Ce ne sono quattro: abbandono di una finestra principale, nuova finestra principale, cambio di selezione all'interno di una finestra di catalogo, cambio di selezione all'interno di una finestra di copertina:

```
static AppDelegate        * applicationGlobals = nil ;

- (id)
init
{
    if ( applicationGlobals )
        return ( applicationGlobals );
    // se arrivo qui devo costruire l'istanza condivisa
    self = [ super init ] ;
    // l' istanza condivisa e' proprio questa
    applicationGlobals = self ;
    // sottoscrivo alcune notifiche
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowChanged:)
        name:NSWindowDidBecomeMainNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(mainWindowResigned:)
        name:NSWindowDidResignMainNotification object:nil];
    // osservatore sul cambiamento della selezione corrente
    [[ NSNotificationCenter defaultCenter] addObserver: self
        selector: @selector( outViewSelChange: )
        name: NSOutlineViewSelectionDidChangeNotification object: nil ] ;
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(cvrViewSelChange:)
        name:@"CoverViewChangedSelection" object:nil];
    return (self );    // restituisco l'istanza
}
```

In seguito al cambiamento della finestra principale, i metodi invocati sono:

```
- (void)
mainWindowChanged: (NSNotification *)notification
{
            NSWindowController   *controller   =   [[notification   object]
windowController];
    if (controller && [controller isKindOfClass:[CoverWinCtl class]])
    {
        [ self updateListWinInfo: nil ] ;
        [ self updateCoverViewInfo: [(CoverWinCtl *)controller cvrView] ] ;
    }
```

```
    else if (controller && [controller isKindOfClass:[ListWinCtl class]])
    {
        [ self updateListWinInfo: [(ListWinCtl *)controller getMainView ] ]
;
        [ self updateCoverViewInfo: nil ] ;
    }
    else
    {
        [ self updateListWinInfo: nil ] ;
        [ self updateCoverViewInfo: nil ] ;
    };
}

- (void)
mainWindowResigned: (NSNotification *)notification
{
    [ self updateListWinInfo: nil ] ;
    [ self updateCoverViewInfo: nil ];
}
```

A seconda del tipo di finestra che si presenta, richiedono l'aggiornamento dei pannelli collegati.

Ugualmente il cambio di selezione in una delle finestre provoca l'esecuzione di:

```
- (void )
outViewSelChange: ( NSNotification *) notification
{
    NSOutlineView * outV = (NSOutlineView*) [ notification object ];
    if ( [ outV tag] == TAG_CDCATDOC_FULLLIST )
        [ self updateListWinInfo: [ notification object ] ];
}

- (void)
cvrViewSelChange: (NSNotification *)notification
{
    id notifObj = [notification object] ;
    if ( [notifObj isKindOfClass:[CoverView class]])
            [ self updateCoverViewInfo: (CoverView*) notifObj ];
    else    [ self updateCoverViewInfo: nil ];
}
```

Il primo metodo controlla che sia modificata la selezione nella outlineView della finestra del catalogo (verifica che il tag corrisponda per essere sicuro), mentre il secondo verifica che la notifica provenga da una `CoverView`.

Le operazioni di aggiornamento vero e proprio sono svolte dai due metodi seguenti:

```
- (void)
updateListWinInfo: (NSOutlineView *) outView
{
     InfoWinCtrl *tempWC1 = (InfoWinCtrl*) [ InfoWinCtrl sharedInfoWinCtrl
];
    theOutlineView = outView ;
    // impongo l'aggiornamento alle varie finestre accessorie
    [ tempWC1 updateInfo: theOutlineView ] ;
}
```

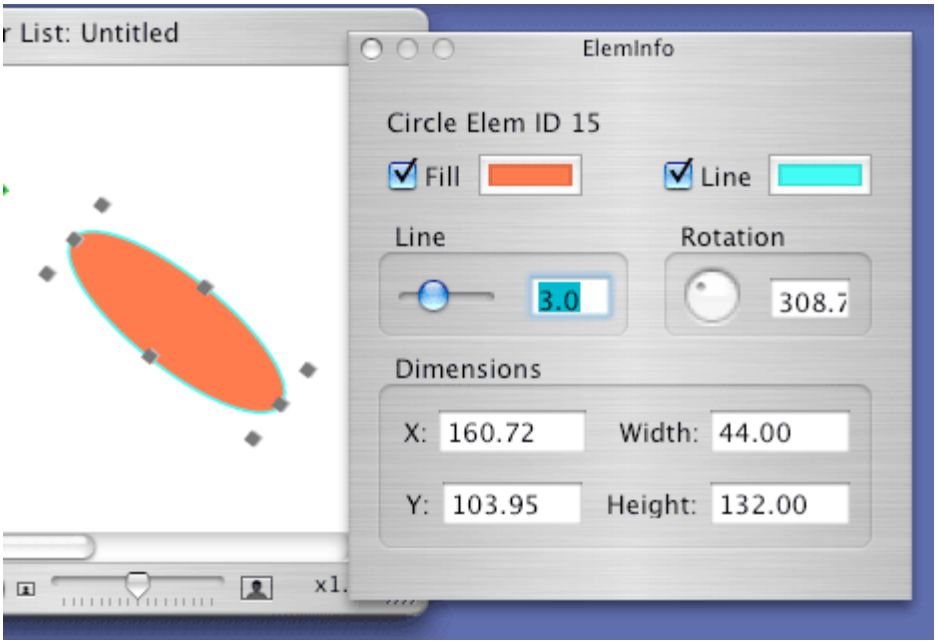
Al momento, l'unica finestra che dipende dalla finestra di catalogo è quella delle

informazioni relative ad un file. Molto più articolato invece il metodo relativo alla finestra della copertina; qui ci sono ben quattro pannelli che ne dipendono (il pannello di informazioni su di un elemento grafico; il pannello per eseguire la rotazione; il pannello per il controllo della griglia; il pannello per i comandi di allineamento). Inoltre, i primi due pannelli utilizzano anche l'elemento correntemente selezionato (se ne è selezionato uno solo) per adeguare il loro aspetto. è questo il motivo per cui il metodo si preoccupa di verificare la presenza di elementi selezionati e di impostare valori adeguati alla variabile `theSelectedElem`:

```
- (void)
updateCoverViewInfo: (CoverView *) coverView
{
    ElemInfoWinCtl *tempWC1 = (ElemInfoWinCtl*)
            [ ElemInfoWinCtl sharedElemInfoWinCtl ];
    RotateWinCtl    *tempWC2 = (RotateWinCtl*)
        [ RotateWinCtl sharedRotateWinCtl ];
    GridDlgWinCtl    *tempWC3 = (GridDlgWinCtl*)
        [ GridDlgWinCtl sharedGridDialogWinCtl ];
    AlignWinCtl        *tempWC4 = (AlignWinCtl*)
        [ AlignWinCtl sharedAlignPaletteWinCtrl ];

        // aggiorno la coverview in primo piano
    theDrawView = coverView ;
    if ( theDrawView == nil )
    {
        // se non c'e' la coverview, non ci sono elementi selezionati
        theSelectedElem = nil ;
    }
    else
    {
        NSMutableArray * selArr ;
        int                numElem ;
        // guardo quanti elementi sono correntemente selezionati
        selArr = [ [ theDrawView theElements] getSelectedGraphics ];
        numElem = [ selArr count ];
        // se non ce ne sono...
        if ( numElem == 0 )
        {
            // dico che se ne fa nulla
            theSelectedElem = nil ;
        }
        // ... o ce ne sono troppi
        else if ( numElem > 1 )
        {
            // dico che se ne fa nulla
            theSelectedElem = (CCE_BasicForm*)(-1) ;
        }
        else
        {
            // c'e' un solo elemento, imposto il puntatore
            theSelectedElem = [ selArr objectAtIndex: 0 ] ;
        }
    }
    // impongo l'aggiornamento alle varie finestre accessorie
    [ tempWC1 setSelectedElem: theSelectedElem ] ;
    [ tempWC2 setSelectedElem: theSelectedElem ] ;
    [ tempWC3 updateWindow: theDrawView ] ;
    [ tempWC4 updateWindow: theDrawView ] ;
}
```

INFORMAZIONI RIVISTE

Piuttosto che illustrare noiosamente tutte le modifiche effettuate sulle classi `AlignPaletteWinCtl` (che ora si chiama `AlignWinCtl`), `ElemInfoWinCtl`, `GridDialogWinCtl` (che ora si chiama GridDlgWinCtl) e `InfoWinCtl`, discuto solo quelli relativi a `ElemInfoWinCtl`, pannello che per altro ha subito anche qualche ritocco a livello grafico.



Al caricamento del pannello, non ci sono più sottoscrizioni a notifiche, ma solo la predisposizione dell'aspetto della finestra:

```
- (void)
windowDidLoad
{
    [super windowDidLoad];
    // predisposizione dei vari elementi della finestra
    [fillCheckbox setState:NSOffState];
    [fillColorWell setColor:[NSColor whiteColor]];
    [lineCheckbox setState:NSOnState];
    [lineColorWell setColor:[NSColor blackColor]];
    [lineWidthSlider setFloatValue:0.0];
    [lineWidthTextField setFloatValue:0.0];
    [rotationSlider setFloatValue: 0.0 ];
    [rotationText setFloatValue: 0.0 ];
    [xTextField setStringValue:@""];
    [yTextField setStringValue:@""];
    [widthTextField setStringValue:@""];
    [heightTextField setStringValue:@""];
    [ self setSelectedElem: [ AppDelegate getTheSelectedElem ] ];
}
```

La prima predisposizione avviene utilizzando l'elemento correntemente selezionato; questo è recuperato attraverso un metodo di classe di `AppDelegate`:

```
+ (CCE_BasicForm *)
```

```
getTheSelectedElem
{
    return ( [ applicationGlobals theSelectedElem ] );
}
```

che sfrutta l'istanza condivisa per restituire il valore della variabile d'istanza corrispondente (questo sì aggiornato dal meccanismo delle notifiche).

L'aggiornamento della finestra è effettuato dal metodo `setSelectedElem:`, che è anche il metodo chiamato da `updateCoverViewInfo:` quando pensa che sia necessario.

```
- (void)
setSelectedElem: (CCE_BasicForm *) selElem
{
    theSelectedElem = selElem;
    // se non ci sono eselementi selezionati
    if ( theSelectedElem == nil )
    {
        // dico che se ne fa nulla
        [ elemId setStringValue: @"No graphic element selected"];
        [ self setControlsEnabled: NO ] ;
        return ;
    }
    // ... o ce ne sono troppi
    else if ( theSelectedElem == (CCE_BasicForm*)(-1) )
    {
        // dico che se ne fa nulla
                [ elemId setStringValue: @"Too many graphic elements
selected"];
        [ self setControlsEnabled: NO ] ;
        return ;
    }
    // se arrivo qui, c'e' un solo elemento selezionato
    // attivo tutti gli elementi della finestra
    [ self setControlsEnabled: YES ] ;
    // aggiorno il contenuto dei vari campi
    [ self updateWindowInfo ];
}
```

In base al valore dell'elemento selezionato, aggiorna il contenuto del pannello, fornendo una adeguata stringa e predispondendo i controlli abilitati o disabilitati. L'ulteriore metodo `updateWindowInfo` si occupa di aggiustare i valori dei controlli stessi:

```
- (void)
updateWindowInfo
{
    // predispongo gli elementi che corrispondono agli attributi comuni
    [fillCheckbox setState: [ theSelectedElem cceIsFilled] ];
    [fillColorWell setColor: [ theSelectedElem cceFillColor] ];
    [lineCheckbox setState: [ theSelectedElem cceIsStroked] ];
    [lineColorWell setColor:[ theSelectedElem cceLineColor] ];
    [lineWidthSlider setFloatValue: [ theSelectedElem cceLineWidth] ];
    [lineWidthTextField setFloatValue: [ theSelectedElem cceLineWidth] ];
    [rotationSlider setFloatValue: [ theSelectedElem rotAngle]];
    [rotationText setFloatValue: [ theSelectedElem rotAngle]];
    [xTextField setStringValue: [ NSString stringWithFormat: @"%f",
            [theSelectedElem drawPoint].x ] ];
    [yTextField setStringValue: [ NSString stringWithFormat: @"%f",
```

```
            [theSelectedElem drawPoint].y ] ];
    [widthTextField setStringValue: [ NSString stringWithFormat: @"%f",
            [theSelectedElem localSize].width ] ];
    [heightTextField setStringValue: [ NSString stringWithFormat: @"%f",
            [theSelectedElem localSize].height ] ];
    // preparo una scritta a seconda dell'elemento
    [ elemId setStringValue: [ theSelectedElem describeMe ] ];
}
```

È da notare che con la nuova struttura degli elementi, cade la distinzione tra `CCE_BasicForm` e `CCE_Line` (che richiedeva in precedenza trattamento distinto), per cui il metodo è molto semplice e lineare.

Ci sono due nuovi elementi, uno slider ed un campo testo per rappresentare l'angolo di rotazione corrente dell'elemento. Ancora una volta, modificando il campo di testo od operando sullo slider circolare, si modifica lo stato dell'elemento, fornendo così un ulteriore meccanismo di rotazione.

Anche il metodo `changedElem`: si giova della uniformità di trattamento dei vari elementi:

```
- (IBAction)
changedElem:(id)sender
{
    float p1, p2, p3, p4 ;
    // a seconda dell'attributo modificato
    switch ( [ sender tag ] ) {
    case 10 :     // stato di riempimento
        [ theSelectedElem setCceIsFilled: [ fillCheckbox state ]];
        break ;
    case 11 :     // colore di riempimento
        [ theSelectedElem setCceFillColor: [ fillColorWell color ]];
        break ;
    case 12 :     // stato di disegno linea/contorno
        [ theSelectedElem setCceIsStroked: [ lineCheckbox state ]];
        break ;
    case 13 :     // colore della linea
        [ theSelectedElem setCceLineColor: [ lineColorWell color ]];
        break ;
    case 14 :     // dimensione della linea
    case 15 :
        [ theSelectedElem setCceLineWidth: [ lineWidthSlider floatValue ]];
        break ;
    case 16 :     // coordinate limite dell'elemento
    case 17 :
        p1 = [xTextField floatValue ];
        p2 = [yTextField floatValue ];
        [ theSelectedElem setDrawPoint: NSMakePoint( p1, p2) ];
        [ theSelectedElem calcLocalTranform ];
        break ;
    case 18 :
    case 19 :
        // recupero i quattro valori
        p3 = [widthTextField floatValue ];
        p4 = [heightTextField floatValue ];
        [theSelectedElem resizeMeWith: NSMakeSize( p3, p4 ) ];
        break ;
    case 20 :     // angolo di rotazione
    case 21 :
        [ theSelectedElem rotateMeAtAngle: [ rotationSlider floatValue ] ];
```

```
        break ;
    default:
        NSLog( @"changed %@", sender );
    }
    // forzo un ridisegno
    [ [theSelectedElem ownerView] setNeedsDisplay: YES ];
    // ed anche un aggiornamento delle informazioni
    [ self updateWindowInfo ];
}
```

Oltre ai due nuovi casi relativi alla rotazione, sono stati divisi i casi relativi allo spostamento del punto origine ed alla variazione delle dimensioni dell'elemento.

Capitolo 053 - Modificare il testo

In questo capitolo miglioro la classe `CCE_Text` introducendo la possibilità di modificare il testo, ed anche i singoli attributi dei caratteri.

Sorgente: Parto dall'esepio *Sketch* come idea, ma poi ballo da solo.

Prima stesura: 20 agosto 2004

DOPPIA IDEA

Fino a questo momento, l'unica possibilità di inserire degli elementi di testo all'interno di una `CoverView` era a disposizione del programmatore; il testo si può inserire solo da programma. Inoltre, non è possibile modificare il testo, o i suoi attributi, se non a livello globale (ad esempio, sostituendo l'intero contenuto di uno specifico `CCE_Text` con il catalogo di un volume). Per consentire all'utente l'inserimento di elementi `CCE_Text`, e la sua successiva modifica, bisogna permettere in primo luogo la modifica di un testo.

Allo scopo, osservo l'idea messa in pratica nell'esempio *Sketch*, fornito con l'ambiente *XCode*. Il testo è normalmente rappresentato in una vista disegnandolo in un metodo `draw`, come del resto già avviene anche per `CCE_Text`. Per permettere la modifica del testo, Sketch produce al volo un oggetto `NSTextView`, e lascia che sia questo oggetto a sbrigare tutte le pratiche. Al termine delle operazioni, lo elimina e passa oltre. Una idea che mi è parsa brillante (perché lavorare quando c'è qualcuno che può farlo per te?), che ho subito messo in pratica con due fondamentali modifiche.

La prima modifica è molto semplice; piuttosto che costruire al volo una `NSTextView`, ne uso una sola, sempre la stessa, adattandola di volta in volta ai miei scopi. In realtà è quanto fa anche Sketch, che usa una unica `NSTextView` globale; tuttavia, la mia pensata è di evitarne del tutto la costruzione da proramma, definendola direttamente all'interno del file nib.

In altre parole, uso *Interface Builder* per aggiungere, all'interno del file `CoversWin.nib`, una `NSTextView`, normalmente nascosta. Poi, quando necessario, la vista è resa visibile per consentire le operazioni, mentre, al termine delle stesse, è nuovamente resa invisibile.

La seconda modifica è un po' più interessante: invece di utilizzare una `NSTextView` semplice, uso una `NSScrollView`, in modo che sia possibile modificare tutto il testo, e non solamente quello rappresentato all'interno del rettangolino che circoscrive `CCE_Text`. Quindi, all'interno del file `CoversWin.nib`, costruisco in realtà un oggetto `NSScrollView`, all'interno del quale si trova, quale suo documento da visualizzare, una `NSTextView`.

La faccenda è ulteriormente complicata dal fatto che il testo è potenzialmente ruotato secondo un angolo arbitrario; piuttosto che modificare tale testo in loco, preferisco fare in modo che quando l'utente vuole modificare del testo, questo gli si presenti sempre bello dritto, in modo che si possa leggere comodamente, piuttosto che ruotato secondo

l'angolo specificato.

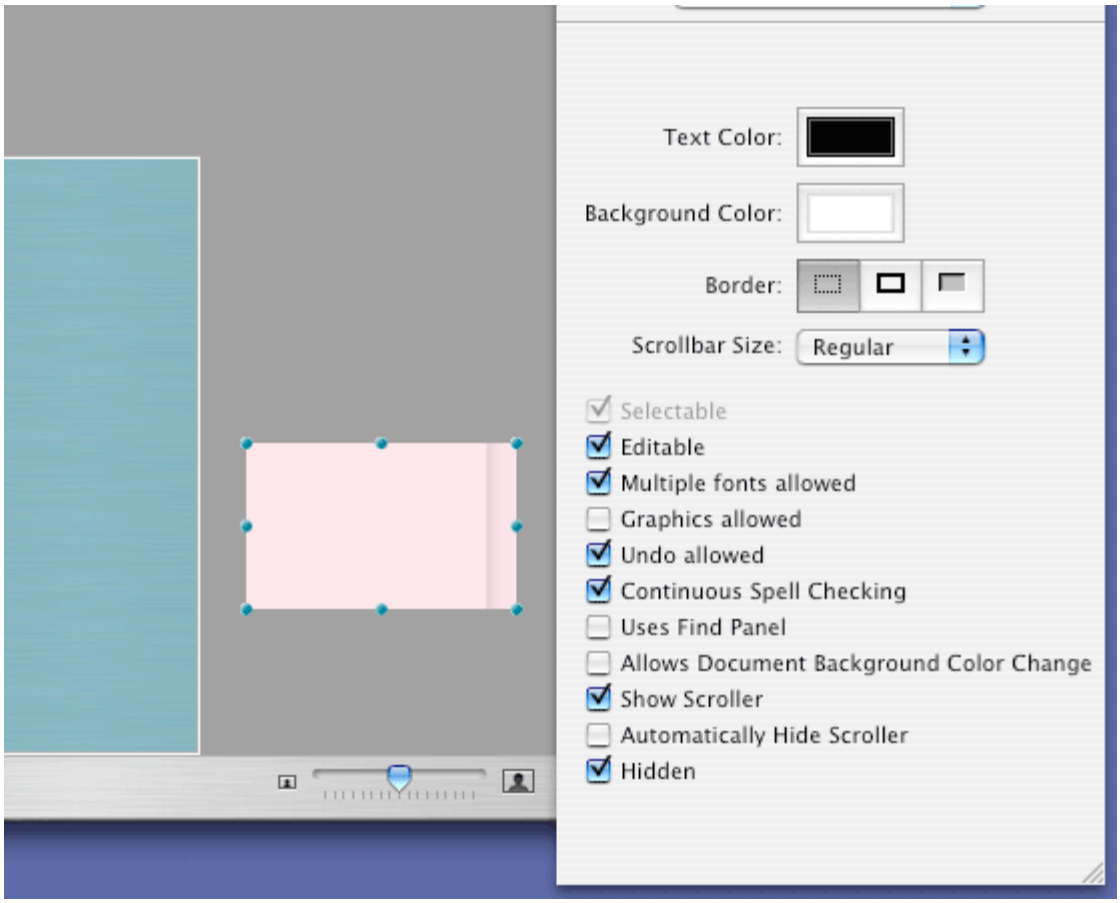
SEQUENZA DELLE OPERAZIONI

Per poter modificare del testo, la sequenza è dunque la seguente: l'utente individua l'elemento `CCE_Text` da modificare facendo doppio cli su di esso. L'oggetto allora si pone in modalità di *editing*: piuttosto che disegnare il testo secondo il metodo normale, lascia che a occuparsene sia una `NSTextView` opportunatamente predisposta. Al termine delle operazioni, l'oggetto `CCE_Text` torna alla normale operatività cessando di impiegare la `NSTextView` e disegnando il testo con il metodo normale. Ci sono quindi le seguente modifiche sul codice: l'individuazione di un doppio clic, l'apertura di una sessione di editing, la chiusura di una sessione di editing, le modifiche al metodo di disegno per gestire il tutto.

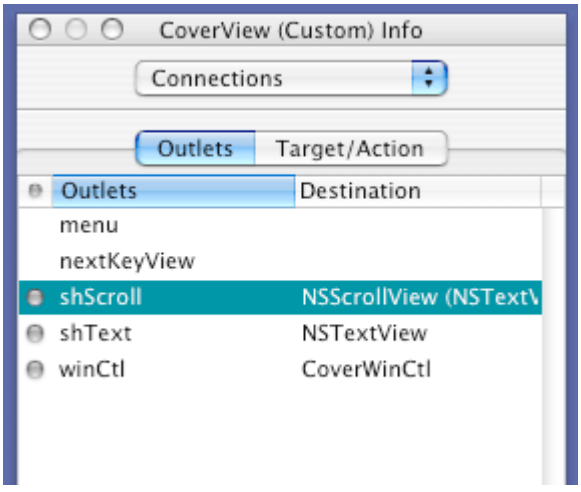
Individuare un doppio clic è piuttosto semplice. All'interno dell'oggetto `NSEvent`, passato ad esempio come argomento al metodo `mouseDown:` dell'oggetto `CoverView`, è in effetti presenti l'informazione se si sono verificati clic ripetuti; attraverso il metodo `clickCount` si determina quante volte l'utente ha fatto clic. Se il numero è maggiore o uguale a due, abbiamo riconosciuto un doppio clic.

Il posto più adatto per effettuare il controllo sul doppio clic mi pare sia all'interno del metodo `handleMouseClick:`, non appena è stato riconosciuto un clic sopra un elemento. In questo modo evito di preoccuparmi se il doppio clic è avvenuto in terra di nessuno. Di conseguenza il metodo si presenta in questo modo:

```
- (void)
handleMouseClick: (NSEvent *)theEvent
{
    NSPoint            whereClick;
    CCE_BasicForm * curElem = nil;
      BOOL                  shiftKeyPress  =  (([theEvent  modifierFlags]  &
NSShiftKeyMask) ? YES : NO);
    short            clicInHandle, numofclic ;
    // vado a vedere dove e' avvenuto il click
          whereClick   =   [self   convertPoint:[theEvent   locationInWindow]
fromView:nil];
    numofclic = [ theEvent clickCount ] ;
    // niente niente l'utente vuole selezionare un elemento
    curElem = [self getClickedElem:whereClick];
    // se c'e' un elemento sotto il mouse
    if ( curElem )
    {
        // abbiamo qualcosa sotto il mouse
        // se e' un doppio clic, editing se il caso
        if ( numofclic >= 2 )
            [ curElem startEditing ] ;
        // vedo se e' gia' selezionato
        else if ( [ curElem cceIsSelected ] )
        {
                if ( shiftKeyPress )
                {
                ...
```



L'effetto di un doppio clic è quindi quello di inviare un messaggio `startEditing` all'oggetto sotto il clic. Normalmente, questo messaggio cade nel vuoto, perché un elemento generico non è in grado di rispondere in maniera sensata (al momento...) ad un doppio clic. Ciò non è vero per un elemento di classe `CCE_Text`, che invece proprio di questo doppio clic fa tesoro.



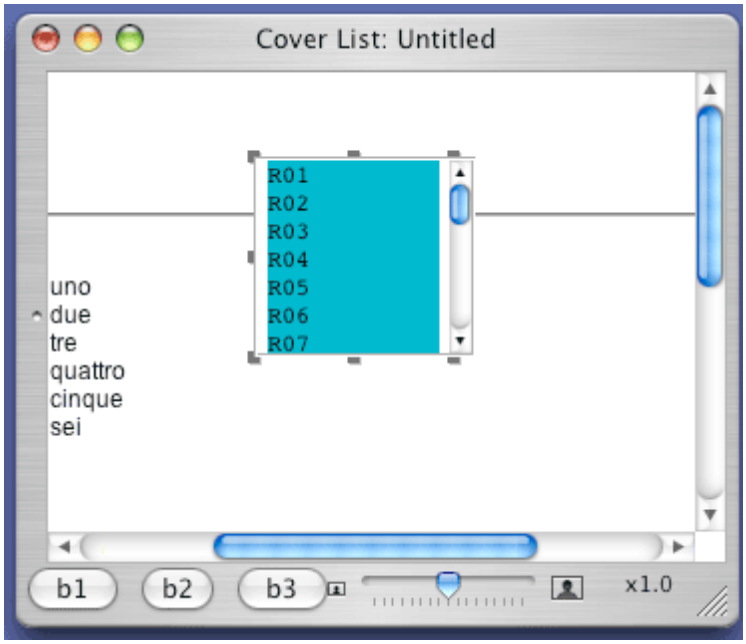
Ma prima di passare ad esaminare il metodo nel merito, occorre fare un passo indietro e parlare del file `CoversWin.nib`. Ho già detto che, per gestire la modifica del testo, utilizzo una `NSScrollView` con `NSTextView` incorporata, costruite direttamente all'interno della finestra. Ecco quindi che ho aggiunto questi oggetti attraverso *Interface Builder*. Questi oggetti sono dichiarati fin da subito `Hidden` attraverso l'apposito

pulsante di spunta. Inoltre, la `NSScrollView` dovrebbe essere una sottovista della `CoverView` (dopotutto, è uno degli elementi che fa parte della copertina), ma non riesco ad ottenere ciò direttamente in *Interface Builder*. Infine, per permettere rapido accesso a questi due oggetti, aggiungo due outlet (`shScroll` e `shText`) alla `CoverView` e li collego direttamente alla `NSScrollView` ed alla `NSTextView`.

Di ritorno in *XCode*, completo l'inizializzazione: all'interno del metodo `windowDidLoad` della classe `CoverWinCtl` aggiungo qualche riga.

```
    ...
    [ [ cvrView shScroll ] removeFromSuperview];
    [ cvrView addSubview: [ cvrView shScroll ]];
    [ [ cvrView shScroll ] setHasHorizontalScroller: NO ];
    [ [ cvrView shScroll ] setHasVerticalScroller: YES ];
        [ [ [ cvrView shScroll ] verticalScroller ] setControlSize:
NSSmallControlSize ] ;
    [ [ cvrView shScroll ] setAutohidesScrollers: NO ];
    [ [ cvrView shScroll ] setBorderType: NSGrooveBorder ] ;
    ...
```

Le prime due istruzioni sono quelle fondamentali: tolgono la `NSScrollView` dalla gerarchia stabilita da *Interface Builder*, e la fanno diventare una sottovista della `CoverView`. Le altre istruzioni non fanno altro che ribadire e precisare le caratteristiche che la `NSScrollView` deve avere.



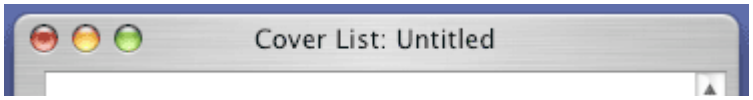
Ho scelto dall'inizio di rappresentare il testo all'interno di un rettangolo di larghezza ed altezza determinata; è possibile che una parte del testo non riesca ad essere visualizzato. Tuttavia, il testo è sempre incolonnato all'interno della larghezza specificata, per cui l'eventuale testo non rappresentato è solo quello delle ultime righe. Questo spiega perché non è presente la barra di scorrimento orizzontale (il testo sarà giustificato sulla larghezza del rettangolo), ma, nella sola fase di modifica, è presente la barra verticale. Infine, dico che la barra è comunque visibile, anche se di dimensioni ridotte.

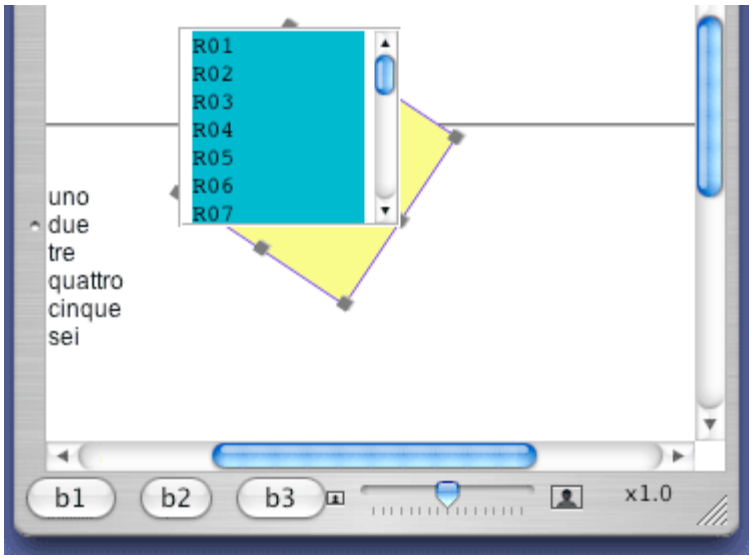
COMINCIARE LE MODIFICHE

Posso adesso commentare in modo adeguato il metodo `startEditing` proprio della classe `CCE_Text`:

```
- (void)
startEditing
{
    // recupero puntatori alla scrollView ed alla textView nascoste
    NSScrollView    * theScroll = [ ownerView shScroll ];
    NSTextView        * theEditText = [ ownerView shText ];
    // calcolo il rettangolo circoscritto; usero' l'origne come punto
    // base per il disegno della scrollview
     NSRect locrect = [ [ localTF transformBezierPath: theDrawPath ] bounds
] ;
    // la visualizzazione funziona meglio se arrotondo a pixel interi
    locrect.origin.x = round( locrect.origin.x );
    locrect.origin.y = round( locrect.origin.y );
    // le dimensioni della scrollview sono quelle del testo, corrette
    // perche' siano positive per non ingenerre confusione
    locrect.size.width = ( localSize.width < 0 ) ?
                    - localSize.width : localSize.width ;
    locrect.size.height = ( localSize.height < 0 ) ?
                    - localSize.height : localSize.height ;
    // dico che questo elemento e' in fase di editing
    editingText = YES ;
    // ed aggiusto la variabile della view per la sua gestione
    [ownerView setEditingInProgress:self ];
    // dimensioni della scrollview, faccio spazio per lo scroller
    [ theScroll setFrame: NSMakeRect(locrect.origin.x , locrect.origin.y,
                locrect.size.width + 11 , locrect.size.height ) ];
        // questa scrollview/textview e' gestita come unlayout manager
aggiuntivo
    [ textStorage addLayoutManager: [ theEditText layoutManager ] ] ;
    // mi riporto in testa e seleziono tutto il testo
    [ theEditText scrollPoint: NSMakePoint( 0, 0 ) ];
    [ theEditText setSelectedRange: NSMakeRange(0, [ textStorage length ] )
];
    // dico che la text e' il primo risponditore degli eventi
    // cosi' che tastiera e menu relativi al testo cadano qui
    [[ownerView window] makeFirstResponder:theEditText];
    // visualizzo la scrollviea
    [ theScroll setHidden: NO ] ;
    // forzo un ridisegno
    [ ownerView setNeedsDisplay: YES ];
}
```

In queste sedici istruzioni ci sono molti concetti che devono essere spiegati. Per prima cosa, il criterio di visualizzane della `NSTextView` (e della `NSScollView` che la contiene); dico che, indipendentemente dall'angolo di rotazione, la modifica del testo avviene nella modalità normale. Questo significa che devo calcolare una buona posizione e dimensione per queste due viste. Piglio quindi il rettangolo che circoscrive l'oggetto `CCE_Text` e stabilisco che il punto di origine di questo rettangolo è il punto di origine della `NSTextView`. In questo modo, se il `CCE_Text` ha un angolo nullo, la `NSTextView` ci va proprio sopra; altrimenti, in qualche modo si trova nei pressi.





Il flag `editingText` posto a Vero è utilizzato per individuare in questo elemento quello in corso di modifica, e sarà quindi il flag che determina il metodo di disegno dell'elemento stesso. Vedremo il suo uso più tardi. Analoga funzione la variabile `editingInProgress`, propria della `CoverView`, utilizzata per discriminare il destinatario di tutte le operazioni in corso.

Ci sono due istruzioni interessanti, di arrotondamento delle coordinate di questo punto origine. Per qualche motivo che non mi è molto chiaro, senza tale operazione il testo può essere rappresentato in malo modo; suppongo che il problema derivi dalle approssimazioni successive sulle coordinate di locrect, che portano il rettangolo a spostarsi di frazioni di pixel causando sovrascritture invedibili.

Una volta stabilito il punto origine della vista, dico che le sue dimensioni sono le stesse utilizzate per la visualizzazione corrente; per evitare di mettere in confusione Cocoa con lunghezze negative, predispongo il tutto perché si abbiamo sempre e solo valori positivi.

A questo punto, posso attribuire le nuove coordinate alla `NSScrollView`, quelle che appena calcolato in termini di punto origine e dimensione; alla dimensione orizzontale aggiungo 11 pixel, che è la dimensione della barra di scorrimento verticale.

Finalmente si arriva all'istruzione che permette a tutto il meccanismo di funzionare, ovvero l'aggiunga di un nuovo `NSLayoutManager` alla variabile `textStorage`, quella che contiene il testo da rappresentare all'interno dell'elemento `CCE_Text`. Riprendendo la discussione del capitolo 49, ricordo l'architettura per il disegno di testo: una classe `NSTextStorage` che conserva il testo, una classe `NSTextContainer` che specifica la zona di visualizzazione, ed una classe `NSLayoutManager` per gestisce l'interazione tra questi due oggetti. In genere una classe `NSLayoutManager` è solidale con l'unica classe `NSTextStorage` per fornire il testo da rappresentare all'interno di più zone specificate da `NSTextContainer` (pensate a `NSLayoutManager` come ad un demone che impagina il testo di `NSTextStorage` in più pagine visualizzate tramite `NSTextContainer`). Nel caso di `CCE_Text`, c'è la classe `NSTextStorage` e c'è una sola classe `NSTextContainer` che rappresenta la zona di spazio all'interno del rettangolo che caratterizza il `CCE_Text`; ad unire queste due classi, una classe `NSLayoutManager`. Questa è la catena che funziona in normali condizioni di disegno. Quando si passa in

modalità di modifica del testo, alla variabile `textStorage` di `CCE_Text` si aggiunge temporaneamente un nuovo oggetto `NSLayoutManager` per una diversa rappresentazione del contenuto del testo; questa rappresentazione utilizza lo `NSTextContainer` specifico della `NSTextView`. Tutto questo paragrafo per spiegare l'unica istruzione:

```
[ textStorage addLayoutManager: [ theEditText layoutManager ] ] ;
```

dove si piglia lo `NSLayoutManager` della `NSTextView` e lo si aggiunge a `textStorage`.

A questo punto, essendoci due gestori per la rappresentazione del testo, potrebbe venire il dubbio che il testo sia rappresentato due volte; ciò non accade, in quanto il primo `NSLayoutManager` (quello standard di `CCE_Text`) è tenuto disattivo nel (prossimamente descritto) metodo di disegno dell'elemento.

Il metodo `startEditing` sta per finire; riporta la barra di scorrimento all'inizio, e seleziona tutto il testo (avrei potuto operare una selezione nulla, ma l'istruzione di selezione va messa anche per attivare il cursore specifico per il testo). Stabilisce poi che la `NSTextView` diventi *First Responder*; in questo modo tutti i comandi di menu relativi al testo, copia, incolla, eccetera, avranno la `NSTextView` come referente, e da questa saranno gestiti in pieno automatismo (ad esempio, è possibile operare con effetti speciali sul testo, rappresentandolo con tutta la potenza messa a disposizione da Cocoa). Finalmente, si visualizza la `NSScrollView` e si forza un ridisegno dalal vista, in modo da mostrare tutto quanto fatto finora.

DISEGNO E FINE DELLE MODIFICHE

Il metodo di disegno del testo ha una piccola, fondamentale modifica, per permettere

l'esecuzione corretta delle operazioni di modifica:

```
- (void)
specificDrawing: (NSRect) inRect
{
    // determino punto e dimensione di dove disegnare
    // per tenere conto di eventuali dimensioni negative
    // disegno del testo
    NSRect locrect = arrangeRect( localSize );
    [ super specificDrawing: inRect ] ;
    // se sto modificando il testo...
    if ( editingText )
    {
        // e' tutto fatto dalla NSTextView
    }
    else
    {
        NSRange glyphRange ;
        // recupero il primo layout
         NSLayoutManager *layoutManager = [ [ textStorage layoutManagers ]
objectAtIndex: 0] ;
        [ theTextCont setContainerSize: locrect.size ];
        // recupero il range del testo
        glyphRange = [layoutManager glyphRangeForTextContainer: theTextCont
];
             [  layoutManager  drawGlyphsForGlyphRange:glyphRange  atPoint:
locrect.origin ];
    }
}
```

Quando la variabile d'istanza `editingText` è Vera, significa che sono in corso le operazioni di modifica; in particolare, è attivo un secondo `NSLayoutManager` preposto alla rappresentazione del testo. È quindi necessario impedire che lo `NSLayoutManager` standard lavori per rappresentare una seconda volta il testo. L'iniziale funzione `arrangeRect(.)` è solo un metodo comodo per aggiustare i valori del punto di disegno e le dimensioni della zona interessata dal disegno. Questa è una correzione di un errore che in precedenza provocava qualche scompenso in presenza di dimensioni negative:

```
NSRect
arrangeRect ( NSSize tmpSize)
{
    NSPoint        wheretodraw = NSMakePoint( 0, 0) ;
    NSSize        newSize = tmpSize ;
    // se ho una dimensione negativa
    if ( tmpSize.width < 0 )
    {
        // sposto il punto di disegno
        wheretodraw.x = tmpSize.width ;
        // in modo da ottnere una dimensione positiva
        newSize.width = -tmpSize.width ;
    }
    if ( tmpSize.height < 0 )
    {
        wheretodraw.y = tmpSize.height ;
        newSize.height = -tmpSize.height ;
    }
       return  ( NSMakeRect(  wheretodraw.x,  wheretodraw.y,  newSize.width,
newSize.height) ) ;
}
```

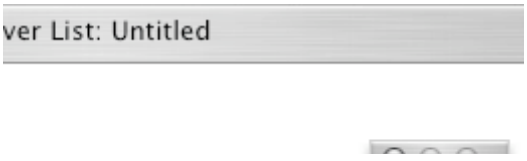
Finché l'utente continua ad operare sulla `NSTextView`, tutto funziona tranquillamente. Ad un certo punto, tuttavia, cambierà qualcosa (ad esempio, un clic altrove). Occorre allora uscire dalla modalità di modifica e tornare a lavorare in maniera normale. Per prima cosa, si deve riconoscere la situazione: all'interno del metodo `mouseDown:` della `CoverView` la prima istruzione è sempre questa:

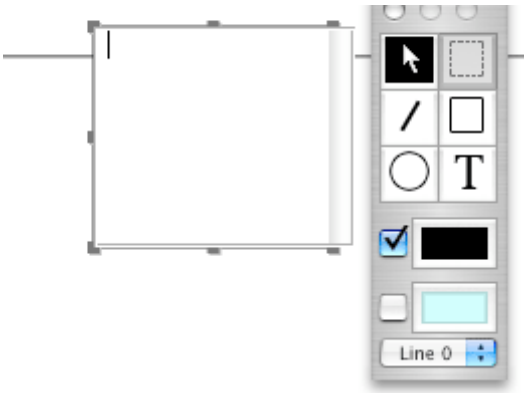
```
- (void)
mouseDown:(NSEvent *)theEvent
{
            Class   selCls   =   [[CoverPalWinCtl   sharedCoverPalWinCtrl]
getCurrClassTool];

    if ( editingInProgress )
    {
        [ editingInProgress endEditing ];
    }
    ...
```

È interessante notare che questo metodo è invocato solamente se il clic è avvenuto fuori della `NSTextView`; infatti questa, in qualità di *First Responder*, accetta direttamente ogni clic (ed ogni input da tastiera) che la riguarda; solo nel caso in cui il clic sia al di fuori della sua zona di interesse, passa l'evento alle view superiori, tra cui appunto la `CoverView` che lo gestisce. Insomma, sono certo che quando il metodo `mousedown:` è chiamto, è giunto il momento di chiudere le operazioni di modifica:

```
- (void)
endEditing
{
    // recupero puntatori alla scrollView ed alla textView
    NSScrollView * theScroll = [ ownerView shScroll ];
    NSTextView * theEditText = [ ownerView shText ];
    // se non e' per me, lascio perdere
    if ([ownerView editingInProgress] != self)
        return ;
    // se arrivo qui, fine editing per me
    editingText = NO ;
    [ ownerView setEditingInProgress:nil ];
    // tolgo il layout manager, che non serve piu'
    [ textStorage removeLayoutManager: [ theEditText layoutManager ] ] ;
    // nascondo la scrollview
    [ theScroll setHidden: YES ] ;
    // lascio che adesso tastiera e menu insistano sulla coverview
    [ [ownerView window] makeFirstResponder:ownerView ];
}
```

A parte l'ovvio controllo sul fatto che le operazioni van chiuse per il solo oggetto `CCE_Text` interessato, sono riportate allo stato originario le variabili d'istanza dell'elemento e della `CoverView`; la `NSScrollView` viene nuovamente resa invisibile, e il *First Responder* torna ad essere la `CoverView`. Notare la rimozione dello `NSLayoutManager` della `NSTextView` da `textStorage`, che a questo punto ritorna ad avere un solo gestore di visualizzazione del testo.





Come tocco finale a tutta la faccenda, ritorno in *Interface Builder* e modifico il file `CoverPalette.nib`, abilitando il controllo che consente la costruzione di elementi di classe `CCE_Text`. Tuttavia, così com'è, la cosa non funziona correttamente; il problema è la mancata inizializzazione delle variabili d'istanza specifiche della `CCE_Text`: guardando all'interno del metodo `mouseDown:`, nella parte dove è costruito un nuovo elemento, questo è costruito con la sequenza `alloc` e `init`, utilizzando quindi un costruttore *povero* per `CCE_BasicForm` al posto del più ricco costruttore standard. Perfar funzionare le cose, occorre sovrascrivere il metodo `init` all'interno di `CCE_Text` affinché chiami in cascata il costruttore designato.

```
- (id )
init
{
    self = [ self initWithId: 0 inView: nil withAttributedString: nil
                inRect: NSMakeRect(0,0,1,1) withAngle: 0 ] ;
    return self ;
}
```

Non sembrano esserci progressi; in realtà, invocare il metodo `initWithId:...` consente la corretta inizializzazione delle variabili d'istanza, tra cui la fondamentale `textStorage`, destinata a contenere tutto il testo che si desidera.

Se poi si aggiunge, sempre all'interno del metodo `mouseDown:` una semplice istruzione di inizio modifica del testo:

```
        ...
        if ([curElem handleCreation: theEvent ])
        {
            // se la creazione e' andata a buon fine,lo aggiungo alla lista
            [ curElem setObjID: [self getNextObjId]];
            [ theElements addElem:curElem atIndex: 0 ];
            [ curElem startEditing ] ;
        }
        ...
```

allora, subito dopo aver tracciato il rettangolo che delimita il testo, si entra subito in modalità di modifica, con la possibilità di inserire subito del testo all'interno del nuovo elemento `CCE_Text`.

Capitolo 054 - Le correzioni

Ma perché nessuno mi dice che il meccanismo dei gruppi non funziona più? In questo capitolo eseguo le necessarie correzioni sulla classe `CCE_ElemGroup`, per poi passare ad alcuni argomenti sparsi: gestione degli stili del testo, riempimento automatico dei campi testo, menu contestuali.

Sorgenti: il titolo del capitolo è anche quello di un libro di *Jonathan Franzen*, edito da Einaudi.

Prima stesura: 25 agosto 2004.

CORREZIONI SUI GRUPPI

Con le modifiche introdotte nei capitoli 51 e 52, gli elementi della classe `CCE_ElemGroup` hanno cessato di funzionare correttamente. In effetti, la riorganizzazione ha invalidati molti comportamenti: più che correggerli puntualmente, ho preferito uniformare il loro comportamento alla luce dell'introduzione dei Sistemi di Riferimento *SdR* dei singoli elementi. L'idea è che alla costruzione di un gruppo, al gruppo sia assegnato, come a tutti gli altri elementi, un SdR proprio, con una origine ed appropriate dimensioni orizzontali e verticali. Il disegno del gruppo quindi parte dopo una trasformazione di coordinate; a questo punto, le coordinate dei singoli elementi appartenenti al gruppo vanno modificate per tenere conto di questo fatto. In altri termini, alla costruzione di un gruppo di elementi, il SdR degli elementi va modificato per riferirlo non più alle coordinate della finestre, ma alle coordinate del gruppo di elementi. Ovviamente, alle distruzione di un gruppo, occorre modificare nuovamente i SdR degli elementi per tenere conto della trasformazione intermedia non più presente.

Il metodo preposto alla costruzione di un gruppo è stato sostanzialmente modificato, anche se a prima vista non appare:

```
- (void)
makeGroupFromSelected
{
    int         i , numElem, firstIdx = -1, lastIdx ;
    CCE_BasicForm * elem ;
    CCE_ElemGroup * newGrp ;
    // ciclo su tutti gli elementi del gruppo
    ...
    // quando arrivo qui, ho costruito un nuovo gruppo
    // se non c'e' nulla, faccio nulla
    ...
    // se c'e' un solo elemento, lo rimetto dov'era
    ...
    // finalmente, se arrivo qui, ho un gruppo vero e proprio
    // aggiusto tutti i valori del gruppo
    [ newGrp adaptModification ] ;
    // tutti gli elementi interni non sono selezionati
    [ newGrp cascadeSelection: NO ];
    // il gruppo invece rimane selezionato
    [ newGrp setCceIsSelected: YES ];
    // inserisco il gruppo nel posto opportuno
    [ elemArray insertObject: newGrp atIndex: lastIdx];
}
```

In effetti, ho sostituito tutta la parte successiva alla costruzione del vettore con gli elementi selezionati con la singola istruzione che invoca il metodo `adaptModification`. È qui che viene svolto tutto il lavoro importante.

```
- (void)
adaptModification
{
    int        i , numElem ;
    CCE_BasicForm * elem ;
    float        xmin, xmax, ymin, ymax ;

    numElem = [ elemArray count ];
    elem = [ elemArray objectAtIndex: 0 ];
    ymin = [[ elem getElemTop] floatValue];
    xmin = [[ elem getElemLeft] floatValue];
    xmax = [[ elem getElemRight] floatValue];
    ymax = [[ elem getElemBottom] floatValue];
    // ciclo su tutti gli elementi del gruppo
    for ( i = 1 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
        // cerco i limiti del rettangolo circoscritto
        if ( ymin > [[ elem getElemTop] floatValue])
            ymin = [[ elem getElemTop] floatValue];
        if ( xmin > [[ elem getElemLeft] floatValue] )
            xmin = [[ elem getElemLeft] floatValue];
        if ( xmax < [[ elem getElemRight] floatValue] )
            xmax = [[ elem getElemRight] floatValue];
        if ( ymax < [[ elem getElemBottom] floatValue] )
            ymax = [[ elem getElemBottom] floatValue];
    }
    [ self setTheDrawPath: [NSBezierPath bezierPathWithRect:
            NSMakeRect( 0, 0, xmax - xmin, ymax - ymin) ] ];
    [ self setDrawPoint: NSMakePoint( xmin, ymin ) ];
    [ self setRotAngle: 0 ];
    [ self setLocalSize: NSMakeSize( xmax - xmin, ymax - ymin) ] ;
    [ self calcLocalTranform ];
    [ self buildHdlList ] ;
    // sposto le origini di ogni elemento riferendole al gruppo
    for ( i = 0 ; i < numElem ; i ++ )
    {
        elem = [ elemArray objectAtIndex: i ];
          [ elem setDrawPoint: NSMakePoint( [elem drawPoint].x -xmin, [elem
drawPoint].y - ymin ) ];
        [ elem calcLocalTranform ];
    }
}
```

È rimasta inalterata la parte che calcola posizioni e dimensioni spaziali del gruppo cercando i punti estremi di tutti gli elementi del gruppo stesso. Con questi quattro punti estremi costruisco in primo luogo il percorso `theDrawPath` come il rettangolo che racchiude tutti gli elementi; gli stessi punti estremi servono per caratterizzare il punto di origine del SdR del gruppo e le dimensioni del gruppo stesso. In base a questi valori si possono costruire gli handle e calcolare le trasformazioni dei SdR. Successivamente, per ogni elemento appartenente al gruppo, eseguo un cambiamento nel SdR. Il cambiamento consiste nella sola traslazione, in quanto la costruzione del gruppo avviene necessariamente ad un angolo nullo.

Per disegnare gli handle, si deve utilizzare un metodo del tutto uguale a quello proprio

degli elementi `CCE_Rect`. Ecco quindi che ho spostato la realizzazione del metodo `buildHdList` della classe CCE_Rect alla superclasse `CCE_BasicForm` (dove in precedenza era vuoto), in modo da evitarne la riscrituttra e sfruttando l'ereditarietà.

La distruzione di un gruppo avviene rovesciando le operazioni della costruzione, con alcune avvertenze:

```
- (void )
ungroupAndInsertIn: (CCE_ElemGroup *) masterArray startingFrom: (int) idx
{
    int        i , numElem ;
    CCE_BasicForm * elem ;
    // ciclo su tutti gli elementi del gruppo
    numElem = [ elemArray count ];
    // per ogni elemento presente, lo aggiungo
    for ( i = 0 ; i < numElem ; i ++ )
    {
        NSAffineTransform    * tf1 = [ NSAffineTransform transform ];
        elem = [ elemArray objectAtIndex: i ];
        [ elem setCceIsSelected: YES ] ;
        [ [masterArray elemArray] insertObject: elem atIndex: idx ];
        [ tf1 initWithTransform: localTF ];
        [ tf1 prependTransform: [ elem localTF ] ];
        [ elem setLocalTF: tf1 ] ;
        [ elem setDrawPoint: [ tf1 transformPoint: NSMakePoint(0, 0) ] ];
        [ elem setRotAngle: ( [elem rotAngle] + rotAngle) ];
        [ tf1 invert ];
        [ elem setReverTF: tf1 ];
    }
}
```

Bisogna calcolare una corretta trasformazione per ogni elemento del gruppo. Si tratta di una operazioen contettualmente molto semplice: occorre concatenare in maniera opportuna le due trasformazioni: da SdR della finestra al SdR del gruppo, e da SdR del gruppo al SdR dell'elemento. Per farlo, parto dalla trasformazione da finestra a gruppo (che inserisco in `tf1`) alla quale poi concateno (mettendola davanti col metodo `prependTransform:` in modo che sia eseguita concettualmente per seconda) quella propria dell'elemento. Con questa nuova trasformazione, calcolo l'inversa ed aggiusto insomma il nuovo SdR dell'elemento. Devo tuttavia ricalcolare punto di origine ed angolo: il metodo più semplice per il punto di origine è di ricavarlo dalla trasformazione, mentre per l'angolo semplicemente opero la somma tra l'angolo del gruppo e quello dell'elemento.

Tutto questo meccanismo di costruzione e distruzione dei gruppi funziona perché ho modificato contestualmente anche il metodo di disegno. scompare il metodo `drawRect:`, sostituito anche qui dal metodo `specificDrawing:`. Con il metodo `drawRect:` generico, è effettuata la trasformazione di coordinate e il disegno degli handle. Quindi, all'interno del metodo specifico devo solo disegnare gli elementi:

```
- (void)
specificDrawing: (NSRect) aRect
{
    int        i , numElem ;
    CCE_BasicForm * elem ;

    numElem = [ elemArray count ];
```

```
    for ( i = numElem-1 ; i >= 0 ; i -- )
    {
        elem = [ elemArray objectAtIndex: i ];
        [ elem drawElement: aRect ];
    }
}
```

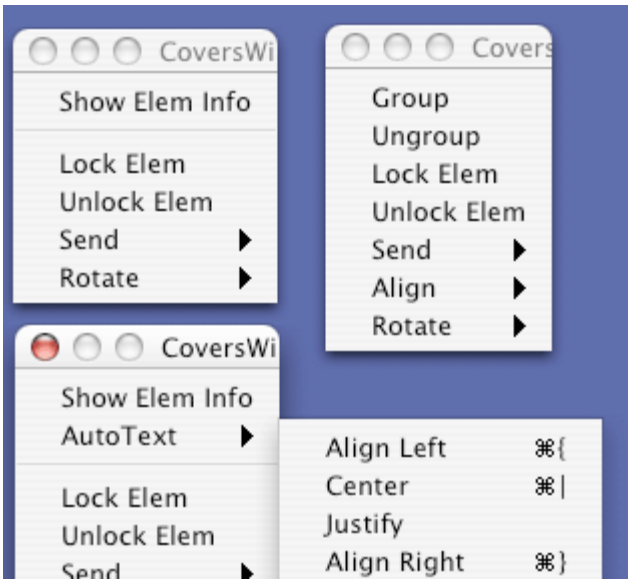
Qui utilizzo il metodo `drawElement:`, in quanto i singoli elementi devono poter effettuare un cambio di coordinate, che adesso, al contrario di quanto succede normalmente, riguarda il passaggio tra il SdR del gruppo e quello dell'elemento proprio.

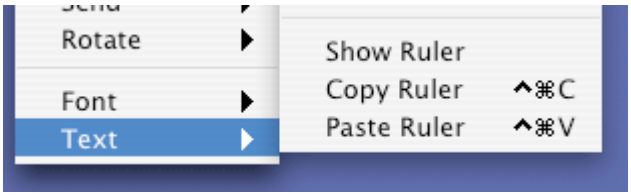
A questo punto, un elemento della classe `CCE_ElemGroup` è trattato in maniera uniforme alle altre classi per quanto riguarda selezione, spostamento e rotazione; posso così eliminare i metodi specifici della classe che sovrascrivevano quelli generici. Ad esempio, per capire se il clic del mouse cade sopra un gruppo, è sufficiente controllare se il clic è contenuto all'interno del rettangolo che racchiude il gruppo. Ciò avviene tranquillamente col metodo generico, visto che tale controllo utilizza solamente il punto di origine e le dimensioni dell'elemento. Parimenti, lo spostamento e la rotazione dell'elemento, modificando solamente la trasformazione dell'elemento gruppo, non interessa più i singoli elementi del gruppo, che mantengono la loro relazione all'interno del SdR del gruppo stesso.

Ecco quindi che la classe `CCE_ElemGroup` non sovrascrive più i metodi di `CCE_BasicForm`, ma aggiunge solamente i metodi specifici della sua natura (manipolazione del gruppo, undo, eccetera).

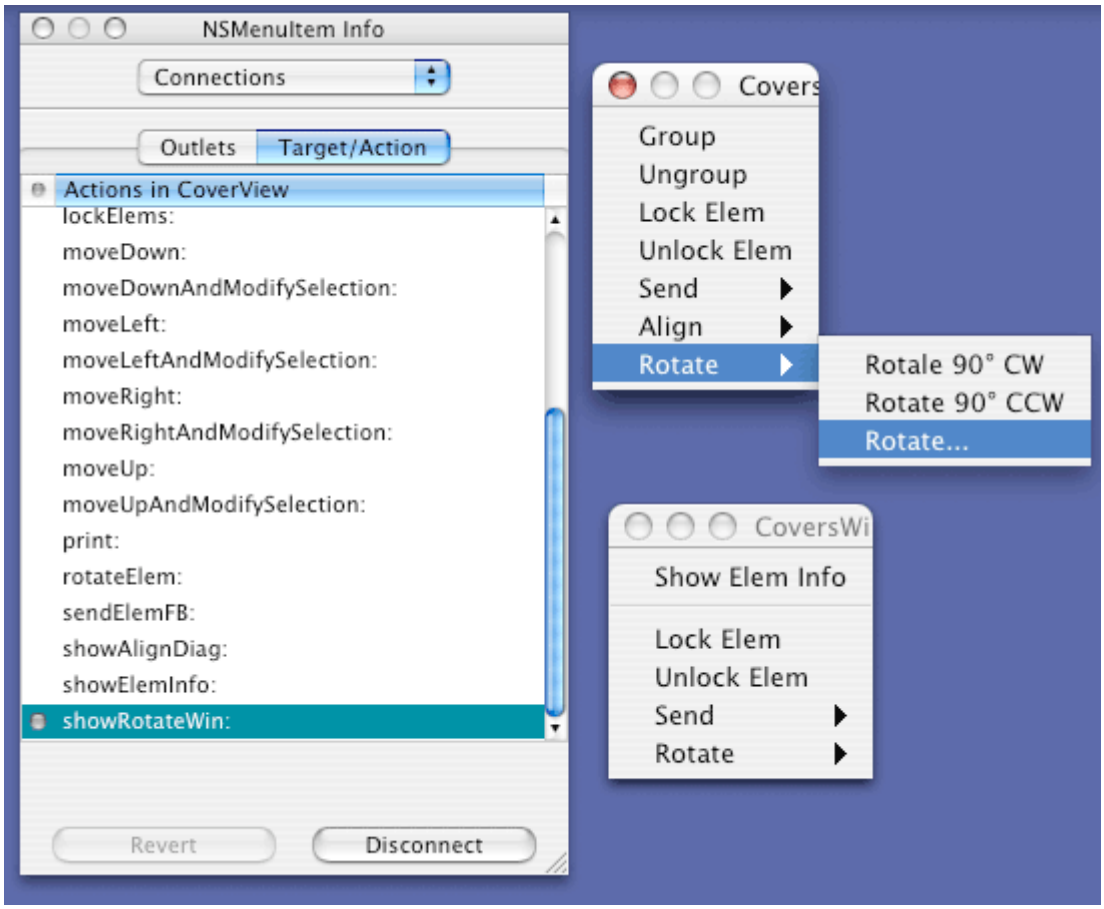
MENU CONTESTUALI

Avere menu contestuali è sempre una bella cosa: semplificano la vita mostrando e permettendo solo le operazioni lecite in quel momento. voglio quindi aggiungere dei menu contestuali all'interno della finestra della copertina, che siano presenti quando sono selezionati uno o più elementi (e quindi, niente menu se non ci sono elementi selezionati). Come primo tentativo, utilizzo tre menu: uno attivo quando è selezionato un singolo elemento, uno quando sono selezionati più elementi, ed un terzo attivo solo quando il singolo elemento selezionato è della classe `CCE_Text`.





Costruisco direttamente questi tre menu all'interno di *Interface Builder*, nel file `CoversWin.nib`. Ho raccolto all'interno del menu le voci che hanno più o meno senso nei tre casi (arriverò più avanti al concetto di testo automatico per `CCE_Text`); ad esempio, per il menu relativo ad una selezione che coinvolge più elementi, è presente la voce che permette la costruzione di un gruppo, assente altrimenti. Aggiungo tre outlet alla `CoverView`, collegati a questi tre menu, per richiamarli quando necessario. Più complicato assegnare le azioni alle voci di menu. Infatti, normalmente i destinatari delle voci dei menu sono il *First Responder* (nel caso, la `CoverView`, e non ci sono quindi problemi); in alcuni casi, tuttavia (quelli che aprono una finestra, come ad esempio la voce *Align...*) il destinatario previsto è un metodo della classe `AppDelegate`. In mancanza di idee migliori, non ho fatto altro che copiare i metodi incriminati all'interno della classe `CoverView`, e rendere la `CoverView` destinataria anche di queste voci di menu. Mi trovo così con alcuni metodi duplicati, ma pazienza.



Per attribuire un menu contestuale, occorre avere a disposizione una istanza di una classe `NSResponder` (o `NSView`, che ne è una discendente). Però, gli elementi `CCE_BasicForm` (e discendenti) non discendono da questa classe. Ciò che si avvicina di più alla classe desiderata è la `CoverView` (che è appunto una sottoclasse di `NSView`), cui giocoforza devo attribuire il menu contestuale. Ce ne fosse uno solo, valido per tutta la

`CoverView`, basta utilizzare il metodo `setMenu:`. Ma questo non è il mio caso; devo invece utilizzare un altro metodo, `menuForEvent:`. Si tratta di un metodo che di default restituisci `nil` (in assenza di menu contestuali) o il menu contestuale impostato con `setMenu:`. Devo sovrascriverlo per fare in modo che restituisca il menu appropriato in base alla posizione del clic dell'utente:

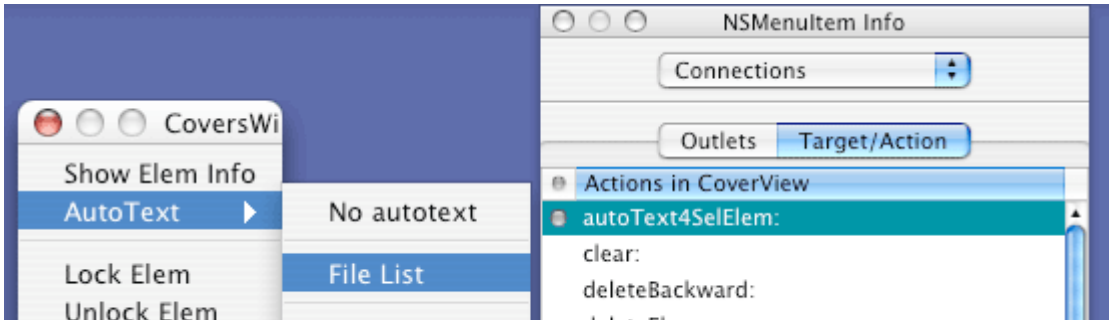
```
- (NSMenu *)
menuForEvent: (NSEvent *)theEvent
{
    NSPoint             whereClick;
    CCE_BasicForm * curElem ;
          whereClick   =   [self   convertPoint:[theEvent   locationInWindow]
fromView:nil];
    curElem = [self getClickedElem:whereClick] ;
    if ( curElem == nil || ( ! [ curElem cceIsSelected]) )
        return ( nil ) ;
    curElem = [ AppDelegate getTheSelectedElem];
    if ( curElem == nil )
        return ( nil ) ;
    else if ( curElem == (CCE_BasicForm*)(-1) )
        return ( grpElemMenu );
    else if ( [curElem isKindOfClass:[CCE_Text class]] )
        return ( txtElemMenu );
    return ( stdElemMenu );
}
```

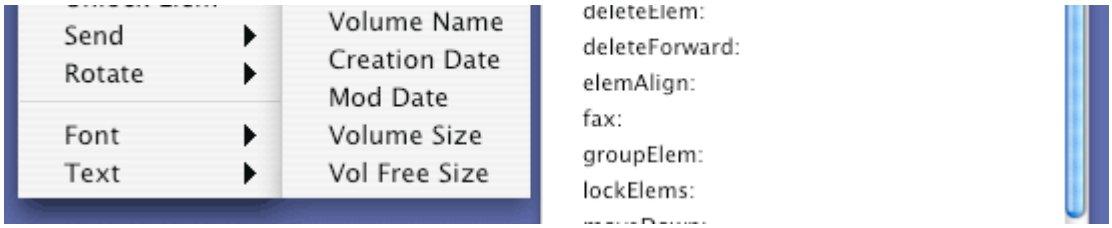
La logica è molto semplice: se il clic avviene fuori da ogni elemento, nessun menu. Se il clic avviene sopra un elemento, ma questo non è selezionato, nessun menu. Solo se il clic avviene sopra un elemento, e questo è selezionato, si comincia a ragionare. Se non è il solo elemento ad essere selezionato, il menu da visualizzare è quello proprio per un insieme di elementi. Se c'è un solo elemento selezionato, ed è della classe `CCE_Text`, mostro il menu specifico per il testo (noto che nel caso di attività di modifica in corso, il menu contestuale visualizzato è quello proprio di una `NSTextView`, diverso da quello da me previsto); altrimenti, mostro il menu standard per il singolo elemento.

Tutto molto semplice e pratico.

TESTO AUTOMATICO

Per fare degli esperimenti, avevo definito un elemento `CCE_Text` come contenitore della lista dei file di un volume. Avevo allo scopo definito una variabile d'istanza `fileList` di `CoverView` che individuasse tale elemento, e metodi specifici della classe `CatDataSrc` in grado di produrre tale lista sotto forma di testo. Il meccanismo era scatenato dalla classe `CatDataSrc`, al variare della selezione del volume corrente.





Voglio estendere questa possibilità, introducendo il concetto di testo automatico: un elemento della classe `CCE_Text` può essere caratterizzato dal fatto che il testo in esso contenuto è prodotto in maniera automatica a partire da informazioni presenti nel catalogo del volume. L'elenco dei file contenuti è una di queste possibilità; altre possibilità sono il nome del volume, la dimensione, la data di creazione, eccetera. Voglio inoltre lasciare all'utente la possibilità di scegliere quale testo inserire: è questo il motivo della presenza, all'interno del menu contestuale specifico degli elementi di testo, di un sottomenu che permette di specificare quale tipo di testo automatico attribuire. Cambia anche la prospettiva: il meccanismo adesso è gestito direttamente dalla classe `CCE_Text`, utilizzando i servizi forniti da `CatDataSrc`.

Parto definendo una nuova variabile d'istanza della classe `CCE_Text`, l'intero `automaticText`, il cui valore discrimina quale tipo di testo automatico è presente; se il valore è nullo, il testo non è automatico, ed è di libero accesso da parte dell'utente (in realtà, anche gli altri testi sono di libera modifica; tuttavia, selezionando un altro volume nella lista, il testo sarà modificato in accordo alle direttive impartite, ed ogni precedente modifica sarà perduta).

All'interno del file `CoverView.m` è quindi presente il metodo che è invocato dalle voci di menu del testo automatico; ogni voce chiama lo stesso metodo, ed ho fatto in modo che ogni voce abbia un tag corrispondente agli identificatori del testo automatico che ho in precedenza definito:

```
#define         CATAUTOTEXT_NULL            0
#define         CATAUTOTEXT_FILELIST        50
#define         CATAUTOTEXT_VOLNAME            51
#define         CATAUTOTEXT_MODDATE            53
#define         CATAUTOTEXT_CREATDATE        52
#define         CATAUTOTEXT_VOLSIZE            54
#define         CATAUTOTEXT_VOLFREESIZE        55

- (IBAction)
autoText4SelElem: (id)sender
{
    int         uot = [ sender tag ] ;
    CCE_Text     * myTxt = (CCE_Text*) [ AppDelegate getTheSelectedElem] ;
    [ myTxt setAutomaticText: uot ];
    [ myTxt updAutoText: [[ winCtl document] dataSource ] ] ;
    // forzo un ridisegno
    [ self setNeedsDisplay: YES ];
}
```

Il metodo è certo che è presente un solo elemento selezionato, e che questo sia di tipo testo: non è al momento possibile chiamare il metodo `autoText4SelElem` se non attraverso menu contestuale, e questo menu compare solo se è selezionato un solo elemento ed è di classe `CCE_Text`. Il metodo imposta la variabile d'istanza secondo il tag del menu, e poi costringe l'elemento ad aggiornare il testo contenuto.

Il metodo `updAutoText:` ha tre realizzazioni distinte. È assolutamente vuoto nel caso generale; passa il messaggio agli elementi del gruppo nel caso di un elemento `CCE_ElemGroup`; contiene istruzioni interessanti nel caso di `CCE_Text`.

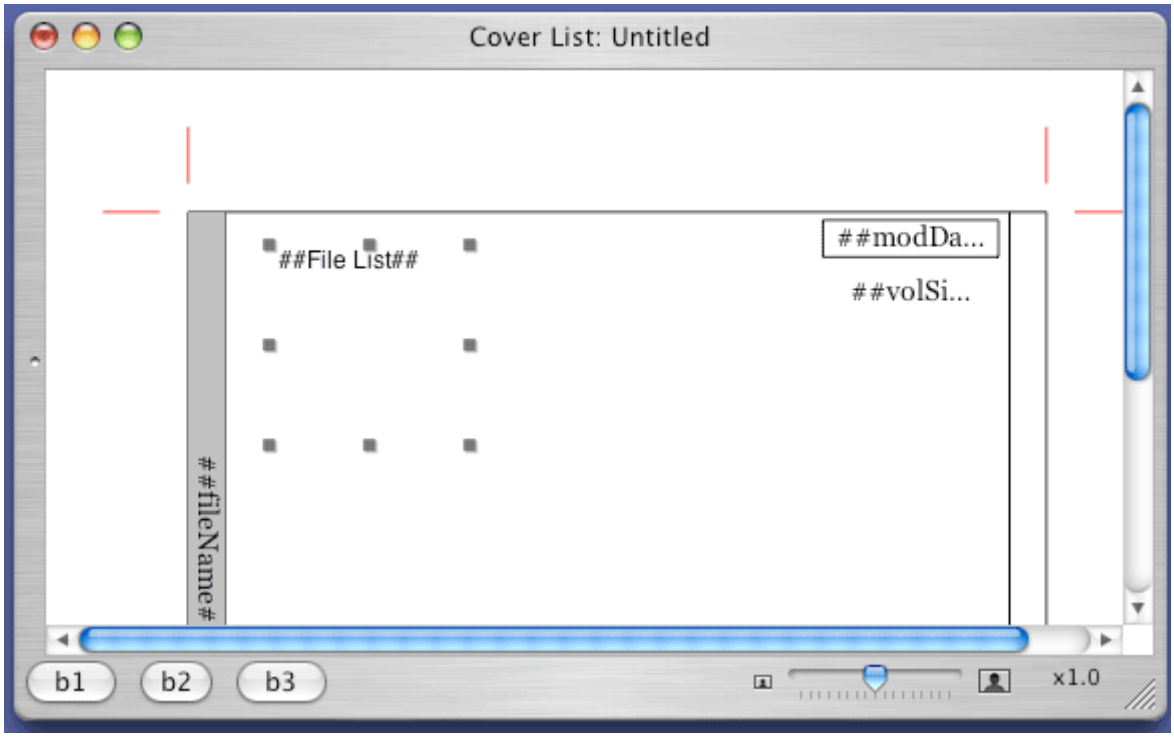
```
- (void)
updAutoText: (CatDataSrc *) catalog
{
    return ;
}

- (void)
updAutoText: (CatDataSrc *) catalog
{
    // passo la palla agli elementi del gruppo
         [  elemArray  makeObjectsPerformSelector:  @selector(updAutoText:)
withObject: catalog ] ;
}

- (void)
updAutoText: (CatDataSrc *) catalog
{
    if ( automaticText == CATAUTOTEXT_NULL )
        return ;
    switch ( automaticText ) {
    case CATAUTOTEXT_FILELIST :
        [ catalog setCACFileListFor: textStorage ] ;
        break ;
    case CATAUTOTEXT_VOLNAME :
        [ catalog setCACAttribFor: textStorage ofName: COLID_FILENAME ] ;
        break ;
    case CATAUTOTEXT_MODDATE :
        [ catalog setCACAttribFor: textStorage ofName: COLID_MODDATE ] ;
        break ;
    case CATAUTOTEXT_CREATDATE :
        [ catalog setCACAttribFor: textStorage ofName: COLID_CREATDATE ] ;
        break ;
    case CATAUTOTEXT_VOLSIZE :
        [ catalog setCACAttribFor: textStorage ofName: @"volSize" ] ;
        break ;
    case CATAUTOTEXT_VOLFREESIZE :
        [ catalog setCACAttribFor: textStorage ofName: @"volFreeSize" ] ;
        break ;
    }
}
```

In realtà il grosso lavoro di produzione del testo è svolto da un paio di metodi di `CatDataSrc`.

Tuttavia, la prima modifica della classe `CatDataSrc` riguarda il metodo chiamato quando cambia la selezione nella tabella che elenca i volumi disponibili:

```
- (void)
tableViewSelectionDidChange:(NSNotification *)aNotification
{
    NSTableView        * ov = (NSTableView*)[ aNotification object ] ;
    CoverView        * cv = [[refDoc coverWin] cvrView] ;
    // predispongo il volume in selezione corrente
    currentVol = [ ov selectedRow ] ;
    [ [[cv theElements] elemArray] makeObjectsPerformSelector:
            @selector(updAutoText:) withObject: self ] ;
```

```
    // forzo un rinfresco della vocerView
    [ cv setNeedsDisplay: YES ];
}
```



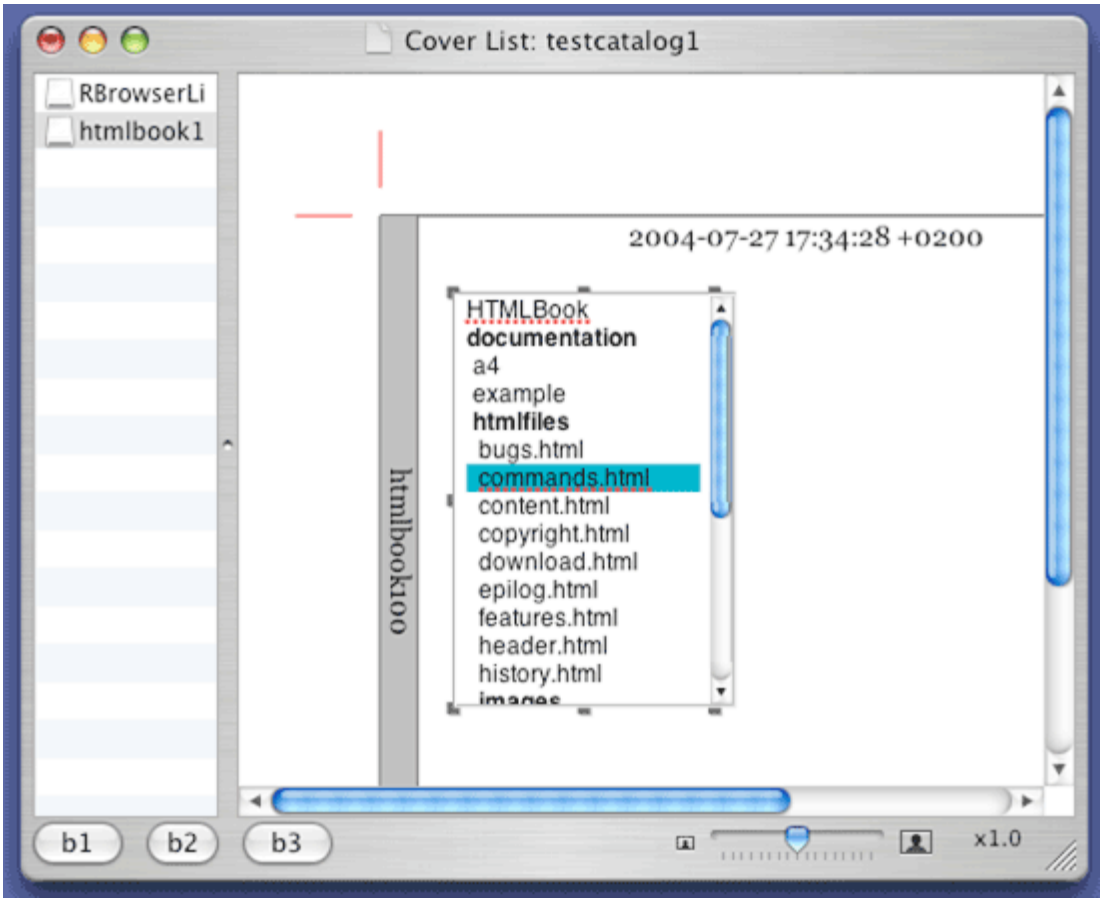
In precedenza, il metodo, oltre ad aggiornare la variabile `currentVol`, suo scopo principale, costruiva un testo con l'elenco dei file e lo attribuiva all'eventuale elemento `CCE_Text` deputato. Adesso, tale operazione non è svolta esplicitamente; piuttosto, è lasciato al gruppo di elementi `theElements` (che è appunto quello che raccoglie tutti gli elementi di una copertina) di eseguire le operazioni di aggiornamento. Qui uso per la prima volta il metodo `makeObjectsPerformSelector:withboject:`. Si tratta di un metodo tipico di un `NSArray` (e discendenti) per far si che ogni elemento del vettore esegua una dato metodo (individuato tramite il costrutto `@selector(.)`) con argomento specificato. Finora ho sempre eseguito questa operazione attraverso un esplicito ciclo for, anche perché quasi sempre mi interessava che l'ordine di esecuzione fosse quello indicato (e che questo metodo non specifica).

Le operazioni di costruzione del testo sono svolte da due metodi; esamino per primo il più semplice.

```
- (void)
setCACAttribFor: (NSTextStorage *) txtSto ofName: (NSString*) colId
{
    VolInfo            * localItem ;
    NSMutableDictionary    * curStyle ;
    NSTextStorage * newFileList ;
    NSString        * localString ;
    id                valloasape ;
    // costruisco un nuovo contenitore per il testo, che calcolo subito
    if ( currentVol == -1 )
    {
        localString = [    NSString stringWithFormat: @"##%@##", colId ] ;
    }
    else
```

```
    {
        localItem = [ startPoint objectAtIndex: currentVol ] ;
        valloasape =[ localItem valueForKey: colId ] ;
             localString = [      NSString stringWithString: [valloasape
description ] ] ;
    }
    newFileList = [[NSTextStorage alloc] initWithString: localString ];
    // adatto il font corrente
    ...
    // aggiorno la stringa
    [ txtSto setAttributedString: newFileList ] ;
}
```

Questo metodo deve produrre un testo in base all'argomento specificato; questo argomento è uno degli identificatori delle colonne per la presentazione dei dati, estesamente utilizzati per le `NSOutlineView`.



Il metodo verifica che ci sia un volume correntemente selezionato; in caso contrario, produce un testo di default con il nome del campo richiesto. Per produrre il testo in condizioni normali, ricorro al metodo `description`, che velocemente e senza troppi patemi produce una descrizione sotto forma di stringa del contenuto della colonna desiderata. Salto alcune istruzioni intermedie (argomento del prossimo paragrafo), per chiudere con l'ultima istruzione, che sostituisce il contenuto dell'istanza di `NSTextStorage` con la nuova stringa calcolata.

Simile nel concetto ma un po' più laborioso l'altro metodo, che recupera dalla vecchia realizzazione la sequenza di costruzione della lista dei file:

```
- (void)
setCACFileListFor: (NSTextStorage *) txtSto
{
    NSMutableDictionary    * curStyle ;
    NSTextStorage * newFileList ;
    int                elNum, jj, nn ;
    if ( currentVol == -1 )
    {
              newFileList  =  [[NSTextStorage  alloc]  initWithString:@"##Fil
List##"];
    }
    else
    {
        // costruisco un nuovo contenitore per il testo
        newFileList = [[NSTextStorage alloc] initWithString:@""];
    }
    ...
    // recupero il primo livello del volume
    if ( currentVol == -1 )
    {
        // aggiorno la stringa
        [ txtSto setAttributedString: newFileList ] ;
        return ;
    }
    // se arrivo qui, ho un vero e proprio volume
    elNum = [ self outlineView: nil numberOfChildrenOfItem: nil ];
    // esploro il primo livello
    for ( jj = 0 ; jj < elNum; jj ++ )
    {
        FileStruct * elem ;
        // esamino tutti gli elementi uno ad uno
        elem = [ self outlineView:     nil child: jj ofItem: nil ];
        // controllo se ha figli
        nn = [ self outlineView: nil numberOfChildrenOfItem: elem ];
        // se non e' da stampare, salto
        if ( ! [ elem is2Print ] ) continue ;
        // se e' una directory, espando di un livello
        if (nn > 0 )
              addStringExpandItem (newFileList, curStyle, self, elem, nn, 0
);
        else    // lo aggiungo al testo gia' predisposto
             terminalWithLevel( newFileList, curStyle, [ elem fileName], 0,
NO );
    }
    // aggiorno la stringa
    [ txtSto setAttributedString: newFileList ] ;
}
```

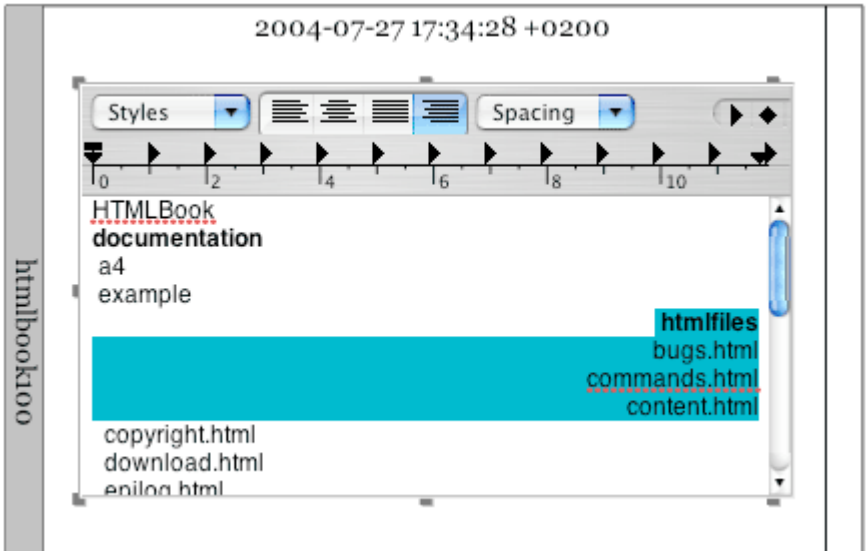
Rispetto alla versione precedente, c'è una importante modifica, riguardante la caratterizzazione del testo. Prima il testo aveva lo stile caratterizzato in maniera esplicita attraverso degli attributi (cosa che richiedeva anche un brutto espediente per impostarli su una stringa ancora vuota); qui invece utilizzo un `NSDictionary` per impostare una serie di attributi ai paragrafi.

LO STILE DEI PARAGRAFI

La cosa migliore per impostare lo stile di un testo è utilizzare appunti uno *Stile*. Lo Stile è un concetto che tutti gli elaboratori di testo hanno fatto conoscere, come una serie di attributi caratterizzanti l'aspetto di un testo: il paragrafo, l'allineamento, margini,

eccetera. Anche Cocoa mette a disposizione questo meccanismo: la sua realizzazione più immediata la trovate nell'applicazione *TextEdit*, nella barra superiore di ogni finestra.

Uno stile è realizzato attraverso la classe `NSParagraphStyle` (e discendenti); tra le caratteristiche che è possibile impostare di un paragrafo, ci sono: l'allineamento del testo (a destra, centrato, eccetera), l'interlinea, i rientri (a destra, a sinistra, specifico della prima linea di testo), i punti di tabulazione, eccetera.



La gestione di queste caratteristiche di testo può avvenire ad esempio attraverso un menu standard, predefinito in *Interface Builder*: è il menu `Text`, che ho aggiunto, così com'è al mio menu standard (e d anche contestuale). Notare che nei menu sono presenti anche delle voci per visualizzare e nascondere i righelli, gli stessi righelli che si trovano nell'applicazione TextEdit (anche se all'interno di un elemento `CCE_Text` sono piuttosto infelici a vedersi).

Nel caso, raccolgo un po' di queste caratteristiche, le unisco ad una specifica di carattere, e costruisco un `NSDictionary` col metodo seguente:

```
- (NSDictionary *)
getTextAttr
{
     NSFont                    * locFont = [ NSFont fontWithName: @"Georgia"
size: 14 ];
      NSMutableDictionary *textAttributes = [[[NSMutableDictionary alloc]
initWithCapacity:2] autorelease];
     NSMutableParagraphStyle *paraStyle = [[NSMutableParagraphStyle alloc]
init];
    [ paraStyle setAlignment: NSCenterTextAlignment ] ;
    [ paraStyle setLineBreakMode: NSLineBreakByTruncatingTail ];
    [textAttributes setObject: locFont forKey:NSFontAttributeName];
                          [textAttributes       setObject:       paraStyle
forKey:NSParagraphStyleAttributeName];
    return ( textAttributes );
}
```

Nel dizionario ho dunque specificato di utilizzare il carattere *Georgia*, 14 punti, allinearlo al centro, e, nel caso in cui la linea occupi più spazio di quanto visualizzabile, troncarne la visualizzazione omettendo gli ultimi caratteri. Inserisco il tutto dentro un

dizionario, che poi utilizzo, ad esempio, in sede di costruzione di un elemento di testo:

```
...
            temp = [ [ NSMutableAttributedString alloc] initWithString:
@"Volume Name" ];
          [ temp setAttributes: [ self getTextAttr] range: NSMakeRange(0,
[temp length])];
            tmpText = [[ CCE_Text alloc] initWithId: [self getNextObjId]
inView: self
            withAttributedString: temp
                                inRect:  NSMakeRect(CM2PX(x1),  CM2PX(y1),
CM2PX(BACK_LABEL_VER_DIM), CM2PX(BACK_LABEL_HOR_DIM) )
            withAngle: 90 ];
        [ tmpText setAutomaticText: CATAUTOTEXT_VOLNAME ] ;
...
```

Con il precedente frammento di codice, costruisco una stringa, imposto i precedenti attributi di testo, e poi utilizzo questa stringa per definire un elemento `CCE_Text`, il quale, guarda caso, sarà un testo a riempimento automatico (col nome del volume).

Quando, all'interno dei metodi di `CatDataSrc` precedenti, produco il testo automatico, recupero questi attributi (in realtà, i primi che trovo, che l'utente potrebbe aver combinato un po' di modifiche sul testo rappresentato) e li impongo al nuovo testo prodotto; ad esempio, all'interno di `setCACFileListFor` ci sono le seguenti istruzioni (che vanno a rimpiazzare ciò che poco sopra erano i tre puntini di sospensione):

```
...
    curStyle = [ NSMutableDictionary dictionaryWithCapacity: 2 ] ;
    [ curStyle addEntriesFromDictionary:
        [ txtSto fontAttributesInRange: NSMakeRange(0, [txtSto length]) ] ]
;
    [ curStyle addEntriesFromDictionary:
        [ txtSto rulerAttributesInRange: NSMakeRange(0, [txtSto length]) ]
];
...
```

Costruisco un dizionario, e ci inserisco la specifica di font e la specifica di righello (al momento, lo stile del paragrafo) del testo che intendo sostituire.

Quando poi, all'interno della funzione `terminalWithLevel` devo aggiungere una riga di testo, utilizzo proprio questo dizionario (che mi sono premunito di passare come parametro alla funzione):

```
void
terminalWithLevel ( NSTextStorage* str, NSMutableDictionary * curStyle,
NSString *fname, int lev, BOOL boldface )
{
    NSString        * xxx ;
    NSMutableAttributedString * temp ;
    NSFont            * locFont ;
        NSMutableDictionary          *  tmpD  =  [  NSMutableDictionary
dictionaryWithDictionary: curStyle ] ;

    // costruisco la stringa col nome del file
     xxx = [ NSString stringWithFormat: @"%@%@\n", prefixLevString( lev ),
fname ] ;
    // la trasformo in una AttributedString
```

```
    temp = [ [ NSMutableAttributedString alloc] initWithString: xxx ];
    // recupero il font attualmente in uso
    locFont = (NSFont*) [ tmpD objectForKey: NSFontAttributeName ] ;
    // se il caso, aggiungo il tratto Grassetto alla stringa
    if ( boldface )
         locFont = [ [NSFontManager sharedFontManager] convertFont: locFont
toHaveTrait: NSBoldFontMask];
    // assegno gli attributi alla stringa
    [ tmpD setObject: locFont forKey:NSFontAttributeName];
    [ temp setAttributes: tmpD range: NSMakeRange(0, [temp length])];
    // e poi l'aggiungo all'ambaradan
    [ str appendAttributedString: temp ] ;
}
```

La prima cosa che faccio è una copia del dizionario, in quanto successivamente potrei aver bisogno di modificarlo localmente (per mettere in grassetto la directory, ad esempio). L'istruzione `setObject:forKey:` rimpiazza la descrizione di font corrente con quella specifica del caso, e poi l'istruzione con `setAttributes:range:` impone alla stringa le caratteristiche desiderate.

Così facendo, quando si deve aggiornare il testo automatico, non sono perse le eventuali formattazioni (diverse da quelle standard) che l'utente ha effettuato sul testo.

Capitolo 055 - Immagina...

Aggiungo (o meglio, ritorno ad aggiungere) un elemento in grado di rappresentare una immagine (jpeg, gif, pict, eccetera) all'interno di una copertina.

Sorgente: la solita documentazione

Prima stesura: 29 agosto 2004

ELEMENTI IMMAGINE

All'inizio della storia della costruzione della finestra con la copertina avevo già introdotto l'elemento `CCE_Image`, che poi avevo barbaramente ucciso il capitolo successivo. Adesso riprendo in mano la questione, perfezionando il tutto.

Per prima cosa, apro in *Interface Builder* la palette degli strumenti della finestra e attivo l'ultimo strumento rimasto, cambiando leggermente l'icona in modo che rappresenti una immagine (data la mia classica incapacità di disegno, copio brutalmente l'icona dell'applicazione *Anteprima* e la ridimensiono come più mi piace). Modifico anche leggermente il metodo

```
- (IBAction)
handleToolSel:(id)sender
{
    switch ( [ [toolMatrix selectedCell] tag ] ) {
    case 0 : currCCE = nil ; break ;
    case 1 : currCCE = [ CCE_Image class ] ; break ;
    case 2 : currCCE = [ CCE_Line class ] ; break ;
    case 3 : currCCE = [ CCE_Rect class ] ; break ;
    case 4 : currCCE = [ CCE_Circle class ]; break ;
    case 5 : currCCE = [ CCE_Text class ] ; break ;
    }
}
```

in modo che restituisca la classe `CCE_Image` nel posto corretto.

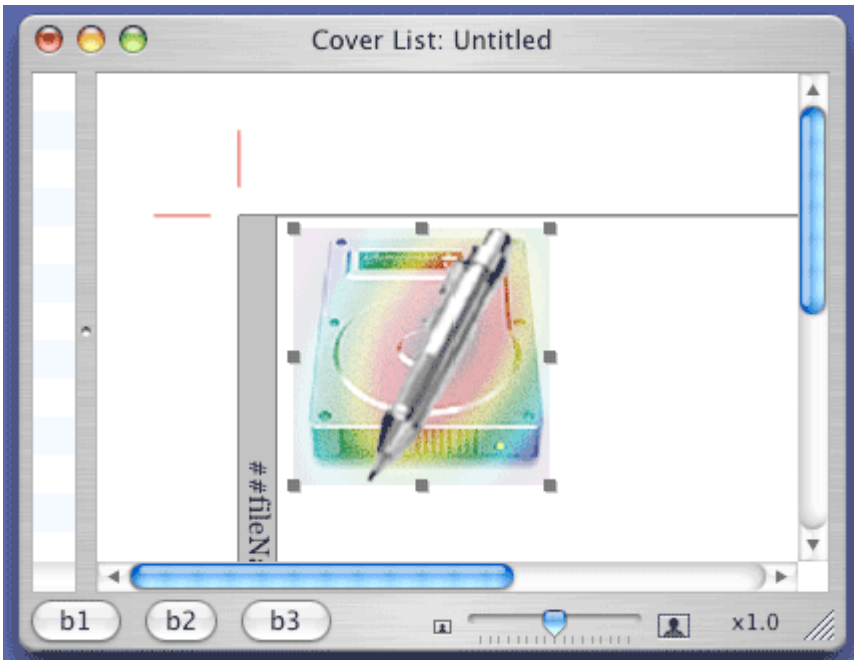
L'idea per l'inserimento di un elemento di tipo immagine è che l'utente tracci un bel rettangolo dove prevede di inserire una immagine; immediatamente dopo, gli viene proposto un dialogo per la scelta di un file, che individua appunto l'immagine da

inserire.

Però, per iniziare, occorre costruire la classe `CCE_Image` e scrivere un po' di metodi. Comincio con le variabili d'istanza:

```
@interface      CCE_Image : CCE_Rect
{
    NSImage         * image ;
    NSSize         imgRepDim ;
    float         compFraction ;
}
```

Le variabili sono tre; con la prima variabile conservo un oggetto della classe `NSImage`. Questa classe è piuttosto ricca, e permette di conservare più o meno ogni tipo di immagine leggibile con *QuickTime* e con l'applicazione *Anteprima*: quindi JPEG, GIF, PICT, EPS, TIFF, eccetera. Per i miei scopi è più che sufficiente. Conservo poi esplicitamente in una variabile di tipo `NSSize` le dimensioni originali dell'immagine; mi servono per dare una buona rappresentazione dell'immagine stessa quando si tratta di disegnarla all'interno della `CoverView`. La terza variabile rappresenta il *canale alfa*, o il fattore di *trasparenza* dell'immagine. Un valore di 1 per la variabile significa che l'immagine è del tutto opaca, e nasconde ogni pixel dietro di essa. Un valore di zero dice che l'immagine è completamente trasparente, mentre un valore intermedio compone l'immagine con lo sfondo combinando i pixel. L'effetto risultate è un'immagine più o meno trasparente, che lascia intravedere lo sfondo.



I metodi di inizializzazione (uno ricco ed uno più povero) utilizzano una immagine di default per la visualizzazione; scelgo l'icona dell'applicazione utilizzando uno degli identificatori predefiniti del metodo `imageNamed:`.

```
- (id)
initWithId: (int) ident inView: (CoverView*) cv
    withRect: (NSRect) aRect andAngle: (float) angle
{
```

```
    self = [ super    initWithId: ident inView: cv
                   withRect: aRect andAngle: angle ] ;
    if ( self )
    {
        // imposto una immagine di default
        [ self setImage: [ NSImage imageNamed: @"NSApplicationIcon" ] ];
        // metto a posto la dimensione originale ed la trasparenza
        imgRepDim = [ image size ];
        compFraction = 1.0F ;
        // l'immagine ha le coordinate rovesciate come la view
        [ image setFlipped: YES ];
    }
    return self ;
}

- (id)
init
{
    return( [ self    initWithId: 0 inView: nil
                   withRect: NSMakeRect(100,100, 50,50)
                   andAngle: 0] );
}
```

Il metodo di inizializzazione è necessario in pratica per predisporre le variabili d'istanza proprie della classe.

Visto che si costruisce un oggetto `NSImage` all'interno dei metodi di inizializzazione, occorre scrivere anche un metodo `dealloc`:

```
- (void)
dealloc
{
    [ image dealloc ];
    [ super dealloc ];
}
```

IMPOSTAZIONE E DISEGNO

Ora, torniamo alla procedura di inserimento di una immagine. Non ci sono metodi da modificare, basta utilizzare il meccanismo di costruzione presente in `CoverView` (all'interno del metodo `mouseDown:`) e sfruttare l'invocazione del metodo `startEditing` per completare le operazioni:

```
- (void)
startEditing
{
    // tutti i tipi di immagine leggibili
    NSArray        * fileTypes = [NSImage imageFileTypes ];
    // il pannello di selezione file
    NSOpenPanel * oPanel = [NSOpenPanel openPanel];
    NSImage        * curImg ;    // l'immagine prescelta
    NSString    * aFile ;    // il path dell'immagine
    // dico che si puo' selezionare un solo file alla volta
    [oPanel setAllowsMultipleSelection:NO];
    // apro il pannello e consento la selezione
    if ( [oPanel runModalForTypes:fileTypes] != NSOKButton)
        return ;
```

```
    // se arrivo qui, e' stato selezionato un file
    // recupero il path del file
    aFile = [ [oPanel filenames] objectAtIndex:0];
    // costruisco l'immagine a partire dal file
    curImg = [[ NSImage alloc] initWithContentsOfFile: aFile ];
    // rovescio le coordinate, visto che anche la CoverView lo fa
    [ curImg setFlipped: YES ];
    // imposto l'elemento immagine con le nuova immagine
    [ self setImage: curImg ];
    // le nuove dimensioni
    [ self setImgRepDim: [ curImg size]];
    // forzo un ridisegno della vista
    [ ownerView setNeedsDisplay: YES ];
}
```

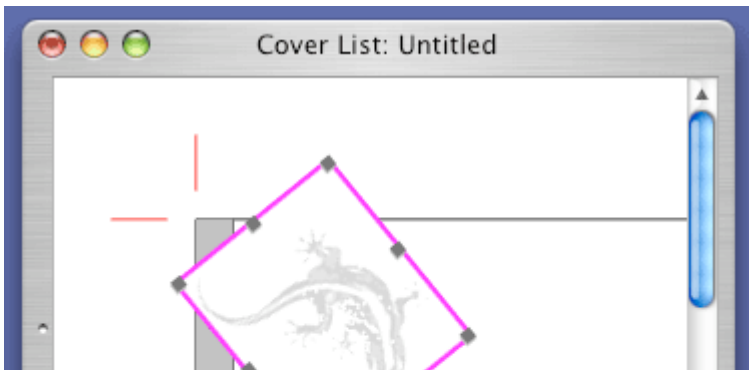
All'interno di questo metodo apro un panello per la selezione di un file di tipo grafico. Mi faccio direttamente specificare dalla classe `NSImage` quali siano i tipi di file in grado di essere gestiti. Il vettore contiene (nel mio caso) ottanta diversi elementi; in realtà i tipi differenti di file sono meno. Sono comunque compresi: PDF, PICT, EPS, PS, icone, JPEG, TGA, TARGA, RGB, SGI, MacPaint, PNG, GIF, BMP e TIFF (e questi sono solo i formati che conosco...). A partire dal file selezionato, costruisco un oggetto `NSImage`, che poi attribuisco all'elemento `CCE_Image`. Metto a posto le dimensioni e forzo il ridisegno.

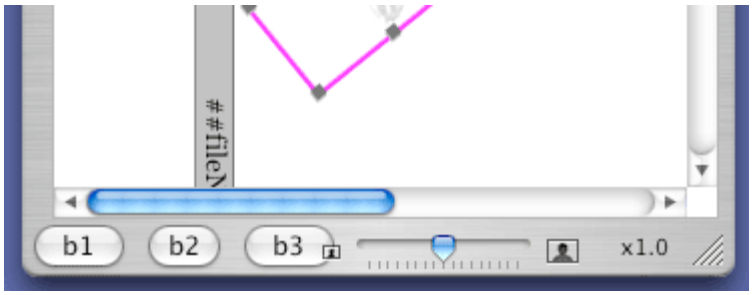
Non occorre un metodo `endEditing`, che non avrebbe nulla da fare.

Rimane il metodo per il disegno vero e proprio.

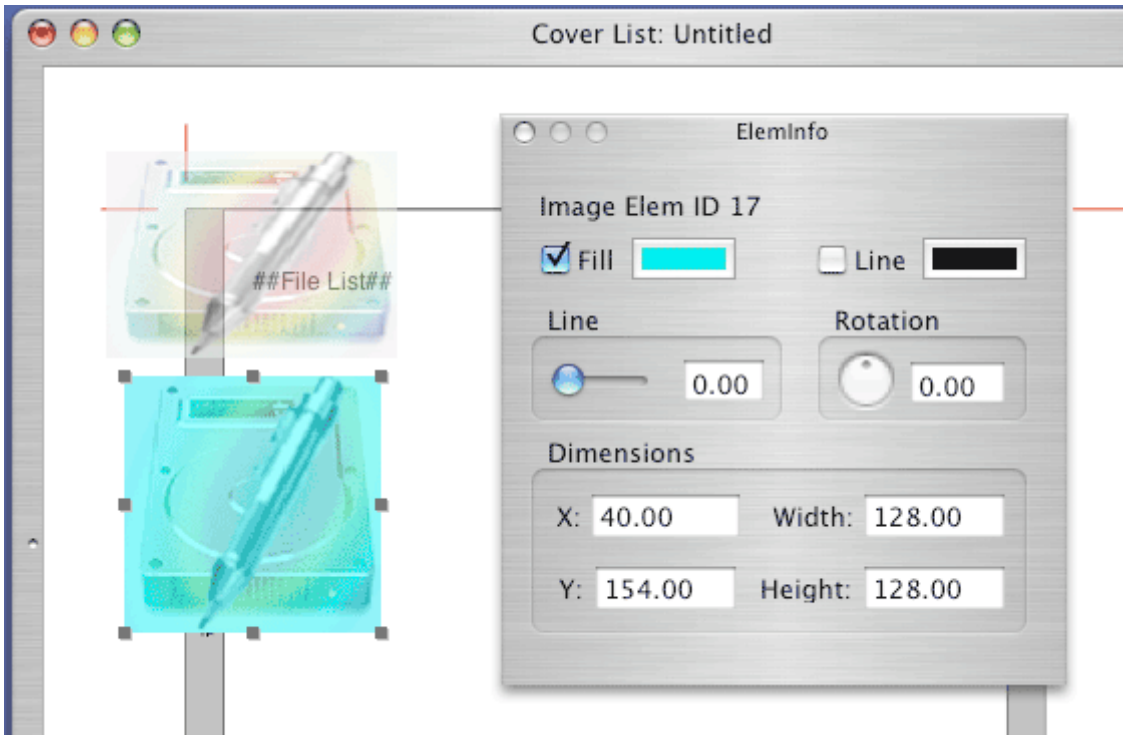
```
- (void)
specificDrawing: (NSRect) inRect
{
    // aggiusto le dimensioni per non ingenerare confusione
    NSRect locrect = arrangeRect( localSize );
    // disegno il rettangolo e lo sfondo
    [ super specificDrawing: inRect ] ;
    // disegno l'immagine scalandola opportunamente
    [ image drawInRect: locrect
            fromRect: NSMakeRect(0,0, imgRepDim.width, imgRepDim.height)
            operation: NSCompositeSourceOver fraction: compFraction ];
}
```

Come prima operazione c'è sempre bisogno di aggiustare il rettangolo in cui disegnare l'immagine, esattamente nello stesso modo in cui ho operato nell'elemento `CCE_Text`. Quindi, sposto la funzione `arrangeRect(.)` dal file `CCE_Text.m` e la inserisco nel file `djZeroUtils.m`, dove vanno a finire tutti quei pezzi di codice che non sono specifici di una classe.





Con l'invocazione del metodo di disegno della superclasse eseguo il disegno del percorso che inscrive l'immagine ed eventualmente il riempimento con il colore di sfondo. Poi eseguo il disegno vero e proprio dell'immagine; qui la classe `NSImage` mette a disposizione diversi metodi, con alcune differenze significative. Tra le varie possibilità, ho scelto il metodo `drawInRect:fromRect:operation:fraction:`; il metodo utilizza due rettangoli. Il primo rettangolo indica il rettangolo destinazione, ovvero il rettangolo in cui deve essere disegnata l'immagine. L'immagine però è disegnata limitatamente a quanto specificato dal secondo rettangolo. Nel mio caso, per il primo rettangolo ho indicato l'effettivo rettangolo che racchiude l'immagine all'interno della `CoverView` (così come aggiustato dalla funzione `arrangeRect(.)`, mentre come secondo rettangolo ho specificato quello che specifica l'intera immagine. In questo modo si ottiene un ridimensionamento dell'intera immagine alle dimensioni specificate dal rettangolo indicato dall'utente.



Gli altri due parametri del metodo indicano come il disegno dell'immagine interagisce col resto del disegno. Il parametro indicato da operation specifica come l'immagine sorgente (cioé quella su cui opera il metodo) si combina con l'immagine destinazione (nel caso, il contenuto della `CoverView`). Piuttosto che dilungarmi, vi rimando all'esempio CompositeLab presente nei file installati con XCode (in /Developer/Examples/AppKit), che mostra tutte le varie possibilità combinatorie (ahimé senza considerare la trasparenza).

Infine il quarto argomento specifica il valore di trasparenza dell'immagine sorgente. Per come utilizzo il metodo, ci sono tre livelli di disegno: la `CoverView`, lo sfondo dell'elemento `CCE_Image`, l'immagine stessa, della classe `NSImage`. La trasparenza permette di intervenire sull'opacità della `NSImage` rispetto allo sfondo, che è sempre disegnato opaco (se però lo sfondo non c'è, allora la trasparenza avviene rispetto alla `CoverView`).

Vorrei che apprezzaste il fatto che l'elemento `CCE_Image` può essere spostato, ruotato e ridimensionato senza dover mettere mano a metodi specifici, ma utilizzando sempre e solo i metodi standard di `CCE_BasicForm`.

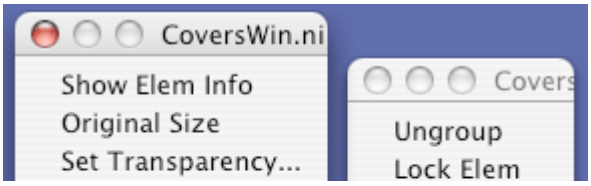
ANCORA SUI MENU CONTESTUALI

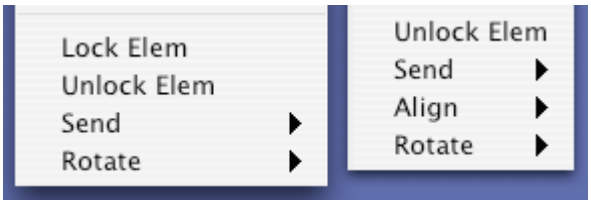
Prima di continuare con i metodi propri della classe `CCE_Image`, devo inserire un intervallo e riparlare dei menu contestuali. La versione precedente non è particolarmente soddisfacente; in primo luogo perché c'è qualche voce errata, ed in secondo luogo perché devo complicare il metodo che sceglie il menu ogni volta che un elemento ha bisogno di un menu specifico. Meglio lasciar fare le cose all'elemento selezionato stesso.

```
- (NSMenu *)
menuForEvent: (NSEvent *)theEvent
{
    NSPoint            whereClick;
    CCE_BasicForm * curElem ;
          whereClick   =   [self   convertPoint:[theEvent   locationInWindow]
fromView:nil];
    curElem = [self getClickedElem:whereClick] ;
    if ( curElem == nil || ( ! [ curElem cceIsSelected]) )
        return ( nil ) ;
    curElem = [ AppDelegate getTheSelectedElem];
    if ( curElem == nil )
        return ( nil ) ;
    else if ( curElem == (CCE_BasicForm*)(-1) )
        return ( mltElemMenu );
    return ( [ curElem contextMenu ]);
}
```

Risolti dunque i casi semplici (nessun elemento selezionato, troppi elementi selezionati), ho scritto un metodo proprio della classe `CCE_BasicForm` che producesse il menu più adatto alla situazione. La versione standard del metodo produce il menu standard:

```
- (NSMenu*)
contextMenu
{
    return ( [ ownerView stdElemMenu ] );
}
```





Per i casi specifici, vale a dire `CCE_Text`, `CCE_ElemGroup` ed il nuovo `CCE_Image`, c'è una versione particolare, in cui il menu restituito è rispettivamente `txtElemMenu`, `grpElemMenu` e `imgElemMenu`. Va da sé che ho dovuto aggiungere i menu nel file `CoversWin.nib`, le variabili d'istanza e i relativi metodi accessor nella classe `CoverView`. Adesso sono presenti due nuovi menu: il primo menu è attivo quando l'unico elemento selezionato è un gruppo (ed ha quindi in particolare la voce di *Ungroup*, tolta dal menu che compare quando ci sono più elementi selezionati); il secondo menu è quello relativo ad un elemento `CCE_Image`. Qui sono presenti due nuovi voci, legate alla gestione delle dimensioni dell'immagine ad all'impostazione del valore di trasparenza.

DIMENSIONI ORIGINALI

Per come ho costruito il metodo di disegno, la manipolazione tramite handle di un elemento immagine ridimensiona l'immagine, cambiandone larghezza ed altezza. È facile arrivare in tal modo a rappresentazioni estreme; conviene permettere all'utente il ridimensionamento dell'immagine alle dimensioni naturali dell'immagine stessa, così come desunte dal file che la contiene. Allo scopo ho approntato la voce di menu nel menu contestuale, che chiama un metodo della `CoverView`:

```
- (IBAction)
setOriginalSize: (id)sender
{
    CCE_Image    * myImg = (CCE_Image*) [ AppDelegate getTheSelectedElem] ;
    [ myImg setOriginalSize ] ;
    [ self setNeedsDisplay: YES ];
}
```

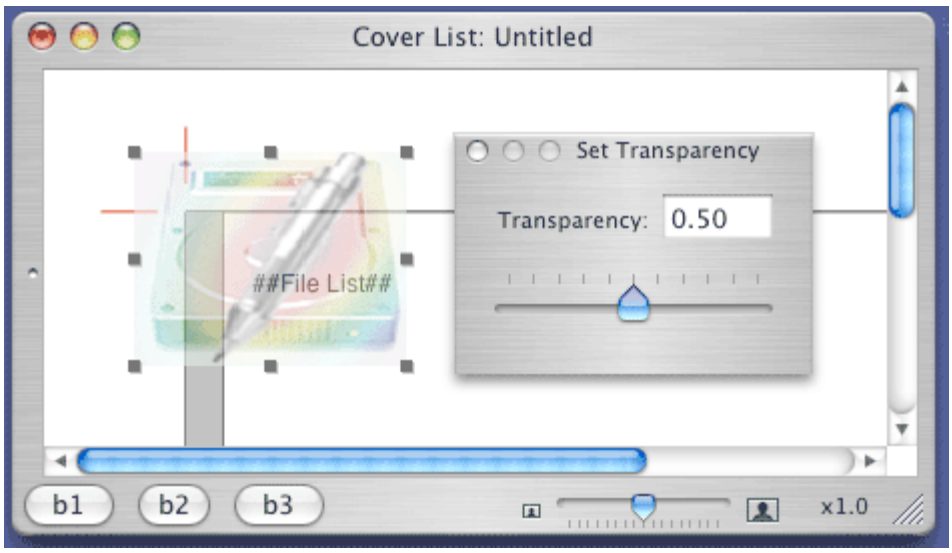
Questo metodo, a sua volta, chiama un metodo proprio dell'elemento, che è normalmente nullo (ancora una volta, per il momento). Nel caso di un elemento `CCE_Image`, il metodo ha invece alcune istruzioni significative:

```
- (void)
setOriginalSize
{
    NSRect        aRect ;
    // reimposto le dimensioni dell'immagine a quelle originali
    [ self setLocalSize: imgRepDim ];
    // ricostruisco il percorso
    aRect = NSMakeRect( 0, 0, localSize.width, localSize.height );
    [ self setTheDrawPath: [NSBezierPath bezierPathWithRect: aRect ] ];
    // e ricalcolo le trasformazioni e gli handle, che son cambiati
    [ self calcLocalTranform ];
    [ self buildHdlList ];
}
```

In pratica, recupero le dimensioni originali, che conservo nella variabile `imgRepDim`, e le utilizzo per modificare la dimensione corrente dell'elemento. Tutto ciò è facile e veloce, anche se ha qualche controindicazione (visiva, non funzionale) quando l'elemento è stato ridimensionato con insistenza.

TRASPARENZA

La seconda voce del menu contestuale ad un elemento `CCE_Image` serve per gestire il valore della trasparenza. Questo è un numero compreso tra Zero (elemento completamente trasparente, quindi invisibile) e Uno (elemento opaco, nessuna trasparenza).



Utilizzo la pratica consolidata: apro una finestra (che ho costruito tramite *Interface Builder* nel file `TranspWin.nib`), dove c'è un campo testo ed uno slider per impostare il valore. La classe controllore associata è `TranspWinCtl`, che è costruita sulla falsariga delle altre finestre accessorie (come ad esempio `RotatewinCtl`). Ho aggiunto la finestra `TranspWin` nella gestione unificata degli aggiornamenti all'interno della classe `AppDelegate` (metodo `updateCoverViewInfo`), ed in generale ho ripetuto le solite operazione spesso descritte.

Riporto quindi solo un metodo di questa classe, quello che imposta il nuovo valore di trasparenza:

```
- (IBAction)
setTransparency:(id)sender
{
    float        newComp = [ sender floatValue ];
    CoverView * cv = [AppDelegate getTheDrawView ] ;
    CCE_BasicForm    * selImg = [AppDelegate getTheSelectedElem ] ;

    [ transpSlider setFloatValue: newComp ] ;
    [ transpText setFloatValue: newComp ] ;
    // non dovrebbe mai succedere...
    if ( cv == nil ) return ;
    if ( selImg == nil || ( selImg == (CCE_BasicForm*)(-1) ) ) return ;
    if ( [selImg isKindOfClass:[CCE_Image class]] )
    {
```

```
        CCE_Image    * selez = (CCE_Image*)selImg ;
        [ selez setCompFraction: newComp ] ;
        [ [ selez ownerView] setNeedsDisplay: YES ];
    }
}
```

L'operazione è svolta quando l'unico elemento selezionato è proprio della classe `CCE_Image`, e si limita ad impostare la variabile d'istanza e a forzare un rinfresco della finestra.

Capitolo 056 - Template

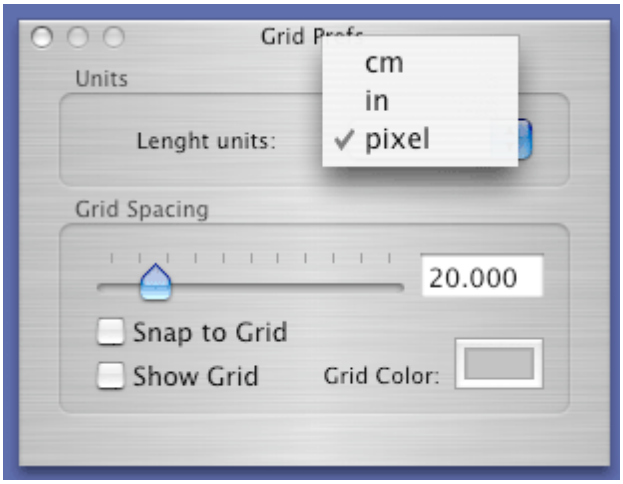
Fino ad ora non c'era la possibilità di salvare il disegno di una copertina. Né in effetti ci sarà; ciò che questo capitolo introduce è il concetto di *template*: con un template indico l'insieme di parametri che caratterizzano il disegno di una copertina (ovvero, gli elementi presenti, la loro disposizione e la loro caratterizzazione). L'utente può salvare un template, e riutilizzarlo poi per disegnare una data copertina. Nel fare tutto ciò, incontro una serie di problemi, che ovviamente risolvo.

Sorgenti: documentazione

Prima stesura: 1 settembre 2004.

UNITÀ DI MISURA

Fino ad ora, le unità di misura degli elementi sono sempre stati i *pixel*, nella quantià di 72 pixel per pollice (oppure 28 e rotti per centimetro). In realtà, è piuttosto scomodo (per me) ragionare in pixel, soprattutto quando voglio poi ottenere una stampa della copertina, da ritagliare ed inserire in un *jewel box* o *slim case* (che riesco a misurare in centimetri). Mi prefiggo quindi di permettere all'utente di scegliere le unità di misura correnti, e di presentare poi all'utente solo misure espresse nelle unità presente.



Per prima cosa, certo un meccanismo perché l'utente possa scegliere le unità. Scelgo di modificare il pannello in cui si imposta la griglia, e lo modifico di conseguenza aggiungendo un menu pop up.

Ho deciso (per il momento) di utilizzare tre unità di misura: i *centimetri*, i *pixel* e i *pollici*. Queste tre unità sono appunto quelle presenti nel pannello. Ho definiti tre costanti per caratterizzarle, ed ho attributo i loro valori come tag nelle tre voci di menu. In questo modo, il metodo che risponde ad una operazione dell'utente sopra il controllo è molto semplice:

```
#define         UNITS_CM        101
#define         UNITS_INCHES    102
#define         UNITS_PIXELS    103
```

```
- (IBAction)
updateUnits:(id)sender
{
    int        newU = [[ sender selectedItem ] tag ] ;
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    // non dovrebbe succedere mai...
    if ( cvrView == nil ) return ;
    [cvrView setCurUnits: newU ] ;
    // faccio finta che sia cambiata la selezione perche' cosi'
    // sia ggiornano tutte le finestre ausiliarie
    [ [NSNotificationCenter defaultCenter]
        postNotificationName:@"CoverViewChangedSelection"
        object: cvrView ];
}
```

All'interno della classe `CoverView` ho introdotto una nuova variabile d'istanza, `curUnits`, che conserva appunto l'unità di misura corrente; tale variabile è inizialmente impostata in pixel, ed è caratteristica della `CoverView`: finestre differenti possono avere unità di misura differenti.

Da notare la presenza dell'ultima istruzione, che scatena il meccanismo delle notifiche e l'aggiornamento di tutte le finestre ausiliarie. Ed infatti, ad un cambio di unità di misura, occorre modificare tutte le rappresentazioni delle grandezze presenti. La prima grandezza presente è proprio la spaziatura della griglia, presente all'interno dello stesso pannello.

Ecco quindi che il metodo che effettua l'aggiornamento del contenuto del pannello in seguito ad un qualche cambiamento della finestra in primo piano:

```
- (IBAction)
updateGrid:(id)sender
{
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    float        sliderValue = [ sender floatValue ] ;
    float        tmpVal ;
    [ gridSpacing setFloatValue: sliderValue ];
    [ sliderSpacing setFloatValue: sliderValue ];
    // non dovrebbe succedere mai...
    if ( cvrView == nil ) return ;
    // aggiusto le variabili della griglia
    [ cvrView setGridLineColor: [gridColor color ]];
    [ cvrView setSnapToGrid: ([gridSnap state] == NSOffState ? NO : YES) ]
;
    // converto sempre e comunque in pixel
    tmpVal = convertMeasure( sliderValue, [cvrView curUnits ], UNITS_PIXELS
);
    [ cvrView setGridSpacing: tmpVal ];
    [ cvrView setShowGrid: ([showGrid state] == NSOffState ? NO : YES) ] ;
    // ridisegno la griglia
    [ cvrView setNeedsDisplay: YES ];
}
```

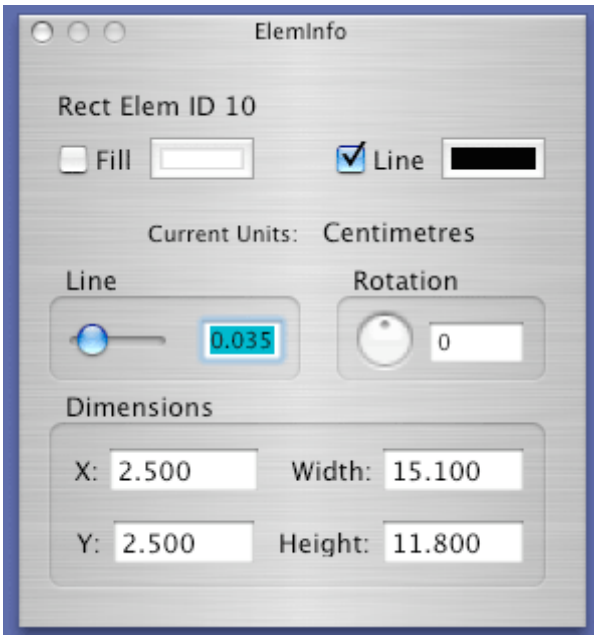
In questo metodo ho fatto collassare sia l'aggiornamento dello slider e del capo testo collegato (prima esisteva un metodo separato). La modifica principale è tuttavia nell'introduzione della funzione `convertMeasure(.)`, che appunto effettua la conversione della grandezza da una unità di misura ad un'altra. Il valore da convertire e le due unità di misura (di partenza e d'arrivo) sono specificate come parametri. La mia

idea infatti è di utilizzare i pixel come misura interna per ogni operazione (dopo tutto, il tracciamento degli elementi avviene in pixel), per cui ogni grandezza è internamente espressa in pixel: la loro modifica e presentazione ha quindi sempre bisogno di una preventiva operazione di conversione.

La funzione di conversione si trova all'interno del file `djZeroUtils.m`, dal momento che sarà utilizzata da vari oggetti presenti all'interno dell'applicazione. La funzione è piuttosto semplice, ma solo perché si trova già tutto pronto:

```
#define         CVT_P2I         72.0F
#define         CVT_C2I         2.54F
#define         CVT_P2C         (CVT_P2I / CVT_C2I)

static    float    cvtFactors[] = {
// cm                in                  pix
    1.0F,             1.0F/CVT_C2I,     CVT_P2C,          // cm
    CVT_C2I,         1.0F,             CVT_P2I,          // in
    1.0F/CVT_P2C,     1.0F/CVT_P2I,     1.0F        } ;     // pix

float
convertMeasure( float meas, int    fromUnit, int toUnit )
{
    int     rr = fromUnit - UNITS_CM ;
    int     cc = toUnit - UNITS_CM ;
    float    cf ;
    if ( rr < 0 || rr > 2 ) return ( 0 );
    if ( cc < 0 || cc > 2 ) return ( 0 );
    cf = cvtFactors[rr *3 + cc];
    return ( cf * meas );
}
```

In primo luogo, mi servono tre fattori di conversioni, tra centimetri, pixel e pollici. Poi costruisco una matrice che contenga tutti i moltiplicatori per passare da una unità all'altra. La funzione ricava l'opportuno indice di riga e di colonna per determinare il moltiplicatore all'interno della matrice; poi moltiplica la grandezza per tale fattore e restituisce il valore: più difficile da spiegare che da fare.



C'è un altro pannello che risente dell'introduzione delle unità di misura, ovvero il pannello delle informazioni relative ad un elemento. Qui occorre convertire di volta in volta lo spessore della linea, il punto di origine e la dimensione dell'elemento. Per ricordare all'utente quali siano le unità di misura corrente, ho modificato la finestra in modo che tale informazione sia ben presente.

Come problema accessorio, ci sono da impostare correttamente i limiti dello slider (in pixel, una linea deve essere compresa tra zero e dieci pixel, ma in centimetri o pollici il limite superiore va modificato di conseguenza). Ecco quindi che il metodo di aggiornamento della finestra `ElemInfoWinCtl` è stato modificato:

```
- (void)
updateWindowInfo
{
    float    tmpVal ;
    int        curUnit ;
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;

    curUnit = [cvrView curUnits ] ;
    [ curUnits setStringValue: stringForUnits(curUnit) ];
    // predispongo gli elementi che corrispondono agli attributi comuni
    ...
    // limite massimo e minimo nelle unita' correnti
    [lineWidthSlider setMinValue: 0 ];
    tmpVal = convertMeasure( 10.0, UNITS_PIXELS, curUnit );
    [lineWidthSlider setMaxValue: tmpVal ];
    // valore corrente dello slider
    tmpVal = convertMeasure( [ theSelectedElem cceLineWidth], UNITS_PIXELS,
curUnit );
    [lineWidthSlider setFloatValue: tmpVal ];
    [lineWidthTextField setFloatValue: tmpVal ];
    ...
    // prima calcolo il valore secondo le unita', poi lo assegno
     tmpVal = convertMeasure( [theSelectedElem drawPoint].x, UNITS_PIXELS,
curUnit );
     [xTextField setStringValue: [ NSString stringWithFormat: @"%f", tmpVal
] ];
     tmpVal = convertMeasure( [theSelectedElem drawPoint].y, UNITS_PIXELS,
curUnit );
    [yTextField setStringValue: [ NSString stringWithFormat: @"%f",tmpVal ]
];
          tmpVal   =   convertMeasure(   [theSelectedElem   localSize].width,
UNITS_PIXELS, curUnit );
      [widthTextField setStringValue: [ NSString stringWithFormat: @"%f",
tmpVal ] ];
         tmpVal   =   convertMeasure(   [theSelectedElem   localSize].height   ,
UNITS_PIXELS, curUnit );
      [heightTextField setStringValue: [ NSString stringWithFormat: @"%f",
tmpVal ] ];
    ...
}
```

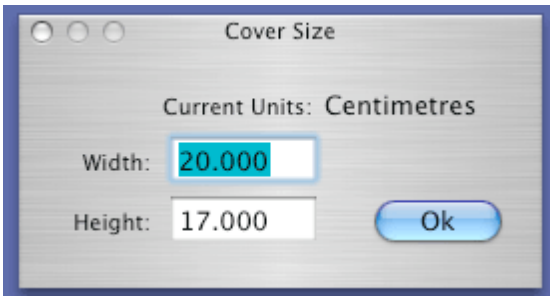
In pratica, prima di presentare ogni grandezza, è eseguita la sua conversione nelle unità correnti della `CoverView`, passando attraverso la funzione `convertMeasure(.)`. In più, c'è la visualizzazione di una stringa (prodotta dalla funzione `stringForUnits(.)` nel file `djZeroUtils.m`) che indica l'unità di misura corrente.

Va da sé che anche l'impostazione di nuovi valori agli elementi deve passare attraverso le operazioni di conversione:

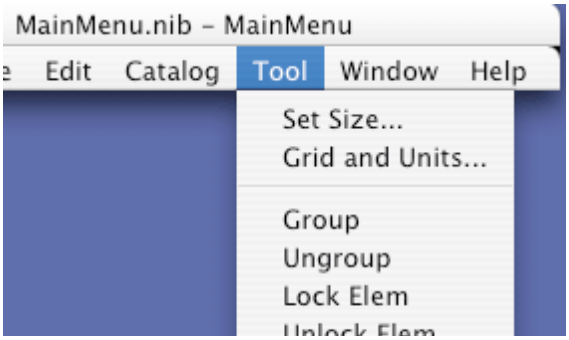
```
- (IBAction)
changedElem:(id)sender
{
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    float        p1, p2 ;
    // a seconda dell'attributo modificato
    switch ( [ sender tag ] ) {
    ...
    case 16 :     // origine dell'elemento
    case 17 :
        if ( [ theSelectedElem cceIsLocked] ) break ;
        // riconverto la dimensione in pixel
         p1 = convertMeasure( [xTextField floatValue ] , [cvrView curUnits
], UNITS_PIXELS ) ;
         p2 = convertMeasure( [yTextField floatValue ] , [cvrView curUnits
], UNITS_PIXELS ) ;
        [ theSelectedElem setDrawPoint: NSMakePoint( p1, p2) ];
        [ theSelectedElem calcLocalTranform ];
        break ;
        ...
    }
    ...
}
```

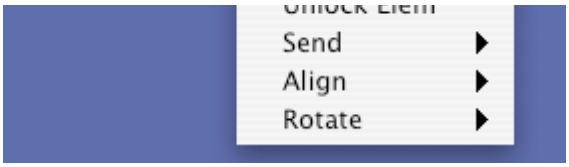
LE DIMENSIONI CONTANO

Fino ad ora, le dimensioni di una copertina sono sempre state le stesse (una ventina di centimetri per lato, più o meno). In realtà si può avere necessità di dimensioni differenti, per cui mi sono posto il problema di cambiare la dimensioni della `CoverView`, la vista che appunto contiene tutti gli elementi di una copertina.



Ho per tanto costruito un nuovo pannello, molto semplice, per impostare le dimensioni della `CoverView` in larghezza ed altezza. Il controllore è la classe `CvrSizeWinCtl` e fa parte della categoria delle finestre accessorie: partecipa quindi al meccanismo delle notifiche centralizzate realizzato da `AppDelegate`; inoltre, presentando grandezze, deve essere realizzato il meccanismo di conversione delle unità.





Ho aggiunto una voce di menu, anzi, ho riorganizzato l'intero menu *Tools* per ricordare anche che il pannello di impostazione della griglia è utile anche per cambiare le unità di misura. Ovviamente, le voci fanno riferimento ad un metodo di `AppDelegate` che va ad aprire l'istanza condivisa (e statica) delle classi relative ai pannelli `GridDlgWinCtl` e `CvrSizeWinCtl`.

Il funzionamento del pannello si basa su una (nuova) variabile d'istanza `curViewSize` (è un `NSSize`) della CoverView, che appunto conserva la dimensioni correnti della view. Per aggiustare la finestra con i valori correnti, alla sua apertura, si usa il metodo seguente:

```
- (void)
updateWindow: (CoverView *) cvrView
{
    float    tmpVal ;
    int        curUnit ;
    if ( cvrView == nil)
    {
        [ self setControlsEnabled: NO ];
        return ;
    }
    // se arrivo qui, c'e' una coverView in fronte a tutti
    // predispongo la finestra in accordo
    // abilito tutti i controlli
    [ self setControlsEnabled: YES ];
    // imposto i vari valori della griglia, utilizzando
    // le unita' correnti
    curUnit = [cvrView curUnits ] ;
    [ showUnits setStringValue: stringForUnits(curUnit) ];
    tmpVal = convertMeasure( [ cvrView curViewSize].width,
                            UNITS_PIXELS, curUnit );
    [ widthText setFloatValue: tmpVal ];
    tmpVal = convertMeasure( [ cvrView curViewSize].height,
                            UNITS_PIXELS, curUnit );
    [ heightText setFloatValue: tmpVal ];
}
```

I due campi che presentano larghezza ed altezza sono riempiti con i valori che si ottengono dalla variabile d'istanza `curViewSize` opportunamente convertita.

Dall'altra parte, quando l'utente imposta nuovi valori per tali grandezze, si procede in senso inverso:

```
- (IBAction)
setUnits:(id)sender
{
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    float    curW = [ widthText floatValue] ;
    float    curH = [ heightText floatValue] ;
    // non dovrebbe succedere mai...
    if ( cvrView == nil ) return ;
    // converto le dimensioni in pixel
```

```
    curW = convertMeasure( curW, [cvrView curUnits ], UNITS_PIXELS ) ;
    curH = convertMeasure( curH, [cvrView curUnits ], UNITS_PIXELS ) ;
    // imposto le dimensioni
    [ cvrView setCurViewSize: NSMakeSize( curW, curH) ];
    // bisogna tenere conto del fattore di zoom corrente
    curW *= [ cvrView curZoomFactor ];
    curH *= [ cvrView curZoomFactor ];
    [ cvrView setFrameSize: NSMakeSize( curW, curH) ];
    [ cvrView setNeedsDisplay: YES ];
    // chiudo questa finestra
    [ [ CvrSizeWinCtl sharedCvrSizeWinCtl ] close ] ;
}
```

Dopo aver impostato i nuovi valori, devo aggiustare il frame della vista. Nel farlo, devo tenere conto del fattore di zoom corrente, che altrimenti succedono dei pasticci. Ho dovuto quindi aggiungere una variabile d'istanza nella `CoverView` che tenesse appunto conto di questo fattore.

TEMPLATE

Fino ad ora, ogni lavoro svolto su di una copertina veniva perduto alla chiusura della finestra: non c'era la possibilità di salvare il lavoro svolto. Con i *template*, introduco la possibilità di salvare l'aspetto di una copertina; non si salva una copertina legata ad una determinato volume, ma un aspetto generale della copertina, che può essere applicato a vari volumi (con i campi di testo automatico che sono aggiornati di conseguenza).

Un template è quindi costituito da una serie di informazioni relative alla `CoverView` (dimensioni, unità di misura, eccetera) e dall'elenco degli elementi che appartengono alla view: la loro natura, la loro disposizione, le loro caratteristiche. Intendo salvare tutte queste informazioni all'interno di un file, e di permettere la costruzione di una `CoverView` a partire dal contenuto di questi file.

In passato, avevo utilizzato il protocollo `NSCoding` per salvare il contenuto di un catalogo, sia direttamente che indirettamente, dal momento che la procedura è gestita in automatico dalla classe `NSDocument`. Qui riprendo appunto il concetto di `NSCoding`, per aggiungere ad ogni elemento i due metodi necessari all'archiviazione ed al ripristino di oggetti su file:

```
- (void)    encodeWithCoder:(NSCoder *)encoder ;
- (id)        initWithCoder:(NSCoder *)decoder ;
```

Ricordo che il primo metodo serve a salvare la struttura dati di un oggetto, mentre il secondo permette di ripristinare tale struttura a partire dai dati salvati.

Ci sono alcune novità rispetto a quanto visto a suo tempo. L'archiviazione effettuata nel capitolo 16 era *seriale*: la struttura di ogni oggetto era convertita in un flusso lineare di dati. Tali dati, una volta scritti in un determinato ordine, richiedevano di essere estratti nel medesimo ordine, pena il fallimento dell'operazione. A partire dalla versione 10.2 del sistema operativo, il meccanismo di salvataggio preferenziale è diventato un altro (la serializzazione è consigliata solo per mantenere compatibilità all'indietro con file e con precedenti installazioni), ovvero tramite *chiave* e *valore*. Ogni grandezza è adesso salvata all'interno di un file non più in rigoroso ordine sequenziale, ma attraverso una

coppia chiave-valore (una sorta di dizionario); ciò permette il ripristino dell'oggetto anche estraendo i dati in ordine differente.

Nel mio caso, il metodo di codifica principale è ovviamente quello relativo a `CCE_BasicForm`:

```
- (void)
encodeWithCoder:(NSCoder *)encoder
{
    if ( [encoder allowsKeyedCoding] )
    {
        // ogni variabile e' codificata con un opportuno identificatore,
        // formato dallo id seguito dal nome
        // alcune variabili si saltano perche' ricostruibili dalle altre
        [ encoder encodeInt32: objID forKey: keyStr( 0, @"objID") ] ;
                [ encoder encodePoint: drawPoint forKey: keyStr(objID,
@"drawPoint") ] ;
         [ encoder encodeFloat: rotAngle forKey: keyStr(objID, @"rotAngle")
] ;
         [ encoder encodeInt: numOfHdl forKey: keyStr(objID, @"numOfHdl") ]
;
        [ encoder encodeSize: localSize forKey: keyStr(objID, @"localSize")
] ;
              [ encoder encodeFloat: cceLineWidth forKey: keyStr(objID,
@"cceLineWidth") ] ;
             [ encoder encodeObject: cceFillColor forKey: keyStr(objID,
@"cceFillColor") ] ;
             [ encoder encodeObject: cceLineColor forKey: keyStr(objID,
@"cceLineColor") ] ;
              [ encoder encodeBool: cceIsFilled forKey: keyStr(objID,
@"cceIsFilled") ] ;
             [ encoder encodeBool: cceIsStroked forKey: keyStr(objID,
@"cceIsStroked") ] ;
              [ encoder encodeBool: cceIsLocked forKey: keyStr(objID,
@"cceIsLocked") ] ;
    }
}
```

Solo le sottoclassi che aggiungono variabili d'istanza hanno bisogno di sovrascrivere tale metodo; ad esempio `CCE_ElemGroup`:

```
- (void)
encodeWithCoder:(NSCoder *)encoder
{
    [ super encodeWithCoder: encoder ] ;
    // devo aggiungere la codifica di tutti gli elementi
    if ( [encoder allowsKeyedCoding] )
    {
        // basta codificare solo lo array
              [ encoder encodeObject: elemArray forKey: keyStr(objID,
@"elemArray") ] ;
    }
}
```

La funzione `keyStr(.)` produce una stringa che identifica in maniera univoca la grandezza salvata (è normalmente composta dallo identificatore dell'elemento seguito dal nome della grandezza).

Per il ripristino dei dati, ci sono i metodi corrispondenti, per `CCE_BasicForm`:

```
- (id)
initWithCoder:(NSCoder *)decoder
{
if ( [decoder allowsKeyedCoding] )
    {
        objID = [decoder decodeInt32ForKey: keyStr( 0, @"objID") ];
                drawPoint  =  [decoder  decodePointForKey:  keyStr(  objID,
@"drawPoint") ];
         rotAngle = [decoder decodeFloatForKey: keyStr( objID, @"rotAngle")
];
          numOfHdl = [decoder decodeIntForKey: keyStr( objID, @"numOfHdl")
];
        localSize = [decoder decodeSizeForKey: keyStr( objID, @"localSize")
];
        [ self calcLocalTranform ];
        [ self setTheDrawPath: [NSBezierPath bezierPathWithRect:
           NSMakeRect(0, 0, localSize.width, localSize.height) ] ];
        [ self buildHdlList ];
              cceLineWidth  =  [decoder  decodeFloatForKey:  keyStr(  objID,
@"cceLineWidth") ];
            [ self setCceFillColor: [ decoder decodeObjectForKey: keyStr(
objID, @"cceFillColor") ] ];
            [ self setCceLineColor: [ decoder decodeObjectForKey: keyStr(
objID, @"cceLineColor") ] ];
               cceIsFilled  =  [decoder  decodeBoolForKey:  keyStr(  objID,
@"cceIsFilled") ];
               cceIsLocked  =  [decoder  decodeBoolForKey:  keyStr(  objID,
@"cceIsLocked") ];
              cceIsStroked  =  [decoder  decodeBoolForKey:  keyStr(  objID,
@"cceIsStroked") ];
        cceIsSelected = FALSE ;
    }
    return ( self );
}
```

Per la classe `CCE_ElemGroup`, invece:

```
- (id)
initWithCoder:(NSCoder *)decoder
{
    [ super initWithCoder: decoder ] ;
    if ( [decoder allowsKeyedCoding] )
    {
          [ self setElemArray: [ decoder decodeObjectForKey: keyStr(objID,
@"elemArray") ] ];
    }
    return ( self );
}
```

Devo inoltre rendere la classe `CoverView` compatibile col protocollo `NSCoding`, scrivendo i due metodi seguenti:

```
- (void)
encodeWithCoder:(NSCoder *)encoder
{
    if ( [encoder allowsKeyedCoding] )
    {
        [ encoder encodeObject: curTemplateName forKey: @"tmpl_Name" ] ;
        [ encoder encodeFloat: gridSpacing forKey: @"tmpl_gridSpacing"] ;
        [ encoder encodeBool: showGrid forKey: @"tmpl_showGrid"] ;
```

```
        [ encoder encodeBool: snapToGrid forKey: @"tmpl_snapToGrid"] ;
        [ encoder encodeObject: gridLineColor forKey: @"tmpl_gridLineColor"
] ;
        [ encoder encodeInt32: nextObjId forKey: @"tmpl_nextObjId"] ;
        [ encoder encodeInt: curUnits forKey: @"tmpl_curUnits"] ;
        [ encoder encodeSize: curViewSize forKey: @"tmpl_curViewSize"] ;
        [ encoder encodeObject: theElements forKey: @"tmpl_Elements"] ;
    }
}

- (id)
initWithCoder:(NSCoder *)decoder
{
    if ( [decoder allowsKeyedCoding] )
    {
        // in realta' l'ordine non ha importanza
                [ self setCurTemplateName: [ decoder decodeObjectForKey:
@"tmpl_Name" ] ];
             gridSpacing = [decoder decodeFloatForKey: @"tmpl_gridSpacing"
];
        showGrid = [decoder decodeBoolForKey: @"tmpl_showGrid" ];
        snapToGrid = [decoder decodeBoolForKey: @"tmpl_snapToGrid" ];
                 [ self setGridLineColor: [ decoder decodeObjectForKey:
@"tmpl_gridLineColor" ] ];
        nextObjId = [decoder decodeInt32ForKey: @"tmpl_nextObjId" ];
        curUnits = [decoder decodeIntForKey: @"tmpl_curUnits" ];
        curViewSize = [decoder decodeSizeForKey: @"tmpl_curViewSize" ];
        [ self setCurCreatingElem: nil ] ;
        [ self setCurSelectRect: nil ] ;
        editingInProgress = nil ;
                  [ self setTheElements: [ decoder decodeObjectForKey:
@"tmpl_Elements" ] ];
    }
    return ( self );
}
```

Per completare la faccenda, ho scritto un metodo che aggiusta l'aspetto di una `CoverView` in base ai dati che riesce ad estrarre da un file (o con valori di default se il file non esiste):

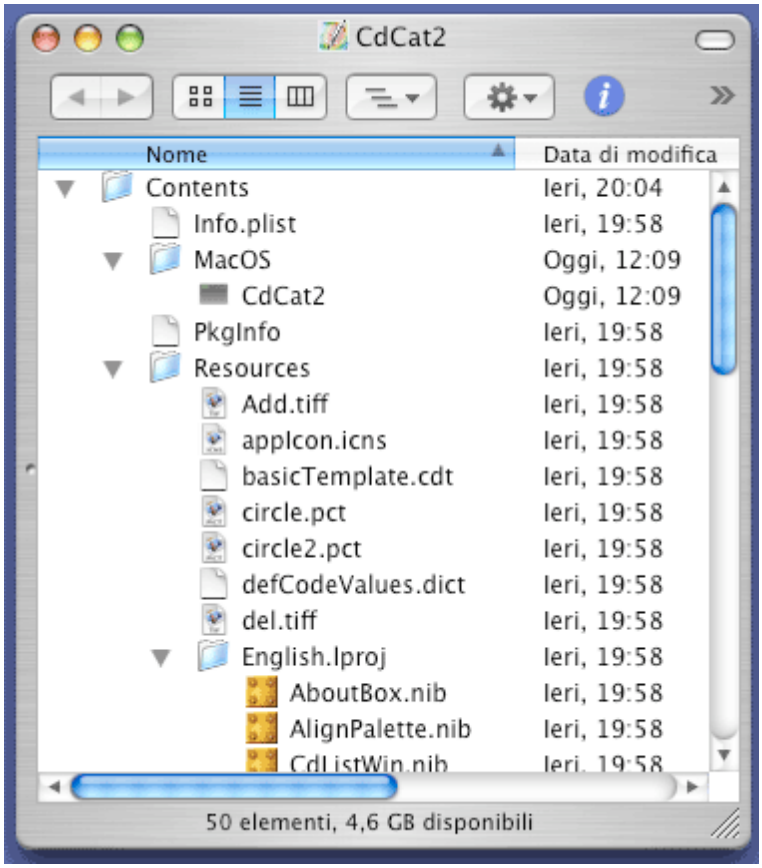
```
- (void)
loadFromTemplate: (NSString *) templatePath
{
    CoverView    * tmpCover ;
    [ winCtl arrangeZoom: 0 ];
    // se il template e' effettivo
    if ( templatePath )
    {
        // estraggo i dati dal template
        tmpCover = [NSKeyedUnarchiver unarchiveObjectWithFile: templatePath
];
        [ theElements setElemArray: [ [tmpCover theElements ] elemArray ] ]
;
        [ [ self theElements] setOwnerView: self ] ;
        // aggiusto tutti gli altri valori della coverView
        [self setCurTemplateName: [tmpCover curTemplateName ] ];
        [self setGridSpacing: [tmpCover gridSpacing ] ];
        ...
    }
    else
    {
        // non ho un template, riempio con valori di default
```

```
        [ theElements setElemArray: [ NSMutableArray array ] ] ;
        [ [ self theElements] setOwnerView: self ] ;
        // aggiusto tutti gli altri valori della coverView
        [self setCurTemplateName: @"---" ];
        [self setGridSpacing: 20 ];
        ...
    }
    // aggiorno il testo automatico
    [ [ theElements elemArray] makeObjectsPerformSelector:
                    @selector(updAutoText:) withObject: [[ winCtl document]
dataSource] ] ;
    // aggiorno l'intera finestra
    [self setNeedsDisplay: YES ];
}
```

Adesso devo costruire l'interfaccia per scatenare tutto questo meccanismo.

DOVE SONO I TEMPLATE

Mi è venuta l'idea di posizionare i template non tanto sparsi per lo hard disk, ma direttamente all'interno dell'applicazione.



Come dovrebbe essere noto, una applicazione, il file *Nome.app*, in realtà è una directory (*Bundle*, secondo la terminologia Apple). All'interno di essa sono disponibili tutte le risorse necessarie all'applicazione, icone, immagini, suoni, file nib, eccetera, ed il codice dell'applicazione stessa. Mi pare un buon posto dove inserire qualche template standard per il disegno di copertine. Ovviamente l'utente potrà scegliere uno di questi template standard, ma anche costruirsene di nuovi e posizionarli dove più gli piace. Tuttavia, i template standard avranno la caratteristica di comparire in un menu pop up

all'interno della finestra.

Ho quindi riarrangiato la finestra `CoverWinCtl`, spostando in alto la barra per lo zoom (e i soliti infami pulsanti per debug), aggiungendo un menu pop up. In questo menu ho messo alcune voci standard, per caricare e salvare template, ed il richiamo di un template *(empty)*, cioè completamente vuoto. Di seguito inserisco tutte le voci relative ai template presenti all'interno dell'applicazione.

Aggiungo quindi i template all'interno del progetto XCode, in modo che in sede di compilazione e costruzione dell'applicazione, i template stessi siano copiati all'interno dell'applicazione.

In sede di preparazione della finestra, all'interno del metodo `windowDidLoad`, aggiusto il menu pop up:

```
- (void)
windowDidLoad
{
    NSArray                 * lt ;
    int                     i ;
    NSString            * tmpl ;
    ...
    [super windowDidLoad ];
    ...
    [[self window] makeFirstResponder:cvrView];
    // cerco tutti i file template
         lt   =   [   [NSBundle   mainBundle]   pathsForResourcesOfType:@"cdt"
inDirectory: @"/" ];
    // e li inserisco nel popup button
    for ( i = 0 ; i < [ lt count ] ; i++ )
    {
        // metto solo il nome, senza extension
        tmpl = [ [ lt objectAtIndex: i ] lastPathComponent ];
        [ tmplList addItemWithTitle: [ tmpl stringByDeletingPathExtension ]
];
        // aggiusto il meccanismo target/Action
        [ [ tmplList itemAtIndex: ( 5 + i) ] setTarget: self ] ;
                [   [   tmplList   itemAtIndex:   (   5   +   i)   ]   setAction:
@selector(loadTemplateFromFile:) ] ;
    }
    // alla partenza, nulla e' selezionato nel pop up menu
    [ tmplList selectItem: nil ] ;
}
```

I template sono dei file con estensione *.cdt*. Il metodo `pathsForResourcesOfType:`

produce un `NSArray` che contiene l'elenco di tutti i file di data estensione all'interno della directory indicata (nel mio caso, la root dell'applicazione, non quella generale del volume...); le istruzioni successive spazzolano questo vettore ed aggiungono il nome del file (senza estensione, che tanto so qual è) come elemento del menu. Inoltre, impostano queste voci affinché invochino il metodo `loadTemplateFromFile:` quando sono selezionate. L'ultima istruzione del metodo `windowDidLoad` fa in modo che nessuna voce del menu sia selezionata.

La voce *Load Template* è collegata (direttamente all'interno di *Interface Builder*) a questo metodo:

```
- (IBAction)
templateLoad: (id)sender
{
    NSOpenPanel        *oPanel = [ NSOpenPanel openPanel ] ;
    int        risposta ;
    // imposto un po' di caratteristiche del dialogo
    [ oPanel setTitle:@"Load Template" ];
    [ oPanel setPrompt:@"Load Template" ];
    // niente directory, un solo file alla volta
    [ oPanel setCanChooseDirectories:NO ];
    [ oPanel setCanChooseFiles:YES ];
    [ oPanel setAllowsMultipleSelection:NO ];
    [ oPanel setResolvesAliases:YES ];
    // posso selezionare solo i file che hanno estensione cdt
      risposta = [ oPanel runModalForTypes: [ NSArray arrayWithObjects:
@"cdt" ] ];
    // se c'e' una selezione effettiva...
    if ( risposta == NSOKButton )
    {
        // ... recupero il nome del file selezionato
        NSArray *filesToOpen = [ oPanel filenames ];
        NSString *aFile = [ filesToOpen objectAtIndex: 0 ];
        // ed uso il template per costruire la view
        [ cvrView loadFromTemplate: aFile ];
    }
    [ tmplList selectItem: nil ] ;
}
```

Si limita a presentare un dialogo di caricamento di un file (niente selezioni multiple) avente estensione *.cdt*. In caso di individuazione corretta di un file, chiama il metodo `loadFromTemplate:` della `CoverView`.

Speculare il metodo associato alla voce *Save Template*:

```
- (IBAction)
templateSave: (id)sender
{
    NSSavePanel    *sPanel = [ NSSavePanel savePanel ] ;
    // personalizzo il pannello di salvataggio
    [ sPanel setTitle:@"Save template" ];
    // i file salvati avranno l'estensione lscat
    [ sPanel setRequiredFileType: @"cdt" ];
    if ( [ sPanel runModal ] == NSOKButton )
    {
        // recupero il nome del file
        NSString *aFile = [ sPanel filename ] ;
        // archivio l'intera struttura dati
```

```
        [NSKeyedArchiver archiveRootObject: cvrView toFile: aFile ];
    }
    [ tmplList selectItem: nil ] ;
}
```

Qui si apre un pannello standard per il salvataggio di un file; in caso di riuscita, si comincia la procedura di archiviazione a partire dalla `CoverView`. A cascata, il salvataggio interessa tutti gli elementi (perché la `CoverView` salva anche il vettore `theElements`).

Rimane il metodo principlae per il caricamento di un template a partire dalla `CoverWinCtl`:

```
- (IBAction)
loadTemplateFromFile: (id)sender
{
    NSString    * selTmpl ;
    NSString    * itTit ;
    // recupero il nome del template
    itTit = [ sender title ] ;
    // recupero il percorso completo del template
     selTmpl = [ [NSBundle mainBundle] pathForResource: itTit ofType:@"cdt"
];
    // carico il template
    [ cvrView loadFromTemplate: selTmpl ];
}
```

Si tratta di recuperare il nome del template e di produrre il path completo del file in cui il template è contenuto; una volta fatto ciò, il lavoro è proseguito dal già visto `loadFromTemplate:`.

ALTRE MODIFICHE

Nel codice ci sono altre modifiche.

Ho spostato in `djZeroUtils.m` la funzione che produce un `NSDictionary` con lo stile ed il font di default per i campi testo. In questo modo può essere chiamata da vari punti, in particolare dall'interno della classe `CatDataSrc` per produrre testo automatico correttamente formattato anche in assenza di stili predefiniti.

Ho completato la classe `CCE_ElemGroup` sovrascrivendo tutti i metodi per l'impostazione degli attributi dell'elemento, in modo che l'attributo sia propagato ai componenti del gruppo. In questo modo, operando ad esempio sulla finestra `ElemInfoWinCtl` si riesce a modificare in un colpo solo, ad esempio, lo spessore delle linee di tutti gli elementi appartenenti al gruppo.

Ci saranno sicuramente altre modifiche qui e lì, ma le ho dimenticate. E comunque, non saranno così importanti...

Capitolo 057 - Visto, si stampi

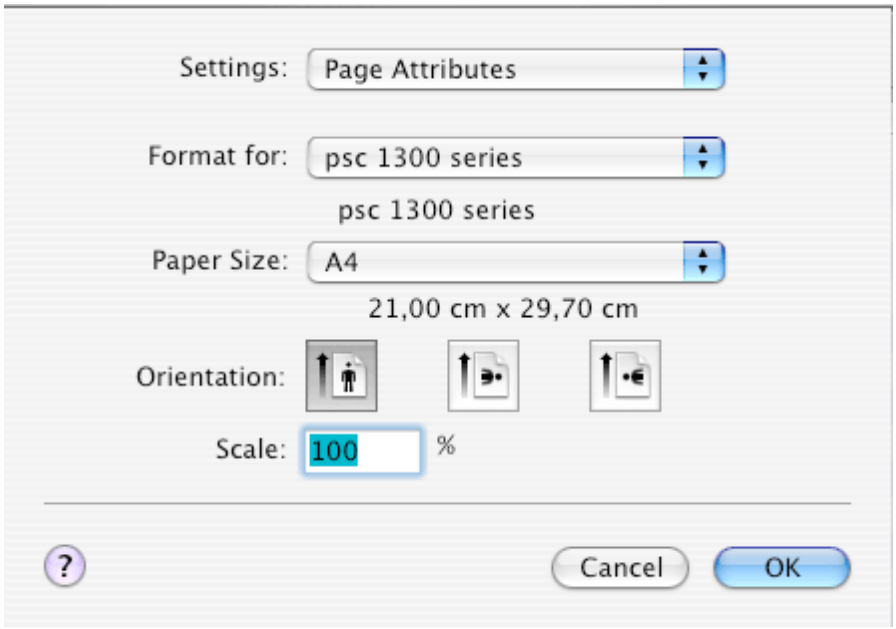
Un capitolo sofferto, non avete idea. Un sacco di problemi, che non ho capito, che ho forse risolto, ma che comunque non mi piacciono. Tutto parte dall'idea di stampare il contenuto delle finestre in modo acconcio. La cosa si rivela piuttosto semplice per la finestra della copertina, mentre una stampa accettabile del catalogo (la lista dei file) è stato un incubo. Tanto tempo perduto, ma, come succede sempre in questi casi, ho imparato molte cose. La cosa più strana, c'è poco codice, ma tante spiegazioni da dare.

Sorgenti: la documentazione Apple, ma potrei essere incocciato in qualche errore.

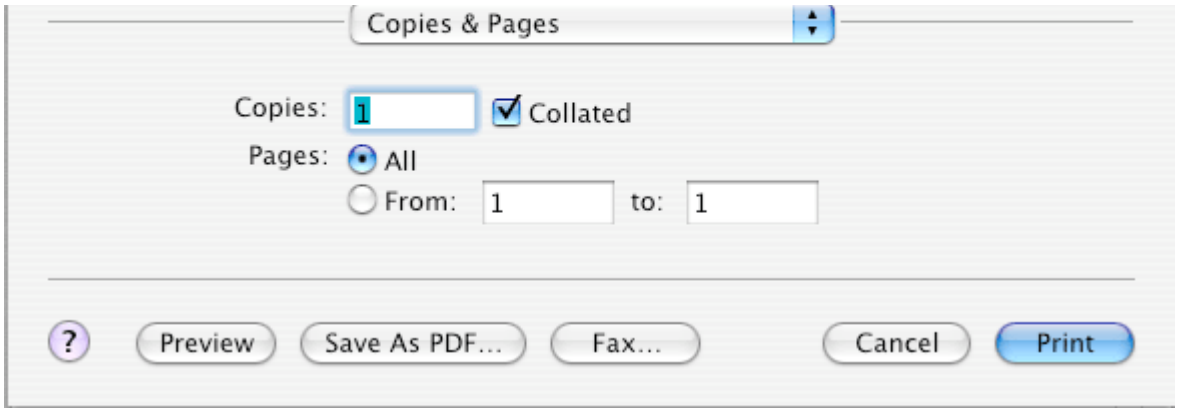
Prima stesura: 9 settembre 2004.

LE CLASSI DELLA STAMPA

Per effettuare le operazioni di stampa, Cocoa mette a disposizione una serie di classe ben organizzate. Il punto di partenza è ciò che si vuole stampare, rappresentato da una classe `NSView` o una sua sottoclasse. Nei casi più semplici, basta inviare alla view il messaggio `print:`, e la cosa finisce lì.



Però, buone operazioni di stampa richiedono due passaggi intermedi: l'impostazione del formato di stampa, e i parametri della stampa vera e propria. L'impostazione del formato è quel simpatico pannello che esce fuori quando si seleziona da menu la voce *Formato di Stampa* o *Page Setup*. Con questo pannello, l'utente determina quale stampante utilizzare, le dimensioni della pagina, l'orientamento della carta, cose del genere.



I parametri della stampa si impostano invece con il pannello che esce quando si seleziona la voce di menu *Stampa* o *Print*; tra i parametri, il numero di copie, caratteri specifici della stampante utilizzata (qualità di stampa, ad esempio), l'eventuale produzione di un file PDF al posto della stampa su carta, eccetera.

Cocoa non distingue la provenienza delle informazioni, se dal *Formato di Stampa* o dal pannello di *Stampa*, ma conserva tutte le informazioni necessarie utilizzando un oggetto della classe `NSPrintInfo`. Di questo oggetto, ne esiste una istanza condivisa con valori di default, ma potrebbe essere necessario produrne una copia per ogni tipologia di stampa desiderata.

Per visualizzare il pannello *Formato di stampa*, Cocoa mette a disposizione la classe `NSPageLayout`, mentre il pannello di stampa è gestito tramite la classe `NSPrintPanel`. Quando si lancia una stampa, l'operazione è gestita da un oggetto `NSPrintOperation`, che interagisce con un oggetto `NSPrinter` contenente una descrizione astratta della stampante.

Per realizzare le operazioni di stampa, occorre rispondere ai messaggi `print:` (se si è una view) o meglio `printDocument:` per la stampa di documenti (dentro la classe `NSDocument`). Il mio caso è il secondo, anzi, è più complicato, perché non solo l'applicazione è in grado di gestire documenti, ma dovrà lanciare due procedure di stampa completamente differenti: una per la stampa della copertina, una per la stampa del catalogo.

FORMATO DI STAMPA

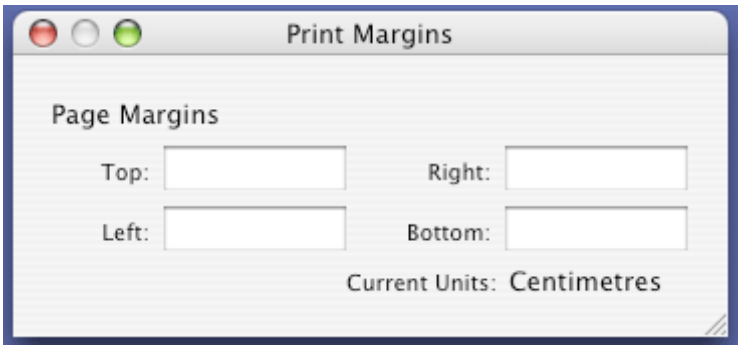
Il pannello *Formato di Stampa* è gestito dalla classe `NSDocument` o `NSApplication` (a seconda se l'applicazione dispone di documenti o no). Normalmente non occorre scrivere codice particolare, in quanto la gestione di default va spesso bene. Ovviamente, voglio provare a fare qualcosa di strano: scopro che la realizzazione standard non permette l'impostazione di margini (a meno di non definire un formato di pagina proprietario). Mi accingo quindi a particolarizzare il pannello con una vista che permetta di impostare i margini di stampa nel caso in cui l'oggetto da stampare sia una copertina (c'è una ragione: la parte frontale di un *jewel box* è un quadrato di dodici centimetri; ripiegato, si ha una copertina di dodici per ventiquattro centimetri, che non si riesce a stampare in un foglio A4 con i margini standard di un pollice ed un quarto).

Il metodo chiamato è `runPageLayout`, e va sovrascritto all'interno della mia classe

NSDocument, ovvero CdCatDoc.

```
- (void)
runPageLayout:(id)sender
{
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    // se ho davanti una CoverView, specializzo la stampa
    if ( cvrView )
    {
        [ cvrView managePageLayout ] ;
        return ;
    }
    // per il momento, gli altri casi usano la tecnica standard
    [ super runPageLayout: sender ] ;
}
```

Ricordo che due sono le possibili finestre: la copertina ed il catalogo. Quindi, per prima cosa, mi chiedo quale sia la finestra di fronte a tutte le altre; se si tratta di una CoverView, la faccio lavorare; altrimenti, utilizzo il meccanismo standard.



Bisogna quindi andare all'interno della classe CoverView e scrivere il metodo da me invocato. Tuttavia, prima c'è da risolvere un problema. La documentazione dice: potete aggiungere al pannello *Formato di Stampa* un vostro pannello, purché si tratti di una NSView, da aggiungere col metodo setAccessoryView. A lungo mi sono domandato come costruire questa view (l'opzione di definire tutti gli elementi tramite istruzioni Cocoa mi sembrava veramente macchinosa), quando ho poi realizzato che potevo fare tutto da *Interface Builder*. Qui ho costruito una finestra minimale, con quattro campi dove modificare i margini di stampa, e la scritta informativa sulle unità di misura in uso. Il tutto è completato dalal classe controllore PrtMrgWinCtl. Ora però, una finestra (NSWindow) non è una NSView (che invece mi serve); ma il trucco sta nell'utilizzare la view interna della finestra. Ecco il codice:

```
- (void)
managePageLayout
{
    NSPageLayout    * pageLayout = [NSPageLayout pageLayout];
    PrtMrgWinCtl    * prtWinCtl ;
    // carico la vista accessoria con l'impostazione dei margini
        prtWinCtl  =  [  [  PrtMrgWinCtl  alloc]  initWithWindowNibName:
@"printMargin" ];
    [ prtWinCtl setRefCvrView: self ] ;
    [ pageLayout setAccessoryView: [ [ prtWinCtl window ] contentView ] ];
    // recupero i margini dall'istanza corrente e li assegno alla vista
    [ prtWinCtl setUpMargins: NSMakeRect(
        [ pInfo topMargin ], [ pInfo leftMargin ],
```

```
        [ pInfo rightMargin ], [ pInfo bottomMargin ]) ];
    // faccio girare l'ambaradan
    [ pageLayout beginSheetWithPrintInfo: pInfo
        modalForWindow: [ winCtl window ]
        delegate: self
        didEndSelector: @selector(pageLayoutDidEnd:returnCode:contextInfo:)
        contextInfo: nil ] ;
       // eventuali modifiche ai margini sono gestiti direttamente da
PrtMrgWinCtl
}
```

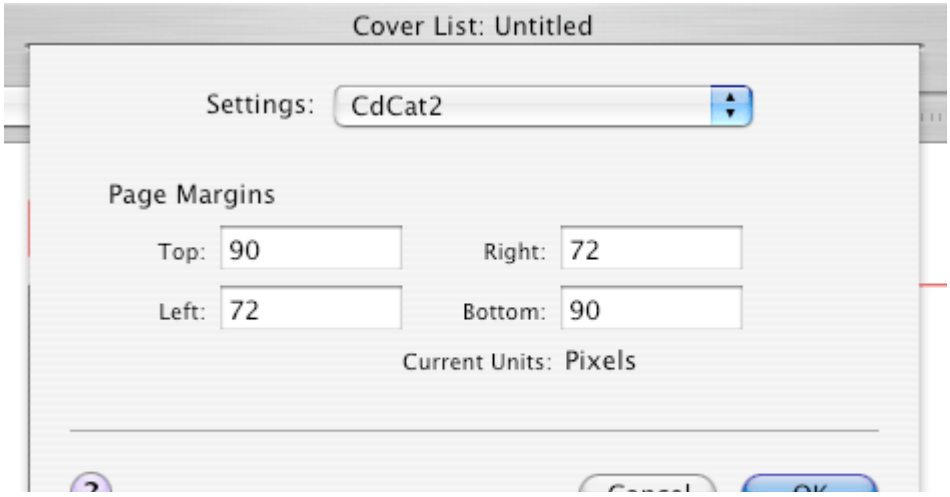
Recupero inizialmente l'istanza condivisa di `NSPageLayout`; carico poi la finestra che ho costruito in *Interface Builder*, ed assegno a `NSPageLayout` la `contentView` della finestra stessa. Poi utilizzo la classe `PrtMrgWinCtl` per impostare il contenuto dai quattro campi con i valori di default. Questi li estraggo dalla nuova variabile d'istanza `pInfo`, che mi sono premunito di inizializzare (all'interno del metodo `initWithFrame:` della `CoverView`) con i valori forniti dall'istanza condivisa di `NSPrintInfo`, utilizzando la semplice istruzione seguente:

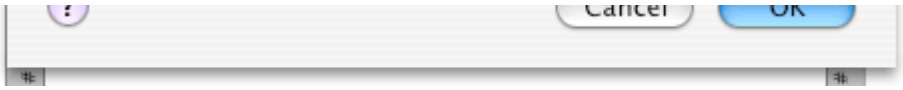
```
[ self setPInfo: [[NSPrintInfo sharedPrintInfo] copy] ];
```

Il metodo setUpMargin è di una facilità estrema, visto come sono andate le cose il capitolo precedente:

```
- (void)
setUpMargins: (NSRect) durRec
{
    int        cu = [refCvrView curUnits ] ;
    float    tmpVal ;
    [ curUnits setStringValue: stringForUnits(cu) ];
    tmpVal = convertMeasure( durRec.origin.x, UNITS_PIXELS, cu );
    [ topMargin setFloatValue: tmpVal ];
    tmpVal = convertMeasure( durRec.origin.y, UNITS_PIXELS, cu );
    [ leftMargin setFloatValue: tmpVal ];
    tmpVal = convertMeasure( durRec.size.width, UNITS_PIXELS, cu );
    [ rightMargin setFloatValue: tmpVal ];
    tmpVal = convertMeasure( durRec.size.height, UNITS_PIXELS, cu );
    [ bottomMargin setFloatValue: tmpVal ];
}
```

Qui `topMargin`, `leftMargin`, eccetera sono gli outlet verso i campi di testo.





Tornando al metodo `managePageLayout`, questo è completato da una istruzione che fa partire la visualizzazione del pannello *Formato di Stampa*, comprensivo della mia nuova vista.

Cosa succede se l'utente cambia i margini? Semplicemente, è chiamato il metodo `updateMargins` della classe `PrtMrgWinCtl`, che mi ero premunito di stabilire come action in risposta ad ogni operazione sui campi di testo.

```
- (IBAction)
updateMargins:(id)sender
{
    // recupero la informazioni di stampa
    int        cu = [refCvrView curUnits ] ;
    NSPrintInfo    * pi = [ refCvrView pInfo ] ;
    float    x1, x2, x3, x4 ;
    x1 = convertMeasure( [ topMargin floatValue ], cu, UNITS_PIXELS) ;
    x2 = convertMeasure( [ leftMargin floatValue ], cu, UNITS_PIXELS) ;
    x3 = convertMeasure( [ rightMargin floatValue ], cu, UNITS_PIXELS) ;
    x4 = convertMeasure( [ bottomMargin floatValue ], cu, UNITS_PIXELS) ;
    [ pi setTopMargin: x1 ] ;
    [ pi setLeftMargin: x1 ] ;
    [ pi setRightMargin: x1 ] ;
    [ pi setBottomMargin: x1 ] ;
}
```

Il metodo non fa altro che recuperare i valori ed impostarli all'interno dell'istanza `NSPrintInfo` propria della `CoverView`.

Per vedere come utilizzare questi valori, ho ad esempio verificato, prima della stamap effettiva, che la `CoverView` sia contenuta per interno all'interno della pagina. Infatti, all'interno di `managePageLayout`, l'ultima istruzione che fa partire la visualizzazione del pannello, stabilisce anche che debba essere chiamato il metodo `pageLayoutDidEnd:returnCode:contextInfo:` al termine delle operazioni; normalmente, questa oeprazione non è necessaria, ma io ho voluto aggiungere appunto un controllo:

```
- (void)
pageLayoutDidEnd: (NSPageLayout *)pageLayout
    returnCode:(int)returnCode
    contextInfo:(void *)contextInfo
{
    [ self checkPageDims ];
}
```

Il metodo che verifica la congruenza delle dimensioni è piuttosto lungo ma tutto sommato semplice. Le istruzioni fondamentali per il controllo sono le prime, tutto il resto è solo amichevolezza verso l'utente.

```
- (void)
checkPageDims
{
```

```
    float        printableWidth, printableHeight, scaleFactor, tmp ;
    // tengo conto del fattore di scala
     scaleFactor = [[[pInfo dictionary] objectForKey:NSPrintScalingFactor]
floatValue];
    tmp = curViewSize.width * scaleFactor ;
    printableWidth = [pInfo paperSize].width - tmp
                    - ([pInfo leftMargin] + [pInfo rightMargin]);
    tmp = curViewSize.height * scaleFactor ;
    printableHeight = [pInfo paperSize].height - tmp
                    - ([pInfo topMargin]+[pInfo bottomMargin]);
    // potremmo avere delle dimensioni troppo piccole
    if ( printableWidth < 0 || printableHeight < 0 )
    {
        // costruisco un alert
        NSAlert *alert = [[NSAlert alloc] init];
        NSString    * messaggio = [ NSString string ];
        // e' un warning, che tanto si puo' andare avanti
        [alert setAlertStyle:NSWarningAlertStyle];
        // c'e' un solo pulsante sdi ok
        [alert addButtonWithTitle:@"OK"];
        // dico che la copertina non ci sta nella pagina
        [alert setMessageText:@"The cover will not fit into the page"];
        // e dico anche per quanto
        if ( printableWidth < 0 )
        {
            float    tmpVal1, tmpVal2 ;
                 tmpVal1  =  convertMeasure(  -printableWidth,  UNITS_PIXELS,
curUnits );
                tmpVal2  =  convertMeasure(  curViewSize.width,  UNITS_PIXELS,
curUnits );
            messaggio = [ messaggio stringByAppendingFormat:
                @"Required width: %7.3f (%@), missing %7.3f\n",
                tmpVal2, stringForUnits(curUnits), tmpVal1 ] ;
        }
        if ( printableHeight < 0 )
        {
            float    tmpVal1, tmpVal2 ;
                 tmpVal1  =  convertMeasure(  -printableHeight,  UNITS_PIXELS,
curUnits );
                tmpVal2  =  convertMeasure(  curViewSize.height,  UNITS_PIXELS,
curUnits );
            messaggio = [ messaggio stringByAppendingFormat:
                @"Required height: %7.3f (%@), missing %7.3f",
                tmpVal2, stringForUnits(curUnits), tmpVal1 ] ;
        }
        [ alert setInformativeText: messaggio ];
        [ alert runModal ];
    }
}
```

Dalla variabile `pInfo` ricavo le dimensioni della pagina, i margini, ed il fattore di

scala; dalla `CoverView` ricavo invece le sue dimensioni, che correggo tenedo conto del fattore di scala (quello che si può impostare in sede di *Formato di Stampa*, non si tratta del fattore di zoom della finestra). Se le dimensioni della copertina superano quelle disponibili per la stampa (le dimensioni della pagina cui sono sottratti i margini), allora mostro un messaggio di avvertimento (un `NSAlert`), opportunamente costruito: un solo tasto di Ok, il messaggio che dice che la copertina non rientra nella pagina, e perfino un testo che spiega quanto spazio manca per una corretta stampa.

Tutto ciò conclude l'argomento sul *Formato di Stampa*. Adesso provo l'effettiva stampa.

STAMPA DI UNA COPERTINA

In una applicazione basata su documenti, il menu *Stampa* normalmente richiede l'esecuzione del metodo `printDocument` alla classe `NSDocument` (attenzione: ho dovuto cambiare la voce di menu *Print* del file `MainMenu.nib` perché, all'inizio dei tempi, l'avevo associata al metodo `print:`; per lo stesso motivo, all'interno di `ListWinCtl` rimuovo un metodo che associava lo stesso metodo ad un elemento della toolbar). L'oggetto documento associato (nel mio caso, `CdCatDoc`) riceve il messaggio e chiama a sua volta il metodo `printShowingPrintPanel:`. Ora, l'oggetto documento standard `NSDocument` non conosce la rappresentazione che del documento viene fatta, nel caso particolare; in altre parole, non conosce quale sia la View da stampare, e non può quindi fornire un metodo di default per la stampa. In definitiva, devo sovrascrivere il metodo `printShowingPrintPanel` per far funzionare le cose.

```
- (void)
printShowingPrintPanel:(BOOL)showPanels
{
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    NSOutlineView * outView = [AppDelegate getTheOutlineView ] ;
    if ( cvrView )
    {
        [ cvrView managePrintWithPanels: showPanels ] ;
        return ;
    }
    if ( outView )
    {
        ...
    }
}
```

Per quando riguarda la stampa della copertina, mi sbrigo subito passando il compito, ancora una volta, alla `CoverView`. Il parametro `showPanels` indica se bisogna o meno visualizzare il pannello di stampa.

Tornando quindi all'interno della classe `CoverView`, ecco il metodo principale per la stampa.

```
- (void)
managePrintWithPanels:( BOOL ) showPanels
{
    NSPrintOperation    * po ;
    // verifico se ci sta nella pagina
```

```
    [ self checkPageDims ];
    // comincio le operazioni di stampa
     po = [ NSPrintOperation printOperationWithView: self printInfo: pInfo
];
    [ po setShowPanels: showPanels ];
    [ po runOperationModalForWindow: [ winCtl window]
        delegate: nil didRunSelector:nil contextInfo: nil ];
}
```

La prima istruzione verifica (sono paranoico, lo so) che la copertina stia all'interno della pagina; quindi, costruisco un oggetto della classe `NSPrintOperation`, destinato a controllare le operazioni di stampa. Questo oggetto è inizializzato con due parametri: la vista che deve essere stampata (facile, la `CoverView` stessa) e le informazioni di stamap (le ho conservate in `pInfo`). Poi, ignorando bellamente il parametro `showPanels`, mostro sempre e comunque il pannello di stampa, come uno sheet per la finestra che contiene la copertina.

Fine. Non c'è altro da fare.

Non è vero, ci sono alcuni piccoli accorgimenti per ottenere un buon risultato; in effetti, la rappresentazione a video e su carta potrebbero essere leggermente differenti in qualche caso. Ad esempio, se la `CoverView` visualizza la griglia, non è bello che la griglia risulti stampata. Ancora, non è bello che ci siano elementi selezionati, che comparirebbero gli inestetici handle anche sulla carta.

Per migliorare dunque la stampa, occorre che i metodi `drawRect` (che sono indifferentemente utilizzati per video e stampa) conoscano il contesto in cui si trovano a disegnare. Basta infatti chiedersi con l'istruzione seguente:

```
[NSGraphicsContext currentContextDrawingToScreen]
```

se il disegno avviene sullo schermo (il risultato dell'istruzione è YES) oppure NO (e per ora l'unica altra possibilità è la stampa).

Quindi modifico il metodo `drawRect:` di `CoverView` nella seguente maniera:

```
- (void)
drawRect:(NSRect)rect
{
    BOOL    scrOrPrt = [NSGraphicsContext currentContextDrawingToScreen] ;
    // sfondo bianco
    if ( scrOrPrt )
    {
        [[ NSColor whiteColor] set ];
        NSRectFill( rect );
    }
    // se c'e' la griglia, la disegno
    if ( scrOrPrt && showGrid)
        [ self drawGrid: rect ];
    // poi dico di disegnarsi a tutti gli elementi
    [ theElements drawElement: rect ];
    // e poi all'elemento in corso
    if ( scrOrPrt && curCreatingElem)
    {
        [ curCreatingElem drawElement: rect ] ;
    }
```

```
    if ( scrOrPrt && curSelectRect)
    {
        NSBezierPath    * curPath = [ curSelectRect theDrawPath ];
        float            strokePattern[2] ={ 5, 5 } ;
        [ curPath setLineDash: strokePattern count: 2 phase: 0.0];
        [ curSelectRect setTheDrawPath: curPath ];
        [ curSelectRect drawElement: rect ] ;
    }
}
```

Con la variabile `srcOrPrt` evito di stampare lo sfondo bianco (che sullo schermo serve a coprire eventuali rimasugli di pixel), la griglia, ed evito anche di disegnare (ma non dovrebbe mai succedere) i rettangoli di selezione corrente e l'elemento in corso di costruzione.

All'interno delle classi `CCE_BasicForm` utilizzo una forma diversa della stessa istruzione per condizionare il disegno degli handle, ma il succo della faccenda non cambia.

```
- (void)
drawElement: (NSRect) inRect
{
    // per salvare il contesto
    NSGraphicsContext *context = [NSGraphicsContext currentContext];
    [context saveGraphicsState];
    [ localTF concat ];
    [ self specificDrawing: inRect ];
    if ( [context isDrawingToScreen ] )
        [ self drawHandles ] ;
    // ripristino del contesto
    [context restoreGraphicsState];
}
```

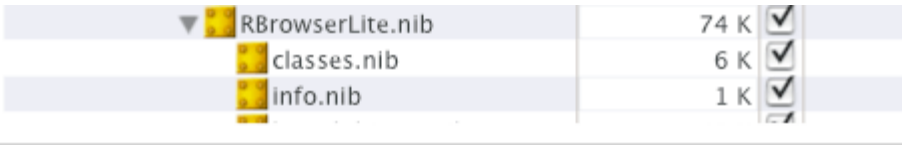
Con queste poche istruzioni, ho prodotto una buona stampa della copertina. Ci sono riuscito con pochi sforzi, e pensavo che altrettanti pochi sforzi servissero per stampare il catalogo. Raramente mi sono sbagliato così tanto.

STAMPA DEL CATALOGO

Se ci si volesse limitare ad una stampa di base, non ci sono problemi. Già ho stabilito che non mi interessa manipolare ulteriormente il pannello *Formato di Stampa*, che lascio gestire in maniera standard dal metodo `runPageLayout`. Potrei cavarmela facilmente facendo qualcosa di simile al metodo `managePrintWithPanels`, dove però si utilizza la `NSOutlineView` come view da stampare. In effetti, in questo modo, si ottengono dei risultati appena sufficienti.

```
- (void)
printShowingPrintPanel:(BOOL)showPanels
{
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    NSOutlineView * outView = [AppDelegate getTheOutlineView ] ;
    if ( cvrView )
    {
        ...
    }
```

```
    // se sto stampando il catalogo, faccio qualche correzione
    // alle impostazioni di stampa
    if ( outView )
    {
        NSPrintOperation    * po ;
        // comincio le operazioni di stampa
         po = [ NSPrintOperation printOperationWithView: outView printInfo:
[ self printInfo ] ];
        [ po setShowPanels: showPanels ];
        [ po runOperationModalForWindow: [ outView window]
            delegate: nil didRunSelector:nil contextInfo: nil ];
    }
}
```



Ci sono diversi punti sui quali lavorare: il primo è di non passare da una pagina alla successiva nel bel mezzo di una riga, ma di fare in modo che si passi alla pagina successiva solo in corrispondenza di righe complete. Un secondo punto è la necessità di trasformare la outlineView in maniera acconcia (è molto brutto stampare una riga a sfondo bianco ed una azzurra). Il terzo punto è la possibilità di inserire intestazioni e piè di pagina, se non altro per indicare il numero di pagina corrente. Bene, si tratterà di sovrascrivere qualche metodo.

Sono pochi, e a prima vista molto chiari:

```
- (BOOL)     knowsPageRange:(NSRangePointer)range ;
- (NSRect)     rectForPage:(int)page ;
- (void)     drawPageBorderWithSize:(NSSize)borderSize;
```

I primi due sono necessari, mentre il terzo è opzionale. Il primo metodo è chiamato una sola volta, all'inizio dei tempi; la realizzazione di default restituisce NO per lasciare fare tutto a Cocoa come preferisce. Per sovrascriverlo occorre restituire YES ed impostare il numero delle pagine che saranno stampate. Poi, per ogni pagina, è chiamato il secondo metodo; gli viene fornito il numero di pagina, e deve restituire un rettangolo; questo rettangolo è poi utilizzato come argomento del metodo `drawRect:` della vista da stampare. Se la vista è ad esempio alta 1000 pixel e la pagina ha una altezza di 400 pixel, im emtodo deve produrre in successione tre rettangoli:

```
Rettangolo 1: x: 0, y: 0, w: <larghezza>, h: 400
Rettangolo 2: x: 0, y: 400, w: <larghezza>, h: 400
Rettangolo 3: x: 0, y: 800, w: <larghezza>, h: 200
```

Infine, il terzo metodo è chiamato poco prima di disegnare una pagina, e permette di inserire elementi grafici accessori, come appunto i numeri di pagina, intestazioni e piè di pagina.

Il problema fondamentale di questo meccanismo è che questi tre metodi sono propri di elementi della classe `NSView` o discendenti; quindi, `NSOutlineView` ha una sua realizzazione standard, che dovrei sovrascrivere in qualche modo. Qui sono cominciati i problemi; ho penato parecchio prima di arrivare ad una soluzione vedibile (o meglio,

stampabile), ricorrendo a diverse strategie e a diversi meccanismi, che mi hanno portato via un sacco di tempo. Alla fine, tra tutti i metodi che ho provato con scarsi risultati, sono pervenuto a quello che descrivo qui di seguito, che mi pare concettualmente il più semplice e flessibile, ma che continua ad avere i suoi (a volte devastanti) problemi.

L'idea è di non stampare direttamente la `NSOutlineView`, ma una vista differente, costruita al volo proprio per la stampa. Questa vista sarà in realtà minimale, solo un pretesto per avere una classe del tipo `NSView` per la quale sovrascrivere i metodi citati (ed anche qualcun altro).

Chiamo questa nuova classe `PrintView`, e per il momento gli fornisco una sola variabile d'istanza `v2p`, un oggetto `NSOutlineView` appunto da stampare. Di più, per inizializzare la classe richiedo proprio tale oggetto:

```
- (id)
initWithOutline:(NSOutlineView *) outView
{
    v2p = outView ;
    ...
    // metto un frame a caso, sara' aggiustato poi
    self = [super initWithFrame: [ outView frame ] ];
    return self;
}
```

Mi disinteresso anche del `frame` della vista, che tanto sarà aggiustato successivamente.

In base a queste premesse, il metodo `printShowingPrintPanel` di `CdCatDoc` diventa:

```
- (void)
printShowingPrintPanel:(BOOL)showPanels
{
    CoverView * cvrView = [AppDelegate getTheDrawView ] ;
    NSOutlineView * outView = [AppDelegate getTheOutlineView ] ;
    if ( cvrView )
        ...
    // se sto stampando il catalogo, faccio qualche correzione
    // alle impostazioni di stampa
    if ( outView )
    {
        PrintView            * pw ;
        NSPrintOperation    * po ;
        NSPrintInfo            * pi = [ self printInfo ] ;
        // dico che la stampa avviene da in alto a sinistra
        // (il default a quanto pare e' al centro)
        [ pi setHorizontallyCentered: NO ] ;
        [ pi setVerticallyCentered: NO ] ;
        pw = [ [ PrintView alloc] initWithOutline: outView ];
        // comincio le operazioni di stampa
        po = [ NSPrintOperation printOperationWithView: pw printInfo: pi ];
        [ po setShowPanels: showPanels ];
        [ po runOperationModalForWindow: [ outView window]
            delegate: nil didRunSelector: nil contextInfo: nil ];
    }
}
```

Rispetto alla realizzazione precedente, c'è appunto la costruzione di una `PrintView`, e una caratterizzazione delle `printInfo`; a quanto pare le opzioni standard di stampa cercano di centrare la vista stampata sia in orizzontale che in verticale; per quanto mi riguarda, la stampa invece dovrà partire in alto a sinistra.

A questo punto, è tutto compito della classe `PrintView`.

PAGINE E PAGINE

Il primo metodo chiamato è `knowsPageRange`, da sovrascrivere se si vuole un meccanismo di paginazione proprietario. Eccolo:

```
- (BOOL)
knowsPageRange:(NSRangePointer)range
{
    [ self prepareViewForPrinting: YES ] ;
    [ self calcPrintValues ] ;
    range->location = 1;
    range->length = totPageNum ;
    return YES;
}
```

Il grosso del lavoro è svolto da due metodi accessori, che predispongono una serie di variabili d'istanza, tra le quali `totPageNum`, che contiene appunto il numero totale delle pagine da stampare.

Con il metodo `prepareViewForPrinting` predispongo la `NSOutlineView` affinché sia stampata in maniera acconcia; userò lo stesso metodo con diverso parametro per ripristinare lo stato precedente (userò il metodo `endDocument`, chiamato da Cocoa al termine delle operazioni di stampa).

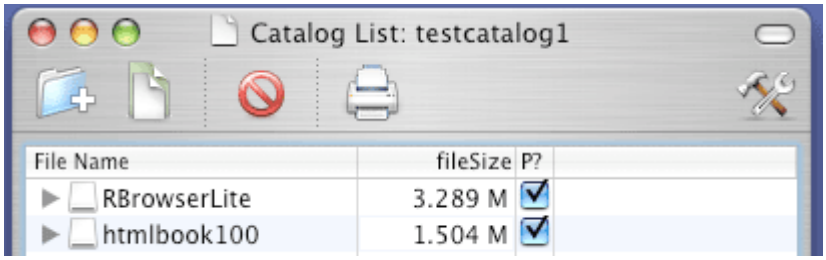
```
- (void )
prepareViewForPrinting: (BOOL) toPrint
{
    if ( toPrint )
    {
        // blocco l'aggiornamento della finestra
        [ [ v2p window] setAutodisplay: NO ] ;
        // adatto la vista ad una situazioen di stampa
        setupColumn( v2p, FALSE, COLID_ADD2PRINT );
        [ v2p setUsesAlternatingRowBackgroundColors: NO ] ;
        [ v2p setGridStyleMask:
                (NSTableViewSolidVerticalGridLineMask |
                NSTableViewSolidHorizontalGridLineMask ) ] ;
    }
    else
    {
        // rispristino la situazione precedente
        setupColumn( v2p, TRUE, COLID_ADD2PRINT );
        [ v2p setUsesAlternatingRowBackgroundColors: YES ] ;
        [ v2p setGridStyleMask: NSTableViewSolidVerticalGridLineMask ] ;
        [ [ v2p window] setAutodisplay: YES ] ;
    }
}
```

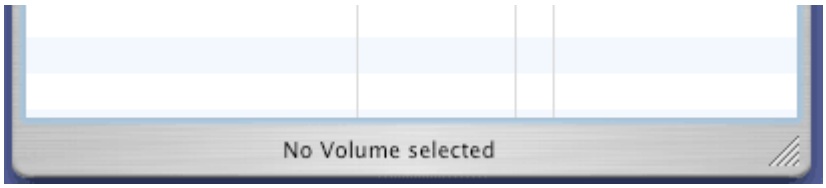
Le modifiche sono sostanzialmente tre (potrei aggiungerne altre, ma mi bastano

queste come prova concettuale della fattibilità): faccio scomparire la colonna `is2Print` (brutta a vedersi in stampa, ed anche poco significativa); le righe hanno sfondo uniforme; in stampa ho la griglia piena (in visualizzazione, avendo righe a colore alternato, le righe della griglia sono solo quelle verticali).

Molto più denso di concetto l'altro metodo, dove sono svolte operazioni fondamentali per l'intero processo di stampa. La spiegazione di questo metodo sarà molto più lunga del metodo stesso.

```
- (void)
calcPrintValues
{
    NSSize            dimPagStampa ;
    // aggiusto le dimensioni della outlineView per
    // comprendere solo fino all'ultima colonna utile
    NSRect             cr = [ v2p rectOfColumn: ([ v2p numberOfColumns] - 1
) ] ;
    float            larghOutline = cr.origin.x + cr.size.width ;
    // Obtain the print info object for the current operation
    NSPrintInfo *pi = [[NSPrintOperation currentOperation] printInfo];
    // Calculate the page height in points
    NSSize paperSize = [pi paperSize];
        float  pageHeight  =  paperSize.height  -  [pi  topMargin]  -  [pi
bottomMargin];
    float pageWidth = paperSize.width - [pi leftMargin] - [pi rightMargin];
    pageMargins = NSMakeRect( [pi leftMargin], [pi topMargin],
                              [pi rightMargin], [pi bottomMargin] );
    // Convert height to the scaled view
    scaleFactor = [[[pi dictionary] objectForKey:NSPrintScalingFactor]
                   floatValue];
    dimPagStampa.height = pageHeight / scaleFactor; ;
    dimPagStampa.width = pageWidth / scaleFactor;

    // scelgo come larghezza la minima tra stamap e outline
    if ( dimPagStampa.width <= larghOutline )
            larghezzaPagina = dimPagStampa.width ;
    else    larghezzaPagina = larghOutline ;
    // calcolo quando spazio occupa in verticale una riga
    altezzaRiga = [ v2p rowHeight ] + [ v2p intercellSpacing ].height ;
    // calcolo quante righe ci stanno in una pagina
    // tengo conto del fattore di scala...
    righePerPagina = dimPagStampa.height / altezzaRiga ;
    // nuovo valore per l'ampiezza pagina
    altezzaPagina = righePerPagina * altezzaRiga ;
    // e per il margine inferiore (un po' piu' grande)
      pageMargins.size.height  +=  (dimPagStampa.height  -  altezzaPagina)  *
scaleFactor ;
    // calcolo quante sono le righe totali da stamapre
    righeDaStampare = [ v2p numberOfRows ] ;
    // basta adesso fare una divisione intera
    totPageNum = righeDaStampare / righePerPagina + 1 ;
}
```





Una delle prime operazioni svolte è il calcolo della variabile `larghOutline`; lo faccio in questo modo strano perché non sempre la dimensione orizzontale della `NSOutlineView` corrisponde all'area effettiva di stampa. Ad esempio, se la dimensione complessiva delle colonne è inferiore alla dimensione della finestra (come nella figura qui vicino), la dimensione della `NSOutlineView` è pari alla dimensione di tutta la finestra, mentre la zona da stampare riguarda solo le colonne effettivamente visibili.

Sono poi calcolate alcune variabili dipendenti dalla pagina scelta per la stampa; in particolare sono conservati in una variabile d'istanza (`NSRect`, ma solo perché è comoda per conservare quattro valori) i margini della pagina (che qui sono sempre quelli di default: un pollice a destra e sinistra, un pollice ed un quarta sopra e sotto), e la dimensione complessiva della pagina in stampa. E qui entra in ballo il fattore di scala (quello impostato con il pannello *Formato di Stampa*). Potrei tediarvi per ore con le mie vicissitudini sull'impatto del fattore di scala nelle mie varie realizzazioni, ma ho pietà dei miei venticinque lettori; qui mi limito a raccontare l'ultimo modello concettuale che mi aiuta nella progettazione.

Quando ho un fattore di scala diverso da 1 (ovvero, diverso dal 100%), significa che devo scalare il disegno in maniera opportuna; oppure, ed è qui l'idea, posso disegnare come se niente fosse, ma ho una pagina che è più grande o più piccola in dipendenza del fattore di scala. Se ad esempio il fattore di scala è 50%, è come se mi trovassi a disegnare su una pagina larga e lunga il doppio. Viceversa, con un fattore di scala del 200%, è come lavorare su un pagina metà larga e lunga della dimensione standard. In base a queste considerazioni, alla variabile d'istanza `dimPagStampa` sono assegnati i valori calcolati direttamente dall'oggetto `printInfo`, opportunamente moltiplicati (divisi, in realtà) per il fattore di scala. L'influenza del fattore di scala sarà d'ora in poi pervasiva.

La variabile `dimPagStampa` descrive l'area di stampa disponibile sulla pagina. Ma in realtà la stampa effettiva occupa una dimensione differente: la `NSOutlineView` potrebbe ad esempio essere più piccola, e non occupare tutta la larghezza. È quindi calcolata la variabile d'istanza `larghezzaPagina`, che appunto contiene quanto spazio in orizzontale è occupato. Se la `NSOutlineView` è molto larga, limito l'area di stampa a quella effettivamente disponibile; se invece è piuttosto stretta, uso la sua larghezza come limite.

Ben più importante è invece il calcolo della variabile `altezzaPagina`. Qui c'è il problema di avere una altezza multipla esatta dell'altezza di una riga, per evitare che nella stampa una riga si trovi a cavallo tra due pagine. Calcolata in `altezzaRiga` l'altezza effettiva della riga (volete che vi racconti quanto ho penato prima di capire l'influenza di `intercellSpacing`?), in `righePerPagina` il numero di righe possibili in una pagina (`righePerPagina` è un numero intero, quindi, la divisione tra l'altezza dell'area di stampa e l'altezza di una riga è sempre troncato all'intero immediatamente più piccolo), si ottiene facilmente il valore `altezzaPagina` moltiplicando i due

precedenti valori. Ed adesso, attenzione al gioco di prestigio (fondamentale in un successivo momento); visto che ho cambiato l'altezza della pagina, il margine inferiore non è più quello giusto, ma lo devo correggere proprio della differenza tra l'altezza dell'area di stampa iniziale e `altezzaPagina`. Questa differenza però fa riferimento a dimensioni già scalate (lo è l'area di stampa), mentre i margini non sono scalati; ecco perché lo moltiplico per il fattore di scala (ed anche questa moltiplicazione è stata un bagno di sangue, prima di capirla).

Coraggio, siamo quasi alla fine; adesso è estremamente facile calcolare il numero delle pagine; trovo quante sono le righe da stampare (tutte quelle della `NSOutlineView`), e le divido per il numero di righe in una pagina; devo aggiungere uno per avere la pagina finale, quella che contiene normalmente un numero di righe inferiore al massimo. Questo è il valore che assegno a `totPageNum`.

IL RETTANGOLO DELLA PAGINA

Col metodo precedente ho calcolato il numero di pagine necessarie; adesso, per ogni pagina, devo specificare il rettangolo all'interno del quale avviene la stampa. Il metodo è il citato `rectForPage`; la realizzazione è la seguente, e me ne vergogno un po':

```
- (NSRect)
rectForPage:(int)page
{
    NSRect    lr ;
    // offset corrente
    float    dimpage = (page - 1) * altezzaPagina ;
    // righe ancora da stampare
    int        restoRighe = righeDaStampare - (page-1)* righePerPagina ;
    // la dimensioen della pagina e' per il momento quella standard
    curAltezzaPagina = altezzaPagina ;
    // se pero' ci sono poche righe rimaste...
    if ( restoRighe < righePerPagina )
            curAltezzaPagina = restoRighe * altezzaRiga ;
    // un trucco per tracciare comunque una pagina
    if ( curAltezzaPagina < 1 )    curAltezzaPagina = 1 ;
    // ecco la pagina da stampare
    lr = NSMakeRect( 0, dimpage , larghezzaPagina, curAltezzaPagina ) ;
    // una strana porcheria per correggere le stampe ingrandite
    if ( scaleFactor > 1 )
    {
        lr.size.height *= scaleFactor ;
        lr.size.width *= scaleFactor ;
    }
    curPageNum = page ;
    return ( lr );
}
```

Questo metodo deve produrre un rettangolo, generalmente di dimensioni fisse (nel caso, `larghezzaPagina` e `altezzaPagina`), ma con il punto di origine che si sposta di pagina in pagina per spazzolare la vista in stampa. Si può immaginare la vista in stampa come una grande rettangolo, mentre il rettangolo prodotto da questo metodo parte dal punto in alto a sinistra e si muove verso il basso, spostandosi di volta in volta della dimensione di un'area di stampa, come un oblò che mette in evidenza la porzione della vista principale da stampare.

Ed in effetti le prime istruzioni vanno in questa direzione: nota `altezzaPagina`, e il numero di pagina (che parte da 1), posso facilmente calcolare `dimpage` come l'origine del rettangolo. Ci sono poi alcune istruzioni per aggiustare il valore di `altezzaPagina` per tenere conto dell'ultima pagina (che presente in genere un numero di righe inferiore al massimo consentito) o di una pagina vuota (se restituissi un rettangolo vuoto, la pagina non sarebbe del tutto stampata; decido invece di stampare comunque una pagina vuota imponendo una dimensione minima di un pixel).

Per tentativi ed errori sono arrivato alla prima porcheria del capitolo (ne troverete altre). Per qualche strano motivo, quando il fattore di scala è superiore al 100% (quindi, in caso di ingrandimenti), il rettangolo prodotto (e poi fornito al metodo `drawRect:`) è troppo piccolo, e provoca un errato clipping della pagina stampata. Con la correzione sulle dimensioni del rettangolo, le cose funzionano correttamente. Ma non chiedete perché.

A questo punto, per disegnare una pagina (lascio per il momento perdere intestazione e piè di pagina, li vedrò più tardi), basta scrivere il metodo di disegno della vista:

```
- (void)
drawRect: (NSRect) rect
{
    // c'e' solo la outline da disegnare...
    [ v2p drawRect: rect ] ;
}
```

Estremamente semplice ed ovvio, e sbagliato.

Ancora una volta, ho penato tantissimo per venirne fuori.

Tutto in effetti funziona finché si stampa al 100%. Non appena ho un fattore di scala differente, si nota che in realtà la `NSOutlineView` non viene scalata correttamente. Ho impiegato parecchie ore prima di capire che non è attiva la trasformazione di scala. La cosa triste è che usando la paginazione standard (in pratica, non sovrascrivendo `knowsPageRange` e `rectForPage`), la trasformazione di scala è attiva, mentre, semplicemente sovrascrivendoli, non lo è più. Chissa perché.

Provo a riscrivere il tutto così, realizzando esplicitamente la scalatura:

```
- (void)
drawRect: (NSRect) rect
{
    NSAffineTransform    * tf = [ NSAffineTransform transform ] ;
    [ tf scaleBy: scaleFactor ] ;
    [ tf concat ] ;
    // c'e' solo la outline da disegnare...
    [ v2p drawRect: rect ] ;
}
```

Ottengo risultati bizzarri: porzioni della `NSOutlineView` non sono stampati, o sono stampati in posti completamente sbagliati, come se fossero spostati in altri posti. Ancora una volta , ho penato molto, confrontando le situazioni in paginazione standard e paginazione proprietaria. Sembra che la trasformazione per il cambio del sistema di riferimento tra pagina e area di stampa non funzioni correttamente. Dopo aver a lungo

navigato nella documentazione, ho trovato un metodo che sembra fare al caso mio: il metodo `beginPageInRect:` è chiamato per ogni pagina dopo `rectForPage:` ma prima di `drawRect:`, ed il suo compito principale è l'impostazione del contesto grafico (tra cui, appunto la trasformazione del sistema di riferimento). L'analogo metodo `endPage` (dopo il disegno della pagina) rispristina la situazione iniziale.

Colto dalla disperazione, mi sono accinto a calcolare da zero la trasformazione. Allora, il punto di partenza è il sistema di riferimento (SdR) della pagina. L'origine è in basso a sinistra, con asse delle ascisse verso destra e delle ordinate verso l'alto. Il SdR dell'area di stampa è però differente. La sua origine è in un punto alto a sinistra, distante dal lato sinistro e dall'alto quanto stabilito dai margini. L'asse delle ascisse è verso sinistra, le ordinate verso il basso. Inoltre, bisogna tenere conto dell'eventuale fattore di scala, che cambia le unità di misura. A tutto questo, occorre aggiungere il fatto che siamo all'ennesima pagina. In sintesi, ottengo il metodo successivo:

```
- (void)
beginPageInRect:(NSRect)aRect atPlacement:(NSPoint)location
{
    float                pageOffset ;
    NSAffineTransform    * tf = [ NSAffineTransform transform ] ;
    [ super beginPageInRect: aRect atPlacement: location ] ;
    // per qualche motivo, la trasformazione operata dalla superclasse
    // non va per nulla bene, mi tocca inventarne una da me
    pageOffset = curPageNum * altezzaPagina * scaleFactor ;
    // l'origine e' il punto in basso a sinistra della pagina
    // mi sposto in alto a sinistra, tenendo conto del margine
    [ tf translateXBy: pageMargins.origin.x
                yBy: (pageOffset + pageMargins.size.height) ] ;
    // cambio la scala e rovescio l'asse delle y
    [ tf scaleXBy: scaleFactor yBy: -scaleFactor ] ;
    [ tf set ] ;
}
```

Ci sono alcuni ragionamenti da fare per tenere conto del fatto che le ordinate nei due SdR sono una opposta all'altra, e che quindi il punto in alto a sinistra va calcolato tenendo conto del margine inferiore (quello magicamente corretto in precedenza), al quale si aggiunge lo spazio della pagina (in unità reali, non quelle corrette dal fattore di scala). Mi spiace per voi, dovete ragionarci da soli, con qualche disegnino ed esempio, perché non riesco a spiegarmi meglio.

È importante notare l'istruzione di `set` piuttosto che `concat` (come finora applicato in tutte le trasformazioni affini); il messaggio `set` rimpiazza totalmente la trasformazione, mentre `concat` semplicemente l'aggiungeva a quella già presente.

Va da sé che, dal momento che la trasformazione è eseguita da questo metodo, il metodo `drawView:` mantiene la sua forma più semplice, con l'unica istruzione di disegno della NSOutlineView.

Infine, per permettermi e permettervi di capire meglio quanto sta succedendo, ho scoperto (non è stato banale) un metodo per conoscere la trasformazione corrente, raccolto nella seguente funzione presente in `djZeroUtils.m`:

```
void
```

```
printCTM(NSString* ctx )
{
    CGAffineTransform    xxx ;
          CGContextRef  myContext  =  [[NSGraphicsContext  currentContext]
graphicsPort];
    xxx = CGContextGetCTM( myContext );
    NSLog(@"%@ x11: %8.3f, x12: %8.3f x21: %8.3f x22: %8.3f, tx: %8.3f, ty:
%8.3f",
            ctx, xxx.a, xxx.b, xxx.c, xxx.d, xxx.tx, xxx.ty );
}
```

La funzione (che ha come argomento una stringa solo per bellezza) recupera il contesto grafico corrente, e da questo risale alla trasformazione attivo in quel contesto; ne stampo tutti i valori così da capire cosa sta succedendo.

DECORAZIONI

Concludo il capitolo su come si possono aggiungere ad una pagina intestazioni, piè di pagina ed in generale decorazioni di vario tipo, che si estendano anche su tutto lo spazio disponibile (al di fuori dell'area di stampa). Il metodo da utilizzare si chiama `drawPageBorderWithSize`, che normalmente fa nulla, e che quindi sovrascritto se si vuole qualcosa di carino. Al metodo è fornito un parametri `NSSize` che contiene le dimensioni della pagina.

La mia realizzazione è piuttosto lunga, ma semplice: voglio disegnare come piè di pagina una stringa che riporti il numero di pagina; come intestazione, riportare il nome del catalogo, e poi inserire sopra la stampa della `NSOutlineView` l'intestazione delle varie colonne, in pratica, disegnare la `headerView` della `NSOutlineView`. Infine, voglio contornare `NSOutlineView` e `headerView` con un bel rettangolo, giusto per dimostrare che sono capace di farlo.

Per capire come procede il disegno, ancora una volta si devono capire i SdR coinvolti. All'ingresso della funzione, il SdR attivo è quello della pagina (origine in basso a sinistra, eccetera). Poi si utilizza la coppia di messaggi `lockFocus` e `unlockfocus` per operare il trasferimento del SdR a quello della `PrintView` (in alto a sinistra, con ordinate rovesciate). Giocando sull'impostazione del frame, si può fare il modo che questo secondo SdR comprenda tutta la pagina o solo porzioni di essa.

La prima operazione è di conservare il `frame` corrente. Cosa ci sia dentro, non mi interessa; tuttavia, la documentazione di metterla da parte e di reimpostarla prima di terminare le operazioni. Successivamente, produco un percorso a forma di rettangolo e produco due stringhe; il tutto mi sarà utile poi.

```
- (void)
drawPageBorderWithSize:(NSSize)borderSize
{
    NSRect            circrect ;
    NSString        * str1, * str2 ;
    NSBezierPath    * cnt ;
    float             vertStrOff ;
    float             invSF = 1 / scaleFactor ;
    // qui e' gia' attiva la trasformazione di scala
    // metto da parte il frame corrente della vista
```

```
    NSRect            prevFrame = [ self frame ];
    // per tenere conto del fattore di scala, faccio finta di avere
    // a disposizione una pagina piu' grande...
    float            tm = pageMargins.origin.y * invSF;
    float            lm = pageMargins.origin.x * invSF;
    // una trasformazione per spostare i disegni delle viste
    NSAffineTransform    * tf = [ NSAffineTransform transform ] ;
    float        headHeight = [ [ v2p headerView ] frame ].size.height ;

    // un rettangolo che contiene tutta la pagina
    circrect = NSMakeRect(lm, tm - headHeight,
            larghezzaPagina, curAltezzaPagina + headHeight ) ;
    circrect = NSInsetRect( circrect, -1, -1) ;
    cnt = [NSBezierPath bezierPathWithRect: circrect ];
    // produco la stringa col numero di pagina
    str1 = [ NSString stringWithFormat: @"Page %d of %d",
                curPageNum, totPageNum ] ;
    str2 = [ NSString stringWithString: [[ v2p window] title ]] ;
```

Comicio adesso il primo blocco di disegno. Imposto come frame l'intera dimensione della pagina. Come al solito devo tenere conto del fattore di scala. Il successivo messaggio di `lockFocus` esegue la trasformazione: adesso l'origine del SdR è il vertice in alto a sinistra della pagina. Il rettangolo `circrect` è stato calcolato in base a questo SdR, e circonda l'intera area di stampa; in più; è stata aggiunto in testa, all'interno del margine superiore, uno spazio adatto a contenere la `headerView`. Disegno il rettangolo. Poi operO una trasformazione di scala. Infatti, voglio che le due stringhe compaiano sempre della stessa dimensione, indipendentemente dal fattore di scala.

Segue l'ennesima porcheria. Per qualche motivo che non capisco, a fattori di scala maggiori del 100%, la zona impostata con la prima istruzione di `setFrame` non comprende tutto lo spazio disponibile; mi tocca quindi cambiare il frame, allargandolo. L'oerazione si rende necessaria perché altrimenti la scritta col numero di pagina non compare.

```
    // primo blocco di disegno, imposto come frame tutta la pagina
    // (tengo conto della scala) per disegnare un bel contorno
    [ self setFrame: NSMakeRect( 0, 0, borderSize.width * invSF,
                                    borderSize.height * invSF ) ] ;
    // comincio il disegno del primo blocco
    [ self lockFocus ] ;
    // riempio giusto per vedere
    [[ NSColor blueColor] set ]; NSRectFill( circrect );
    // e traccio giusto per evidenziare
    [ [ NSColor redColor] set ]; [ cnt setLineWidth: 1 ]; [ cnt stroke ];
    // cambio la scala
    [ tf scaleBy: invSF ] ;
    [ tf concat ] ;
    // adesso il frame e' tutta la pagina (non chiedete...)
    [ self setFrame: NSMakeRect( 0, 0, borderSize.width, borderSize.height)
] ;
    // come footer il numero il pagina
    vertStrOff = pageMargins.origin.y + altezzaPagina * scaleFactor + 3 ;
    circrect = NSMakeRect(pageMargins.origin.x, vertStrOff,
                        larghezzaPagina * scaleFactor, 25 ) ;
    [ str1 drawInRect:circrect withAttributes: nil ] ;
    // come header, il nome del catalogo
    vertStrOff = pageMargins.origin.y - headHeight * scaleFactor - 20 ;
    circrect = NSMakeRect(pageMargins.origin.x, vertStrOff,
                        larghezzaPagina * scaleFactor, 25 ) ;
```

```
    [ str2 drawInRect:circrect withAttributes: nil ] ;
    // rispristino la scala per le successive operazioni
    [ tf scaleBy: scaleFactor ] ;
    // fine primo blocco di disegno
    [ self unlockFocus ] ;
```

Se il fattore di scala è molto grande, c'è il pericolo che la scritta in intestazione esca dalla pagina. A poco vale costringerla all'interno: il successivo disegno della `headerView` coprirebbe comunque la stringa.

Ad ogni modo, le istruzioni successive ripristina la trasformazione di scala ed eseguono un `unlockFocus`. L'operazione mi serve per cambiare ancora una volta il frame corrente e procedere al secondo blocco di disegno, consistente nella sola `headerView`. Dopo l'istruzione di `unlockFocus`, il SdR è tornato quello con origine in basso a sinistra. Il nuovo frame è proprio la zona sopra l'area di stampa, con l'esclusione dei margini sinistro e destro. Eseguo quindi uno spostamento del SdR, in modo che la successiva istruzione di `drawRect:` della `headerView` avvenga correttamente.

```
    // secondo blocco, cambio il frame per disegnare l'intestazione
    [ self setFrame: NSMakeRect( lm, borderSize.height * invSF - tm,
                borderSize.width * invSF - 2*lm, tm ) ] ;
    [ self lockFocus ] ;
    // sposto l'origine per disegnare lo header (un po' piu' su)
    [ tf translateXBy: 0 yBy: tm - headHeight ] ;
    [ tf concat ] ;
    // disegno lo header
    [ [ v2p headerView ] drawRect: [ self bounds] ];
    [ self unlockFocus ] ;

    // ripristino il frame iniziale
    [ self setFrame: prevFrame ] ;
    return ;
}
```

CONCLUSIONI PRELIMINARI

È stato un lavoraccio, e non sono soddisfatto del risultato. Ci sono alcuni punti ancora oscuri, ed alcune magie di origine misteriosa. Tuttavia, la stampa adesso funziona, con risultati tutto sommato accettabili. Nel mio pragmatismo (quando va che tanto basta, non toccare che si guasta), non vorrei metterci più mano. Ci sono sicuramente alcuni punti che non ho chiari, altri che non ho capito per nulla, e il tutto mi da la sensazione di essere un lavoro complicato svolto solo perché non ho trovato la soluzione semplice a portata di mano. Se invece a voi appare evidente, fatemi una cortesia: ditemela, ma abbiate tatto.

Capitolo 058 - Mela C Mela V

Benché l'argomento più succoso del capitolo sia la realizzazione delle operazioni di *Copia* ed *Incolla* sugli elementi della copertina, sono presenti anche altri interessanti pezzi di codice: gli elementi di testo sono in grado di ripartire il loro testo su più colonne, e sono corretti una serie di errori. Ma per cominciare si parla di XML.

Sorgente: quando non si sa dove sbattere la testa, si copia da qualcuno (nel caso, dall'esempio *Sketch*).

Prima stesura: 13 settembre 2004

Proprietà XML

Non sapendo come realizzare le operazioni di copia ed incolla sulla finestra che contiene la copertina, sono andato a guardare un esempio, ovvero *Sketch*, fornito di serie con i *Developer's Tool*. Nel farlo, ho notato che gli elementi grafici sono passati nella *clipboard* (o *Archivio Appunti*) in formato XML. è una cosa simpatica ed interessante, ed anche piuttosto facile.

In effetti, non si tratta che di realizzare una versione alternativa dell'accoppiata `encodeWithCoder/initWithCoder`. Pigliamo ad esempio il metodo che produce una *property list* (che è proprio un documento che segue gli standard XML per conservare informazioni) per un elemento generico `CCE_BasicForm`:

```
- (NSMutableDictionary *)
getXMLDescription
{
    // un dizionario per contenere le varie proprieta' XML
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];
    // nome della classe
    [dict setObject: NSStringFromClass([self class])
            forKey: kStrElem_ClassName ];
    // tutte le altre variabili d'istanza fondamentali
    [dict setObject:[NSString stringWithFormat:@"%d", objID]
            forKey: kStrElem_objID ];
    [dict setObject:NSStringFromPoint(drawPoint)
            forKey: kStrElem_drawPoint ];
    [dict setObject:[NSString stringWithFormat:@"%f", rotAngle]
            forKey: kStrElem_rotAngle ];
    [dict setObject:[NSString stringWithFormat:@"%d", numOfHdl]
            forKey: kStrElem_numOfHdl ];
    [dict setObject:NSStringFromSize(localSize)
            forKey: kStrElem_localSize ];
    [dict setObject:[NSString stringWithFormat:@"%f", cceLineWidth]
            forKey: kStrElem_cceLineWidth ];
    [dict setObject:[NSArchiver archivedDataWithRootObject: cceFillColor ]
            forKey: kStrElem_cceFillColor ];
    [dict setObject:[NSArchiver archivedDataWithRootObject: cceLineColor ]
            forKey: kStrElem_cceLineColor ];
    [dict setObject:(cceIsFilled ? @"YES" : @"NO")
            forKey: kStrElem_cceIsFilled ];
    [dict setObject:(cceIsStroked ? @"YES" : @"NO")
            forKey: kStrElem_cceIsStroked ];
```

```
    [dict setObject:(cceIsLocked ? @"YES" : @"NO")
            forKey: kStrElem_cceIsLocked ];
    return dict;
}
```

Costruisco un dizionario, all'interno del quale inserisco, con opportune chiavi, le varie variabili d'istanza in un formato opportuno. Numeri, rettangoli, punti, eccetera, sotto forma di stringhe, mentre oggetti più complicati (come `NSColor`) in un formato d'archivio. Da notare che all'inizio è inserito anche il nome della classe dell'elemento. In effetti questo sarà un punto chiave per la procedura opposta, che costruisce un elemento a partire da una property list. La procedura è divisa in due metodi: il primo metodo serve proprio a costruire un elemento:

```
+ (id)
buildElemFromXMLDesc:(NSDictionary *)dict
{
    id elem = nil;
    // determino il tipo della classe
    Class theClass = NSClassFromString(
            [dict objectForKey: kStrElem_ClassName ]);
    // se ce l'ho, vado a costruire l'elemento
    if (theClass)
    {
        // costruisco l'elemento
        elem = [ [ [ theClass alloc ] init] autorelease];
        // e poi finisco di caricarlo con i suoi dati
        if (elem)
        {
            [elem loadElemXMLDesc:dict];
        }
    }
    return elem;
}
```

Si tratta di un metodo di classe (che un + davanti al nome del metodo); estrae dal dizionario il nome della classe, ed in base a tale nome costruisce un elemento di classe corretta. Passa poi il controllo ad un altro metodo per terminare le operazioni di riempimento delle variabili d'istanza. Nella realizzazione base, il metodo è il seguente:

```
- (void)
loadElemXMLDesc:(NSDictionary *)dict
{
    // estraggo ogni singola variabile e l'assegno
    objID = [ [ dict objectForKey: kStrElem_objID ] intValue ] ;
    drawPoint = NSPointFromString( [dict objectForKey: kStrElem_drawPoint ]
) ;
    rotAngle = [ [ dict objectForKey: kStrElem_rotAngle ] floatValue ] ;
    numOfHdl = [ [ dict objectForKey: kStrElem_numOfHdl ] intValue ] ;
    localSize = NSSizeFromString( [ dict objectForKey: kStrElem_localSize ]
) ;
    [ self calcLocalTranform ];
    [ self setTheDrawPath: [ NSBezierPath bezierPathWithRect:
            NSMakeRect(0, 0, localSize.width, localSize.height) ] ];
    [ self buildHdlList ];
    cceLineWidth = [ [dict objectForKey: kStrElem_cceLineWidth ] floatValue
] ;
    [ self setCceFillColor: [ NSUnarchiver unarchiveObjectWithData:
                        [dict objectForKey: kStrElem_cceFillColor ] ] ];
    [ self setCceLineColor: [ NSUnarchiver unarchiveObjectWithData:
```

```
                        [dict objectForKey: kStrElem_cceLineColor ] ] ];
    cceIsFilled = [ [dict objectForKey: kStrElem_cceIsFilled ]
                            isEqualToString:@"YES"];
    cceIsLocked = [ [dict objectForKey: kStrElem_cceIsLocked ]
                            isEqualToString:@"YES"];
    cceIsStroked = [ [dict objectForKey: kStrElem_cceIsStroked ]
                            isEqualToString:@"YES"];
    cceIsSelected = FALSE ;
}
```

È in effetti concettualmente molto simile ad `initWithCoder:`: anche qui sono inizializzate tutte le variabili d'istanza estraendone i valori dal dizionario fornito come parametro.

Come è costume, alcune sottoclassi di `CCE_BasicForm` devono riscrivere i due metodi `getXMLDescription` e `loadElemXMLDesc` nel caso in cui possiedano variabili d'istanza aggiuntive (`CCE_Text`, `CCE_Image` e CCE_ElemGroup). Non si deve riscrivere `buildElemFromXMLDesc`, che è unico per tutta la gerarchia degli elementi.

Mi interessa in particolare evidenziare i metodi per la classe `CCE_ElemGroup`:

```
- (NSMutableDictionary *)
getXMLDescription
{
    // la descrizione dell'elemento stesso
    NSMutableDictionary * dict = [super getXMLDescription];
    // ed un array con la descrizione degli elementi del gruppo
    NSMutableArray         * newArr = [ NSMutableArray arrayWithCapacity: 1
] ;
    int                    i , numElem ;
    // ciclo sugli elementi del gruppo
    numElem = [ elemArray count ];
    for ( i = 0 ; i < numElem ; i ++ )
    {
        CCE_BasicForm        * elem ;
        NSMutableDictionary * dictx ;
        // piglio l'elemento
        elem = [ elemArray objectAtIndex: i ];
        // e costruisco un dizionario con le sue proprieta'
        dictx = [ elem getXMLDescription ];
        // che aggiungo allo array
        [ newArr addObject: dictx ];
    }
    // al dizionario aggiungo appunto lo array
    [dict setObject: newArr forKey: kStrElem_elemArray ];
    return ( dict );
}
```

Per la costruzione della property list, oltre alle variabili standard, bisogna aggiungere un vettore, che contiene tutte le descrizioni degli elementi del gruppo.

Speculare la costruzione di un elemento a partire dalla property list:

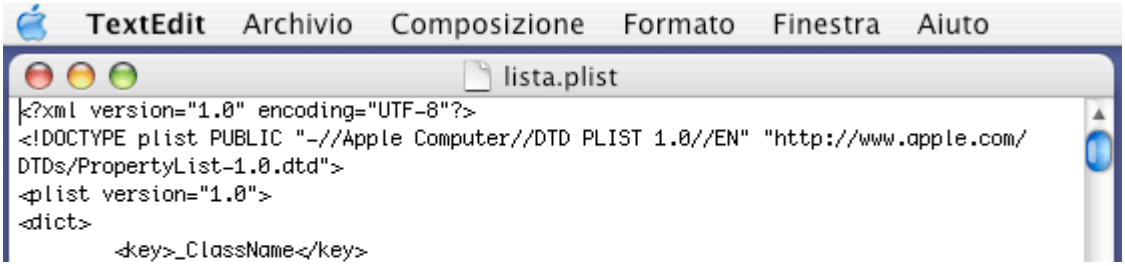
```
- (void)
loadElemXMLDesc:(NSDictionary *)dict
{
    NSArray        * newArr ;
    int            i ;
```

```
    // predispongo le variabili comuni
    [super loadElemXMLDesc:dict];
    // recupero lo array con i vari elementi del gruppo
    newArr = [ NSArray arrayWithArray:(NSArray *)
                [dict objectForKey: kStrElem_elemArray ] ] ;
    // guardo i singoli elementi
    for ( i = 0 ; i < [ newArr count ] ; i ++ )
    {
        CCE_BasicForm        * elem ;
        NSMutableDictionary * dictx ;
        // recupero la descrizione dell'elemento in XML
        dictx = [ newArr objectAtIndex: i ];
        // costruisco l'elemento stesso
        elem = [ CCE_BasicForm buildElemFromXMLDesc: dictx ] ;
        // e lo aggungo al gruppo di elementi
        [ elemArray addObject: elem ] ;
    }
}
```

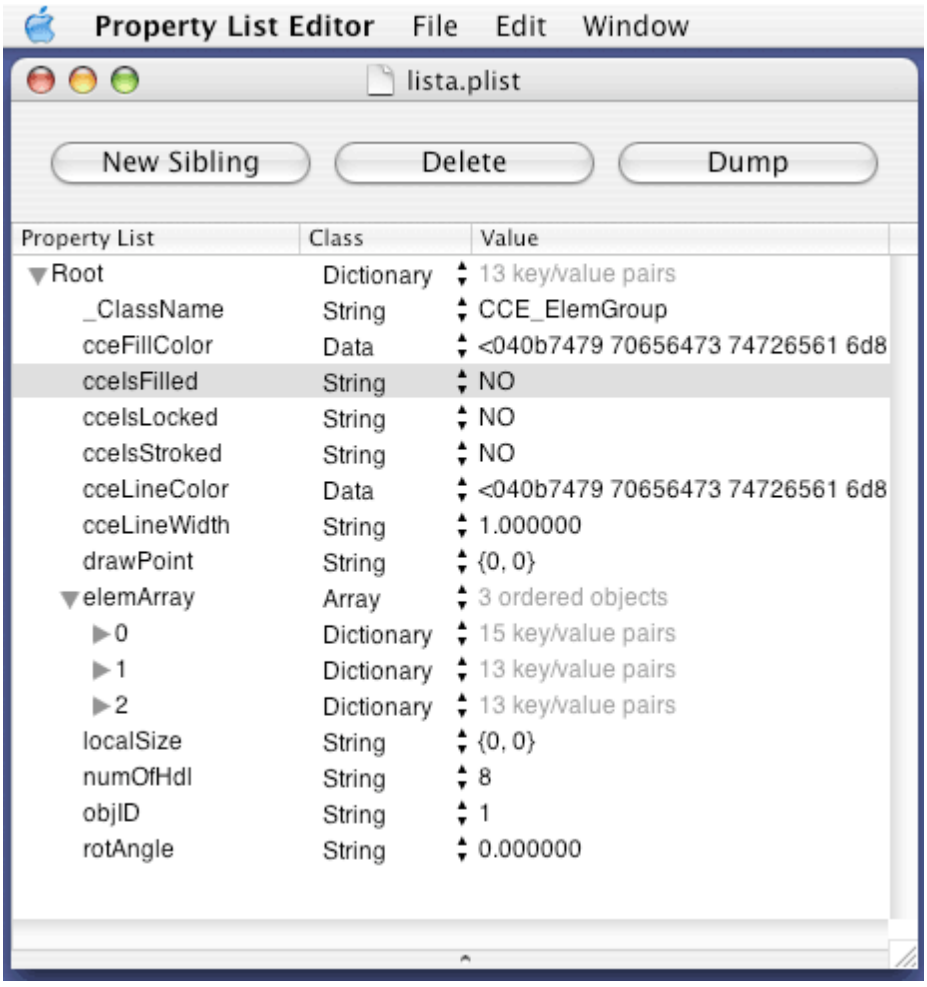
Per vedere cosa salta fuori da un meccanismo del genere, ho aggiunto ai pulsanti di debug presenti nella `CoverWinCtl` una serie di istruzioni per scrivere i dati su file (che ho copiato brutalmente dalla documentazione Apple):

```
- (void)
clicBtn02: (id)sender
{
    NSString    * path = @"/Users/djzero00/Temporaneo/lista.plist";
    NSData        * xmlData;
    NSString    * error;

    xmlData = [NSPropertyListSerialization
              dataFromPropertyList:[[ cvrView theElements] getXMLDescription
]
             format:NSPropertyListXMLFormat_v1_0
             errorDescription:&error];
    if(xmlData)
    {
        NSLog(@"No error creating XML data.");
        [xmlData writeToFile:path atomically:YES];
    }
    else
    {
        NSLog(error);
        [error release];
    }
}
```

In pratica a partire dalla property list dell'elemento `theElements` (che, essendo il gruppo principale, contiene tutti gli elementi di una copertina), si costruisce una rappresentazione della stessa sotto forma di dati nella variabile `xmlData`, che poi è scritta all'interno di un file.



```
<string>CCE_ElemGroup</string>
<key>cceFillColor</key>
<data>
BAt0eXBlZHN0cmVhbYED6IQBQISEhAdOU0NvbG9yAISECE5TT2JqZWN0AIWEAWMDhAJm
ZgEBhg==
</data>
<key>cceIsFilled</key>
<string>NO</string>
<key>cceIsLocked</key>
<string>NO</string>
<key>cceIsStroked</key>
<string>NO</string>
<key>cceLineColor</key>
<data>
BAt0eXBlZHN0cmVhbYED6IQBQISEhAdOU0NvbG9yAISECE5TT2JqZWN0AIWEAWMDhAJm
ZgABhg==
</data>
```



Il contenuto del file `lista.plist` si può esaminare tramite un qualsiasi editor di testo oppure tramite l'applicazione *Property List Editor*, fornita di serie con *Mac Os X*. Ricordo che le preferenze dell'applicazione (ma non solo, anche molte altre caratteristiche) sono conservate in property list o file XML, niente altri che file di tipo testo con una opportuna struttura imposta da una serie di tag.

Ma sto divagando. Torno al meccanismo di copia ed incolla.

COPIA E TAGLIA

Per realizzare il meccanismo di *Copia*, *Taglia* ed *Incolla*, bisogna realizzare tre metodi, che sono ovviamente chiamati:

```
- (IBAction) copy:(id)sender ;
- (IBAction) cut:(id)sender ;
- (IBAction) paste:(id)sender ;
```

Questi metodi devono essere inseriti all'interno di un *First Responder*; se il *First Responder* corrente è uno predefinito (come accade ad esempio quando si sta editando un campo di testo, o un elemento `CCE_Text`), il meccanismo è già attivo. Non è ancora attivo invece quando il *First Responder* è quello normale per una `CoverView`, cioé la `CoverView` stessa. In effetti, bisogna dire ad una `CoverView` cosa copiare e cosa incollare.

L'idea alla base del meccanismo è la seguente: se ho un po' di elementi selezionati nella copertina, posso provare a copiarli (o tagliarli). Nel caso, prendo una rappresentazione di questi elementi sotto forma di property list e la inserisco all'interno della clipboard. Se invece c'è qualcosa nella clipboard, si può provare ad incollare gli eventuali elementi `CCE_BasicForm` presenti all'interno della CoverView, estraendoli da una property list.

La clipboard è realizzata attraverso la classe `NSPasteboard` (l'ho già incontrata parlando di *drag and drop*); questa classe può contenere un po' di tutto, sia oggetti predefiniti come testo, immagini, stringhe, eccetera, ma anche tipi strani, proprietari (come sarà nel caso di elementi `CCE_BasicForm`).

Sbrigo subito il meccanismo di *Cut* (*Taglia*), che è di una semplicità inaudita:

```
- (IBAction)
cut:(id)sender
{
    [self copy:sender];
    [ self deleteSelectedElements ];
    [[theElements undoManager] setActionName: @"CoverView -> cut"];
}
```

In effetti, l'operazione di tagliare non è altro che una operazione di copiatura seguita dalla cancellazione degli elementi coinvolti.

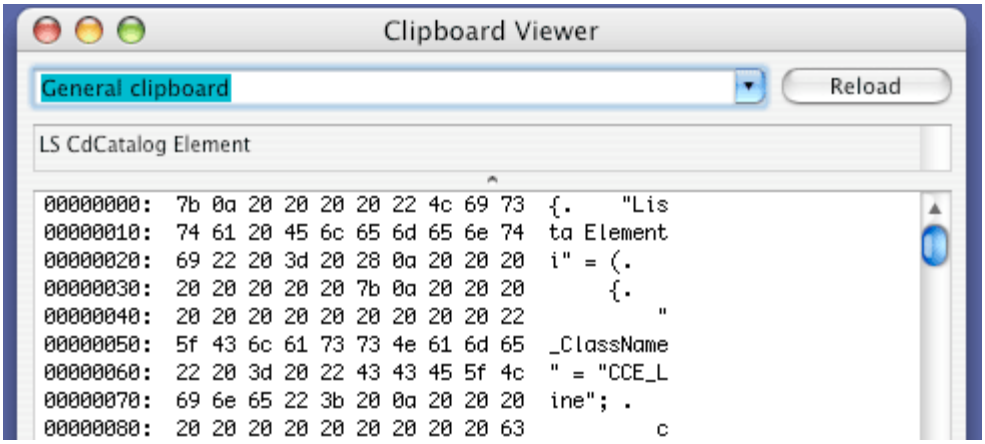
Le cose si complicano col metodo di copia. Ora, all'interno di una classe `NSPasteboard` posso inserire solo dati racchiusi in una classe `NSData`. La cosa è complicata dal fatto che è possibile eseguire operazioni di copia su più di un elemento alla volta (su tutti quelli selezionati). Allora, i dati sono così costruiti: costruisco un `NSDictionary` con all'interno un unico oggetto, che è però un `NSMutableArray`; questo array consiste di una serie di `NSDictionary`, uno per ogni elemento selezionato.

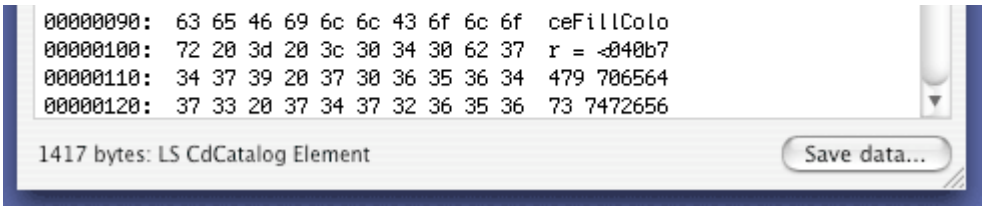
In base a quanto detto, si spiegano le operazioni di copia:

```
- (IBAction)
copy:(id)sender
```

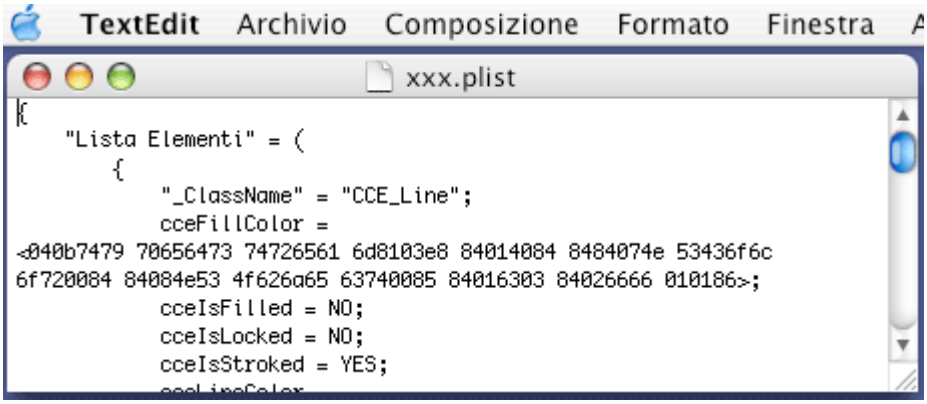
```
{
    NSMutableDictionary * copyPasteDict ;
    NSMutableArray    * elementsDict, * selArr ;
    int                i, numElem ;
    NSString        * dictRepAsStr ;
    // accedo alla clipboard
    NSPasteboard    * pb = [NSPasteboard generalPasteboard];
    // recupero gli elementi selezionati
    selArr = [ theElements getSelectedGraphics ];
    numElem = [ selArr count ];
    // se non ce ne sono, ho gia' finito
    if ( numElem == 0 ) return ;
    // dico che nella clipboard ci sara' un mio tipo proprietario
    [ pb declareTypes: [NSArray arrayWithObjects:CCE_ELEMTYPE, nil]
                owner:nil];
    // mi serve un NSData da mettere nella clipboard
    // costruisco un dizionario con gli elementi selezionati
    copyPasteDict = [ NSMutableDictionary dictionary ];
    // il dizionario e' composto da un array
    elementsDict = [NSMutableArray arrayWithCapacity: numElem ];
    for (i = 0; i < numElem; i++)
    {
        // ogni elemento dello array e' un dictionary che raccoglie
        // la descrizione dell'elemento in XML
        NSMutableDictionary * elemDict ;
        // descrizione dell'elemento in XML
        elemDict = [[selArr objectAtIndex:i] getXMLDescription] ;
        // aggiungo il dizionario allo array
        [elementsDict addObject: elemDict ];
    }
    // il dizionario principale e' composto dal solo array
    [ copyPasteDict setObject:elementsDict forKey: @"Lista Elementi"];
    // trasformo il dizionario in una stringa
    dictRepAsStr = [copyPasteDict description];
    // e quindi in un NSData che inserisco nella clipboard
    [ pb setData:[dictRepAsStr dataUsingEncoding: NSASCIIStringEncoding]
                forType: CCE_ELEMTYPE ];
}
```

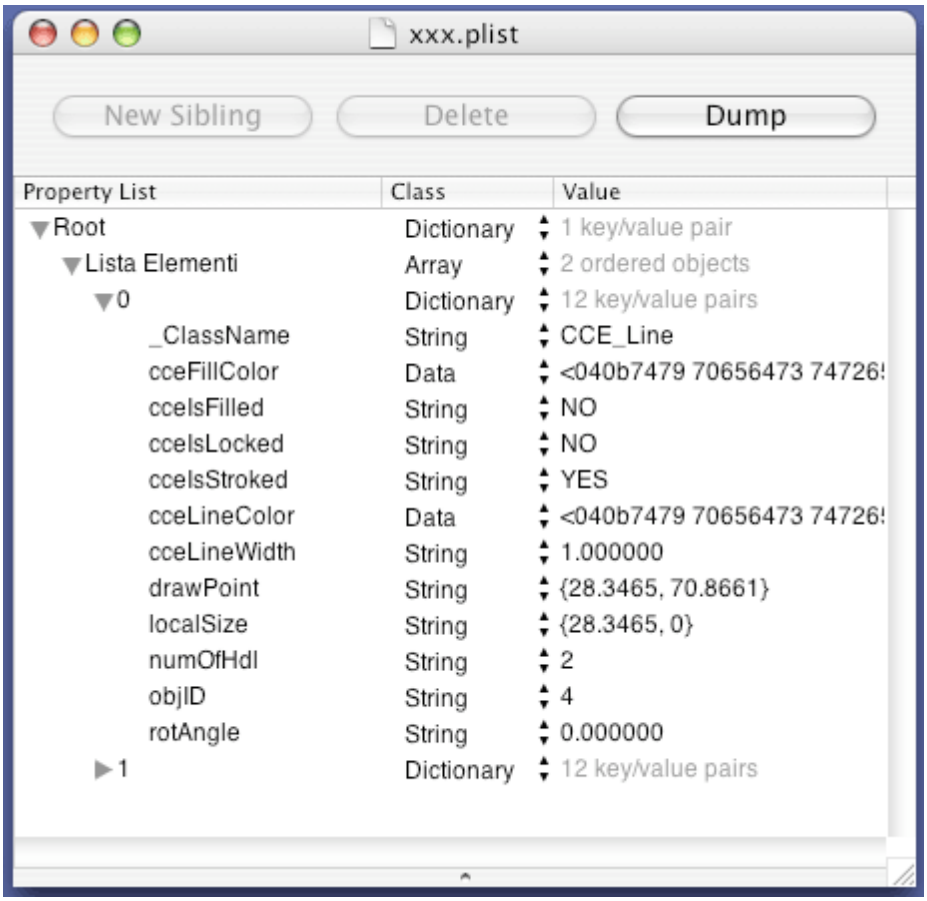
Piglio la `NSPasteboard` generica (ce ne sarebbero altre di specializzate in font, in righello, eccetera, ma non mi interessano). La prima operazioneda fare è di specificare che tipo di dati saranno inseriti all'interno della clipboard. Dico che sono dati `CCE_ELEMTYPE` (una stringa che dice `@"LS CdCatalog Element"`). Seguono poi le operazioni di costruzione del dizionario generale, e del vettore dei dizionari propri di ogni elemento. Alla fine, il dizionario principale è convertito prima in una stringa e poi in un oggetto `NSData`, che può finalmente essere passato alla clipboard.





Poiché ciò che ho copiato nella clipboard è in formato proprietario, comprensibile solo dalla mia applicazione, non se ne può vedere direttamente in contenuto con qualche applicazione d'utilità. Si può però usare l'esempio *ClipboardViewer* (fornito coi *Developer's Tool*, nella cartella `Examples/AppKit`), che appunto visualizza i dati grezzi presenti nella clipboard.





Si tratta ancora di una property list, questa volta però in formato puramente testo (non è stata effettuata l'operazione di serializzazione, che avrebbe rappresentato il tutto con i

tag).

INCOLLA

L'operazione opposta, per incollare gli elementi eventualmente presenti nella clipboard all'interno della `CoverView`, è a questo punto piuttosto facile da spiegare.

```
- (IBAction)
paste:(id)sender
{
    NSArray        * elementsDict ;
    int            i, numElem ;
    NSDictionary * copyPasteDict ;
    NSString    * dictRepAsStr ;
    // accedo alla clipboard
    NSPasteboard * pb = [NSPasteboard generalPasteboard];
    // guardo che tipo di dati e' presente
    NSString    * type = [pb availableTypeFromArray:
                        [NSArray arrayWithObjects:CCE_ELEMTYPE, nil]];
    // se ci sono tipi di dati che posso trattare
    if (! [type isEqualToString:CCE_ELEMTYPE])
        return ;
    // se arrivo qui, ho dati per me nella clipboard
    // estraggo i dati sotto forma di stringa
    dictRepAsStr = [[NSString allocWithZone:[self zone]]
                initWithData:[pb dataForType:type]
                encoding:NSASCIIStringEncoding];
    // che trasformo in un dizionario
    copyPasteDict = [dictRepAsStr propertyList];
    [dictRepAsStr release];
    // dal quale ricavo poi lo array con la descrizione degli elementi
    elementsDict = [copyPasteDict objectForKey:@"Lista Elementi"];
    numElem = [elementsDict count];
    // non dovrebbe accadere...
    if ( numElem <= 0 )    return ;
    // deseleziono tutto
    [ theElements cascadeSelection: NO ] ;
    // per ogni elemento dello array costruisco un l'elemento descritto
    for (i = 0; i < numElem ; i++)
    {
        id elem ;
              elem  =  [  CCE_BasicForm  buildElemFromXMLDesc:  [elementsDict
objectAtIndex:i]] ;
        [ elem setOwnerView: self ];
        [ elem setCceIsSelected: YES ];
        [ theElements addElem:elem atIndex: 0 ];
    }
    [[theElements undoManager] setActionName: @"CoverView -> paste"];
}
```

Si accede alla clipboard, e si cerca di capire se al suo interno si trovano dati di interesse: devono essere del tipo `CCE_ELEMTYPE`, dacché al momento l'applicazione non è in grado di interpretare altri dati. Se è il caso, le operazioni sono concettualmente lineari: si estraggono i dati, li si trasformano in una stringa, si trasforma la stringa in una property list, che è il dizionario principale. Da questo dizionario principale recupero il vettore dei dizionari dei singoli elementi. Per ogni dizionario, costruisco l'elemento con metodo apposito, e poi lo inserisco all'interno del vettore `theElements`, cioè lo aggiungo agli elementi della `CoverView`.

Non occorre nemmeno predisporre strani meccanismi per lo *Undo*, in quanto già presenti all'interno dei metodi `addElem` e `deleteSelectedElements`.

INTERVALLO: CORREZIONI

Usando un po' l'applicazione, mi sono accorto di qualche errore che mi è scappato nei capitoli precedenti.

La classe `PrtMrgWinCtl`, ad esempio, sbagliava a gestire i margini di stampa, per cui, in realtà, l'intero meccanismo di predisposizione dei margini di stampa non funzionava. L'errore stava nell'assegnazione alla variabile sbagliata dei valori dei margini destro, superiore e inferiore. Pazienza.

Già che c'ero, mi sono accorto che l'impostazione dei margini e del formato della pagina di una copertina andava perduta tra una sessione e l'altra. Questo perché la variabile `pInfo` (appunto della classe `NSPrintInfo`, nata a questo scopo) non era coinvolta nelle operazioni di salvataggio del template, pur essendo una variabile d'istanza di `CoverView`. Messo a posto anche questo (ma questo significa che il formato dei template è cambiato; poco male, ne ho fatti altri più bellini).

L'errore più brutto però coinvolgeva gli elementi `CCE_Line`, che subivano misteriose metamorfosi quando modificati attraverso la finestra `ElemInfoWinCtl`. Il problema è dovuto al metodo `resizeMeWith:` utilizzato, che era quello generico. Per la classe `CCE_Line` va invece riscritto per tenere conto che la dimensione `height`, per via del meccanismo di definizione delle linee, è sempre nullo. Ho dunque sovrascritto il metodo:

```
- (void)
resizeMeWith: (NSSize) newSize
{
    NSSize        realSize ;
    float        aaa = rotAngle/180*M_PI ;
    // gestione dello undo del movimento
    [[self undoManager] setActionName: @"BasicForm -> resizeMe"];
          [[[self   undoManager]   prepareWithInvocationTarget:   [ownerView
theElements]]
        resizeElem: self withSize: localSize ];
    // ricalcolo la posizione del secondo punto
     realSize.width = newSize.width * cos ( aaa ) + newSize.height * sin (
aaa ) ;
     realSize.height = newSize.width * sin ( aaa ) + newSize.height * cos (
aaa ) ;
    [self buildMeFromPt: drawPoint withSize: realSize ];
}
```

Infine, c'era un errore quando si cercava di selezionare più elementi tracciando un rettangolo che li racchiudesse tutti. Se la costruzione del rettangolo partiva dall'alto a sinistra ed andava verso il basso a destra, tutto andava bene. Se invece si procedeva in altro modo, poiché il rettangolo risultava con almeno una dimensione negativa, le successive istruzioni che cercavano di determinare le intersezioni tra il rettangolo di selezione e i rettangoli delimitanti gli altri elementi fallivano. Ho quindi modificato il metodo `handleMouseClick` di `CoverView`:

```
- (void)
handleMouseClick: (NSEvent *)theEvent
{
    ...
    curElem = [self getClickedElem:whereClick];
    // se c'e' un elemento sotto il mouse
    if ( curElem )
    {
        ...
    }
    else
    {
        int        i , numElem ;
        CCE_BasicForm * elem ;
        NSRect selRect, locrect ;
        // comincio la selezione multipla
        selRect = [ self handleRectSelect: theEvent ];
        locrect = arrangeRect( selRect.size );
        selRect.origin.x += locrect.origin.x ;
        selRect.origin.y += locrect.origin.y ;
        selRect.size = locrect.size ;
        // ciclo su tutti gli elementi del gruppo
        numElem = [ [ theElements elemArray] count ];
        for ( i = 0 ; i < numElem ; i ++ )
        {
            elem = [ [ theElements elemArray] objectAtIndex: i ];
             [ elem selectMeIfInRect: selRect addingToSelez: shiftKeyPress]
;
        }
    }
}
```

Ho corretto il rettangolo con l'ormai consueta funzione `arrangeRect(.)`, costruendone uno con dimensioni sicuramente positive.

Già che c'ero, ho introdotto il metodo associato alla voce di menu *Select All* o *Seleziona Tutto*, che è inspiegabilmente assente. È piuttosto semplice, basta sapere il nome giusto, che per altro si può vedere esaminando la voce di menu:

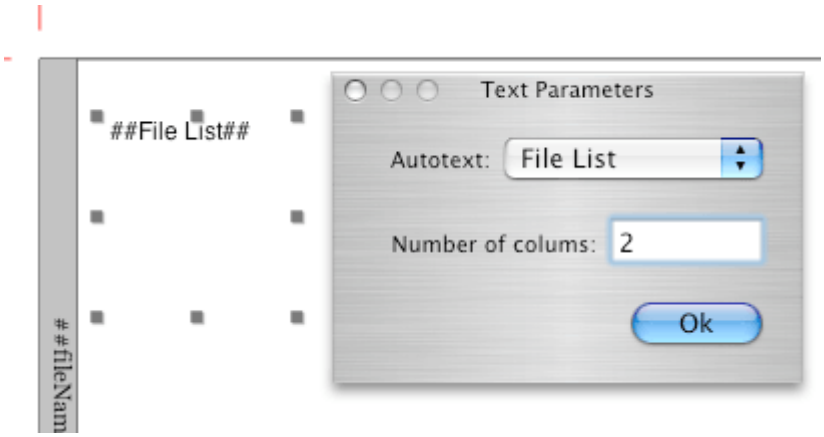
```
- (IBAction)
selectAll:(id)sender
{
    [ theElements cascadeSelection: YES ] ;
}
```

COLONNE MULTIPLE

Ho scoperto che è piuttosto semplice fare in modo che il testo presentato all'interno di una elemento `CCE_Text` sia distribuito su più colonne. La maggior parte del lavoro è svolto dalle classi di Cocoa predisposte alla bisogna, ovvero `NSLayoutManager` e `NSTextContainer`.

Prima di tutto, ho però bisogno di un meccanismo che permetta all'utente di stabilire su quante colonne distribuire il testo. Ho così modificato il menu contestuale associato agli elementi `CCE_Text`: ho eliminato le voci per l'impostazione del testo automatico, e le ho sostituite con l'unica voce *Text Params*. Selezionando questa voce, faccio

comparire una nuova finestra.



In questa finestra ripristino il menu del testo automatico (sotto forma di menu pop up, ed aggiungo un campo di testo dove impostare il numero delle colonne.

Non mi dilungo sulle solite questioni relative alle finestre accessorie e sulle modifiche alla classe `CoverView` per la gestione del tutto, ma mi limito a dire che con queste finestra si modificano le due variabili d'istanza `automaticText` e (nuova) `numOfColums`.

Adesso, in base al valore di `numOfColums`, bisogna predisporre la classe `NSTextStorage` perché la sua visualizzazione sia corretta.

C'è un primo metodo, `prepareTextStorage`, che è chiamato da tutti i metodi che inizializzano un elemento della classe `CCE_Text`, che si limita ad aggiungere un layout manager all'elemento:

```
- (void)
prepareTextStorage: (NSTextStorage *) ts
{
    NSLayoutManager *layoutManager;
    // layout gestisce come il testo e' rappresentato
    layoutManager = [[NSLayoutManager alloc] init] ;
    [ ts addLayoutManager:layoutManager];
    [ self adjustTextColumn ] ;
    [ layoutManager release ] ;
}
```

Il lavoro più interessante è svolto invece all'interno di `adjustTextColumn`, da chiamare anche ogni volta che cambia il numero delle colonne:

```
- (void)
adjustTextColumn
{
    int                 i, notcAr ;
    NSLayoutManager     * lm ;
    // recupero il layout manager del textStorage
    lm = [ [ textStorage layoutManagers ] objectAtIndex: 0] ;
    // guardo quanti textContainer sono presenti
    notcAr = [[ lm textContainers ] count ];
    // li tolgo tutti, che non mi servono piu'
    for ( i = (notcAr-1); i >= 0 ; i -- )
```

```
        [ lm removeTextContainerAtIndex: i ];
    // ci aggiungo quelli nuovi in base al numero di colonne
    for ( i = 0; i < numOfColumns ; i ++ )
    {
        NSTextContainer * tc;
        // ricalcolo le dimensioni per non averne negative
        NSRect locrect = arrangeRect( localSize );
        // costruisco un nuovo textContainer
        tc = [[NSTextContainer alloc] init ];
        // dovrei specificare un NSSize per il textcontainer
        [ tc setContainerSize:
                NSMakeSize( locrect.size.width / numOfColumns - 2,
                            locrect.size.height) ];
        // lo aggiungo al layout manager
        [ lm addTextContainer: tc];
        [ tc release ] ;
    }
}
```

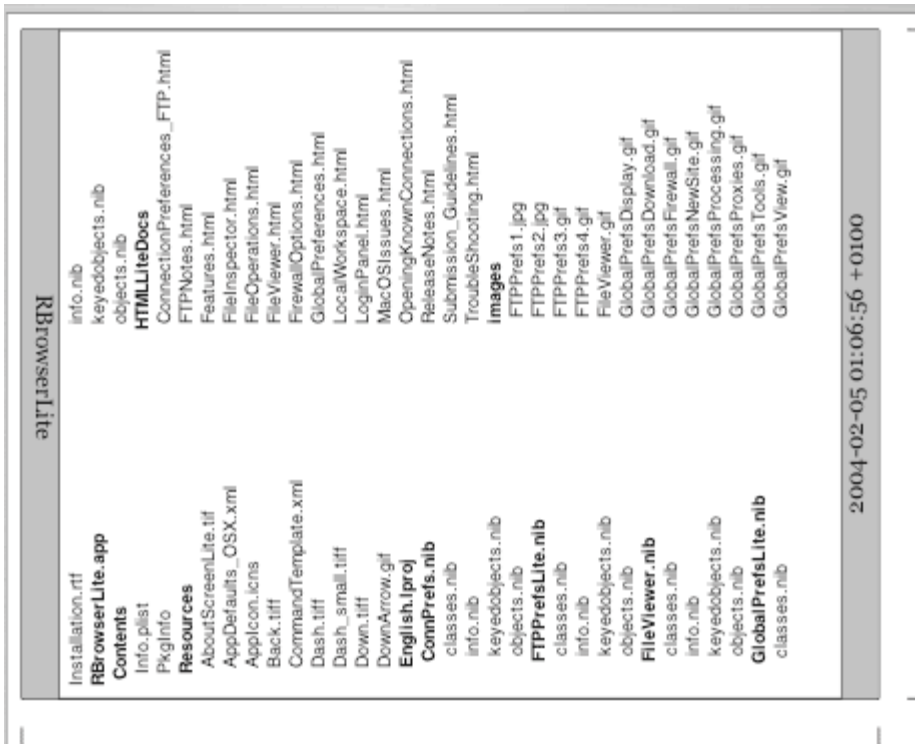
Il metodo è piuttosto semplice: toglie tutti gli oggetti `NSTextStorage` presenti, e ne aggiunge tanti quante sono le colonne. Le dimensioni di questi `NSTextContainer` sono poi le stesse in altezza, mentre in larghezza dipendono appunto dal numero delle colonne presenti.

Quando poi si deve disegnare l'elemento `CCE_Text`, si procede così:

```
- (void)
specificDrawing: (NSRect) inRect
{
    // determino punto e dimensione di dove disegnare
    // per tenere conto di eventuali dimensioni negative
    // disegno del testo
    NSRect locrect = arrangeRect( localSize );
    [ super specificDrawing: inRect ] ;
    // se sto modificando il testo...
    if ( ! editingText || [NSGraphicsContext currentContextDrawingToScreen]
)
    {
        int            i ;
        NSLayoutManager *layoutManager ;
        NSArray    * tcAr ;
        // recupero il layout manager
         layoutManager = [ [ textStorage layoutManagers ] objectAtIndex: 0]
;
        // recupero tutti i textContainer (sono numOfColumns)
        tcAr = [ layoutManager textContainers ] ;
        // li spazzolo tutti
        for ( i = 0 ; i < numOfColumns ; i ++ )
        {
            NSRange glyphRange ;
            NSPoint    pt ;
            NSTextContainer * tc1 ;
            // recupero i textContainer in ordine
            tc1 = [ tcAr objectAtIndex: i ] ;
            // predispongo la loro dimensione
            [ tc1 setContainerSize:
                NSMakeSize( locrect.size.width / numOfColumns - 2,
                            locrect.size.height) ];
            // recupero il range del testo
            glyphRange = [layoutManager glyphRangeForTextContainer: tc1 ];
            // calcolo il punto di origine del singolo textContainer
```

```
            pt = NSMakePoint(    locrect.origin.x +
                                    i * locrect.size.width / numOfColumns +
2 ,
                                locrect.origin.y );
            // disegno i caratteri nel singolo textContainer
            [ layoutManager drawGlyphsForGlyphRange:glyphRange
                    atPoint: pt ];
        }
    }
    else
    {    // sto editando il testo e non sto stampando
        // e' tutto fatto dalla NSTextView
    }
}
```

In pratica, si esaminano in successione i vari `NSTextContainer`, in numero pari a quello delle colonne; per ognuno si essi si ricalcolano le dimensioni, e si determina il punto dove sarà rappresentato. L'ordinata di questo punto è sempre la stessa, mentre l'ascissa è determinata tenendo conto della dimensione delle colonne.



Quando invece il testo è in corso di modifica, continuo a mostrare sempre una `NSTextView`, ma larga quanto una colonna.

Capitolo 059 - Suonala ancora Sam

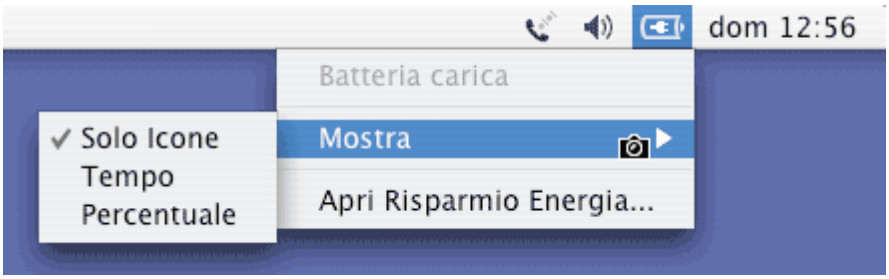
Questo capitolo esce dal flusso relativo alla catalogazione di CD affrontando un diverso argomento, o meglio, alcuni argomenti differenti, racchiusi all'interno di una cornice unificante: come interagire con *iTunes*?

Sorgenti: c'è un tutorial su CocoaDevCentral, che ho ripreso quasi uguale. Di più: questo capitolo è una scopiazzatura...

Prima stesura: 19 settembre 2004

Il problema e la soluzione

Spesso mi accade di lavorare al computer con *iTunes* in sottofondo che suona brani musicali. La mia età è tale per cui spesso non ricordo più cosa abbia programmato di suonare: mi trovo quindi a chiedermi quale canzona sia suonata in quel momento, sembrandomi nota ma non avendo presente autore e titolo. Altre volte invece, suona il telefono o ci sono altri disturbi, e voglio interrompere il flusso musicale. In entrambi i casi, devo abbandonare l'applicazione in primo piano, richiamare iTunes, guardare cosa sta suonando oppure fermare la musica. Ritornando dopo le incombenze, riprendo la musica e ritorno all'applicazione iniziale. Sarebbe bello, mi sono detto, se tutto questo potesse avvenire con meno fatica.



L'idea è di utilizzare un menu sulla barra principale, come ad esempio quelli relativi al volume, alla carica della batteria, la rappresentazione della data e ora, eccetera.

Questo menu dovrà presentare le ultime dieci canzoni suonate da iTunes, ed aggiungere un paio di comandi per la gestione di iTunes: l'apertura di iTunes, la fermata o il riavvio della riproduzione.

Gli argomenti principali del capitolo sono sostanzialmente due: la costruzione di un menu nella barra principale e l'interazione con iTunes. Ora, l'interazione con applicazioni esterne può avvenire comodamente solo le applicazioni comunicanti condividono un linguaggio o una struttura comune. L'unico modo che ho trovato per comunicare tra la mia nuova applicazione e iTunes è stato *AppleScript*. Quindi, all'interno di questo capitolo, ci sarà una programmazione mista ObjectiveC-Applescript, e un meccanismo per eseguire script Applescript all'interno del progetto.

GLI SCRIPT APPLECRIPT

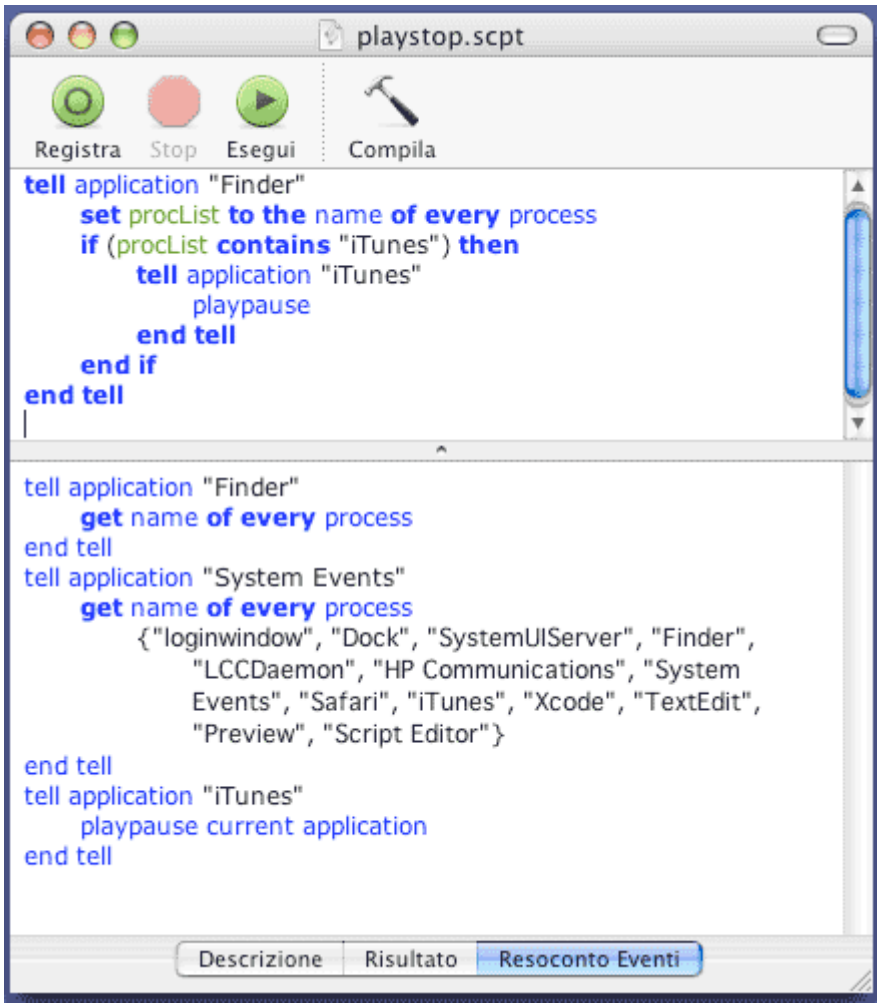
Per prima cosa, scrivo i due script che servono per interagire con iTunes. I due script servono il primo per recuperare informazioni sulla canzone in esecuzione al momento, mentre il secondo serve per fermare o far partire l'esecuzione. Non ho intenzione qui di fare un corso su AppleScript (so cos'è, so scrivere degli script, ma non mi sento ancora di raccontare tutto su questo linguaggio). Tuttavia, gli script presentati sono estesamente commentati, e il linguaggio è in effetti piuttosto semplice.

Redigo gli script avvalendomi dell'applicazione *Script Editor*, presente in ogni installazione standard di Mac OS X. Parto dal secondo script, quello che controlla l'esecuzione di iTunes; è molto semplice:

```
tell application "Finder"
    set procList to the name of every process
    if (procList contains "iTunes") then
        tell application "iTunes"
            playpause
        end tell
    end if
end tell
```

Per prima cosa, si interagisce con il Finder; alla variabile `procList` attribuisco la lista delle applicazioni in quel momento in esecuzione. Se all'interno di questa lista compare iTunes, allora comunico direttamente con iTunes. Dico di eseguire il comando `playpause`. Questo comando serve a modificare lo stato corrente di riproduzione: se è attiva, ferma la riproduzione, se ferma, l'attiva.

Si può conoscere l'intera serie di comandi e di proprietà AppleScript messe a disposizione da iTunes consultando il *dizionario AppleScript* di iTunes stesso; basta usare *Script Editor* e la voce di menu apposita (*Apri Dizionario...*).



Sempre attraverso Script Editor, è possibile provare l'efficacia dello script, operando sul pulsante di esecuzione, ed anche rendersi conto dell'andamento delle operazioni.

Sono pronto adesso per lo script successivo, un po' più complicato; scopo dello script è di ricavare informazioni sulla canzone (traccia) correntemente in esecuzione. Dal dizionario AppleScript di iTunes ricavo che è possibile conoscere tale canzone attraverso la proprietà `current track`, dalla quale, a sua volta, è possibile ricavare autore, titolo, e tutte le altre informazioni accessorie di un file mp3 (i tag).

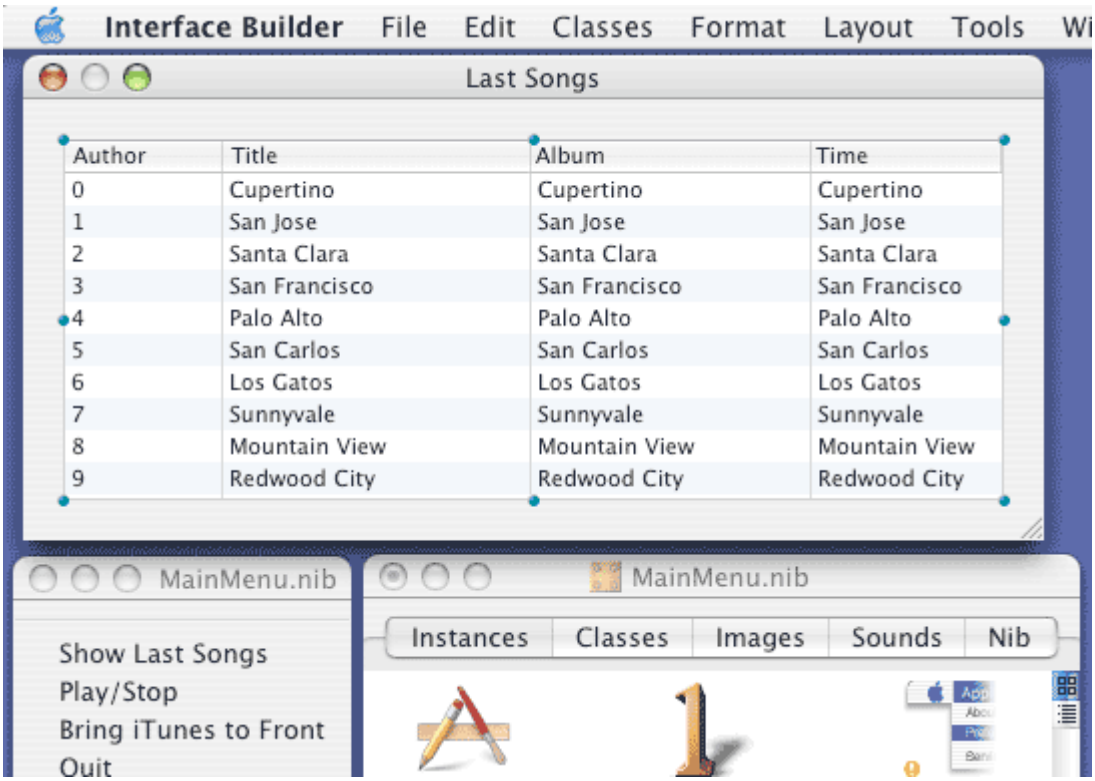
```
tell application "Finder"
    set procList to the name of every process
    if (procList contains "iTunes") then
        tell application "iTunes"
            if the player state is playing then
                -- leggo qualche informazione sulla traccia
                set curTrack to the current track
                -- indice della playlist e della traccia
                set trkPlayIdx to the index of the current playlist
                set trkIdx to the index of curTrack
                -- autore, titolo, album e durata
                set trkAut to the artist of curTrack
                set trkName to the name of curTrack
                set trkAlb to the album of curTrack
                set trkDur to the duration of curTrack
                -- restituisco tutto in una riga separata da tab
                return (trkPlayIdx & tab & trkIdx & tab & trkAut ¬
```

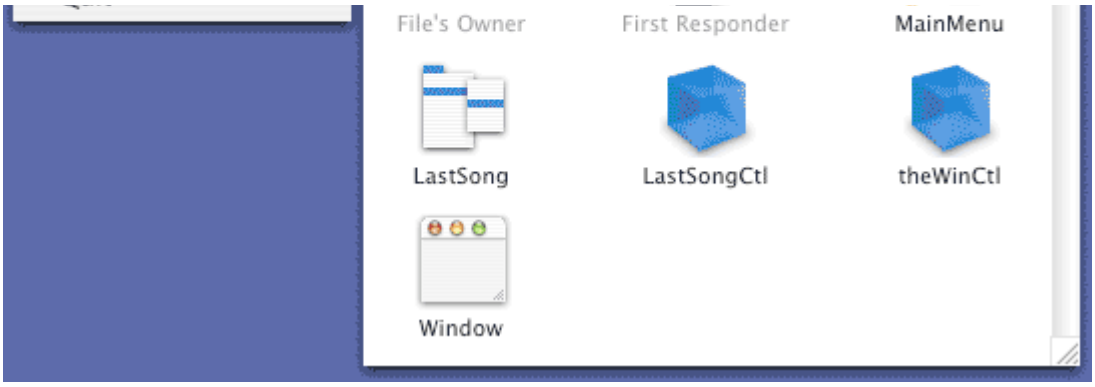
```
                        & tab & trkName & tab & trkAlb & tab & trkDur) as
string
            else
                -- errore: iTunes non sta suonando
                return "ERR_ITUNES" & tab & "iTunes not playing"
            end if
        end tell
    else
        -- errore: iTunes proprio non c'e'
        return "ERR_ITUNES" & tab & "iTunes not present"
    end if
end tell
```

Lo script comincia nello stesso modo del precedente, verificando la presenza di iTunes tra le applicazioni in esecuzione. Se questo non è vero, restituisce una stringa con un messaggio di errore. Se iTunes è presente, verifica che ci sia effettivamente una canzone in esecuzione: basta consultare la proprietà `player state`, che deve essere `playing`. A questo punto, si ricava la traccia, ed alcune informazioni relative, giusto per far vedere che ne sono capace. In effetti basterebbe autore e titolo, secondo i miei requisiti iniziali; tuttavia, metto da parte anche un paio di valori (indice di playlist ed indice di traccia) per poter in un momento successivo risuonare la traccia (sarà chiaro più avanti). Tutte queste informazioni sono ritornate all'interno di una stringa, dove i vari campi sono separati da un carattere di tabulazione. Anche di questo script se ne può verificare l'efficacia direttamente tramite Script Editor.

L'INTERFACCIA DI LASTSONGS

Chiamo la mia applicazione *LastSongs* e la costruisco a partire da *XCode*; si tratta di una *Cocoa Application*, ovvero di una applicazione che non è basata su documenti; infatti consta di un menu e di una finestra (che chiamerò accessoria, visto che serve a molto poco).





Il menu è un altro menu rispetto al `MainMenu` prodotto di default da XCode; è infatti il menu che poi sarà inserito nella barra principale dei menu, e non il menu dell'applicazione (che per altro non vorrò mai utilizzare). La finestra aggiuntiva serve solo per mostrare in piena gloria (cioè con tutti gli attributi ricavati da iTunes) le canzoni suonate di recente, ma non ha un uso sensato. Questa finestra è visualizzata quando si seleziona la voce *Show Last Song* dal menu.

La voce di menu *Play/Stop* avvia o ferma la riproduzione, e dovrà essere associata all'esecuzione del primo script sopra presentato. La voce *Bring iTunes to the front* intende portare davanti a tutte le alpre applicazioni iTunes. Se iTunes non è attivo, lo lancia direttamente. Infine la voce *Quit* esce dall'applicazione; è presente esplicitamente dal momento che il menu standard dell'applicazione non sarà visualizzato.

All'interno di *Interface Builder* dichiaro e definisco subito due oggetti; il primo, `LastSongCtl`, dell'omonima (nuova) classe, funziona da controllore dell'intera applicazione (sarà in effetti l'unica classe originale dell'applicazione). A questa classe fanno riferimento tutte le voci di menu; funziona anche da classe sorgente di dati e delagata dalla `NSTableView` presente nella finestra. L'oggetto `theWinCtl` è uno standard `NSWindowController`, presente per poter gestire la visualizzazione della finestra, ma non ha alcuna particolarità.

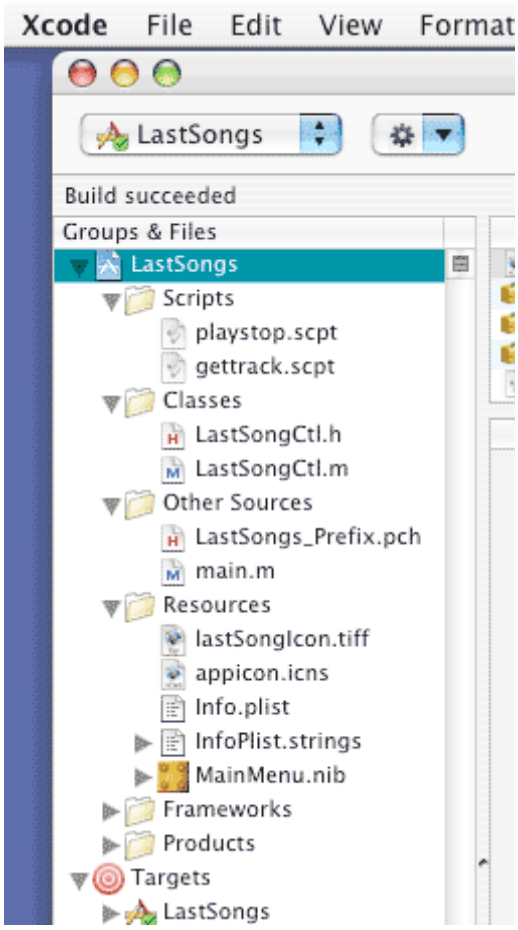
La classe `LastSongCtl` possiede outlet verso il menu e la `NSTableView`, possiede come variabile d'istanza la lista delle (massimo dieci) ultime canzone suonate da iTunes e due oggetti della classe `NSAppleScript`.

La classe NSAppleScript

Cocoa mette a disposizione la classe `NSAppleScript` per caricare, compilare ed eseguire AppleScript; ad esempio, si può caricare uno script da un file, oppure produrne uno al volo all'interno di una `NSString`; costruita l'istanza della classe, lo script può essere semplicemente compilato o direttamente mandato in esecuzione; si otterrà un risultato (un oggetto della classe `NSAppleEventDescriptor`) ed eventualmente un messaggio di errore, il tutto sotto controllo di istruzioni Objective C; che meraviglia.

Utilizzo due oggetti di questa classe per conservare, compilati e pronti all'uso, i due script sopra preparati tramite lo Script Editor. Preparo tutto all'inizializzazione della classe `LastSongCtl`:

```
-(id)
init
{
    // inizializzo
    self=[super init] ;
    if( self )
    {
        NSURL        * scriptURL ;
        NSString     * scriptPath ;
        // recupero i due script e li inserisco nelle mie variabili
        scriptPath = [[[NSBundle mainBundle] resourcePath]
            stringByAppendingPathComponent:@"gettrack.scpt"] ;
        scriptURL = [NSURL fileURLWithPath: scriptPath ];
        getTrackScript =[[NSAppleScript alloc]
            initWithContentsOfURL:scriptURL error:nil];
        scriptPath = [[[NSBundle mainBundle] resourcePath]
            stringByAppendingPathComponent:@"playStop.scpt"] ;
        scriptURL = [NSURL fileURLWithPath: scriptPath ];
        playStopScript =[[NSAppleScript alloc]
            initWithContentsOfURL:scriptURL error:nil];
        // inizializzo la lista delle canzoni e le altre variabili
        listaCanzoni = [[NSMutableArray alloc ]init];
        curNumOfTracks = 0 ;
    }
    return self;
}
```



I due file che contengono gli script sono stati aggiunti al progetto, in modo che poi saranno incorporati all'interno dell'applicazione. Il percorso del file è quindi dato dalla cartella *Resources* presente all'interno del *bundle* dell'applicazione, percorso al quale si

deve ovviamente aggiungere il nome del file. Per inizializzare un oggetto `NSAppleScript` è però necessario un URL o una stringa (un banale file pare non gli basti); ho scelto la strada di produrre un URL e di inizializzare con questo, piuttosto che caricare in una stringa il contenuto del file e inizializzare l'oggetto `NSAppleScript` dalla stringa ottenuta.

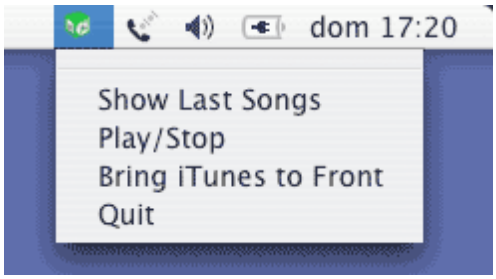
Se ora è abbastanza chiaro quando debba essere eseguito lo script `playStop.scpt`, c'è il problema di quando eseguire l'altro script, quello che ricava da iTunes le informazioni relative alla canzone in corso di riproduzione. In mancanza di partecipazione attiva da parte di iTunes, l'unico metodo rimane quello di chiederlo ripetutamente e periodicamente. In altre parole, occorre eseguire più volte, ad intervalli di tempo fissi è l'opzione più semplice, un metodo che ricavi le informazioni. Ho già incontrato un meccanismo del genere quando ho realizzato la finestra *About* con testo in movimento: si tratta di utilizzare un `NSTimer`.

La classe `NSTimer` in pratica attende un dato intervallo di tempo (stabilito in sede di inizializzazione), per poi mandare in esecuzione un determinato metodo. La cosa può essere o meno ripetuta in continuazione. Una finestra che rappresenta un orologio con sole ore e minuti, ad esempio, potrebbe avere un timer che scatta ogni sessanta secondi. L'applicazione *LastSongs* usa un timer come meccanismo principale e unico per svolgere tutte le operazioni.

AGGIORNAMENTO DEL MENU

Il metodo `awakeFromNib` è responsabile della predisposizione di tutte le operazioni: installa il menu nella barra principale, e fa partire il timer periodico:

```
- (void)
awakeFromNib
{
    // recupero un puntatore alla barra dei menu
    NSStatusBar *bar = [NSStatusBar systemStatusBar];
    // costruisco il menu nella status bar
    lastSongMenu = [bar statusItemWithLength:NSSquareStatusItemLength];
    [ lastSongMenu retain ];
    // metto a posto le caratteristiche del menu
    [lastSongMenu setHighlightMode:YES];
    [lastSongMenu setImage: [ NSImage imageNamed: @"lastSongIcon" ] ];
    [lastSongMenu setMenu: theMenu];
    [lastSongMenu setEnabled: YES];
    // aggiusto la tabella con l'elenco delle ultime canzoni
    [ theTable setDoubleAction: @selector(playThatSong:) ] ;
    [ theTable setTarget: self ] ;
    // predispongo il timer per l'aggiornamento del menu
    menuUpdater = [NSTimer
            scheduledTimerWithTimeInterval: TIMER_UPDATE_INTERVAL
            target:         self        // chiamo questo metodo
            selector:    @selector(readTrackFromITunes:)
            userInfo:    nil
            repeats:    YES] ;    // e continuo
    [ menuUpdater retain] ;
    // faccio partire il timer la prima volta
    // che cosi' aggiorno tutto l'ambaradan
    [menuUpdater fire];
}
```



Le prime istruzioni sono responsabili per l'installazione del menu sulla barra. Si recupera l'istanza condivisa a tutte le applicazioni attraverso il metodo `systemStatusBar`, e si riserva uno spazio fisso dove inserirlo. Le istruzioni successive inseriscono il menu `theMenu` (costruito all'interno del nib) e fanno in modo che sia visualizzata l'immagine specificata (contenuta in un file che è inserito all'interno del progetto, per cui è sufficiente il metodo `imageNamed:` per prelevarla). Alternativamente, o contemporaneamente, avrei potuto impostare una stringa con il metodo `setTitle:`, ma l'icona occupa meno spazio ed è più carina (nei limiti delle mie capacità artistiche).

Le due istruzioni successive si riferiscono alla tabella della finestra accessoria; in questa tabella sono visualizzate le ultime canzoni eseguite. Con un doppio clic su una di queste canzoni, la canzone stessa sarà riprodotta di nuovo: è questo l'effetto finale del metodo `playThatSong:`. Infine, le ultime tre istruzioni impostano il timer e la funzione da eseguire allo scattare del timer. Il metodo `readTrackFromITunes:` della classe `self` (e quindi, `LastSongCtl`) deve essere eseguito ogni `TIMER_UPDATE_INTERVAL` secondi (che sono quindici). Il metodo `fire` serve per far eseguire subito questo metodo e provocare il primo aggiornamento del menu (e della tabella).

Tutto (o quasi) il resto del lavoro è svolto quindi dal metodo `readTrackFromITunes:`. Le operazioni sono divise in due momenti successivi: l'aggiornamento della base di dati interna (ovvero, la lista delle canzoni suonate), e l'aggiornamento della visualizzazione (il menu e la tabella).

```
-(void)
readTrackFromITunes:(NSTimer *)timer
{
    int             i, whereIs, numMenu ;
    NSString    * curTrack ;
    NSArray         * trackInfo ;
    NSMenuItem    * item;
    // - - - - - - aggiorno i dati interni
    // recupero autore e titolo della traccia in esecuzione
     curTrack = [ [getTrackScript executeAndReturnError: nil] stringValue ]
;
    trackInfo = [ curTrack componentsSeparatedByString: @"\t" ];
    // se c'e' un errore, lascio perdere
    if([ [trackInfo objectAtIndex: 0 ] isEqualToString:@"ERR_ITUNES"])
        return ;
```

La prima operazione da eseguire è ovviamente lo script; allo scopo, è sufficiente chiamare il metodo `executeAndReturnError:`. Questo metodo compila, se necessario, lo script; lo esegue e riporta il risultato all'interno di un oggetto di classe `NSAppleEventDescriptor`. In realtà l'oggetto in sé non interessa: piuttosto, mi

interessa ciò che ha restituito lo script stesso. Lo si ottiene con il metodo `stringValue`, che produce appunto una stringa. Questa è la stringa che ha prodotto lo script: una riga di elementi separati dal carattere di tabulazione; spezzo allora la stringa in tante sottostringhe, in modo da accedere ai vari campi. In questo modo posso anche verificare se per caso il metodo ha restituito una stringa di errore: in tal caso, non ho nulla da fare (perché iTunes non è presente, o non sta suonando).

```
    // se la lista e' vuota, aggiungo sen'altro
    if( curNumOfTracks == 0 )
    {
        [listaCanzoni addObject:curTrack];
        curNumOfTracks += 1 ;
    }
    else
    {
        // il piu' delle volte, e' l'ultima inserita
        // in tal caso, devo fare nulla
         if( [ curTrack isEqualToString: [ listaCanzoni objectAtIndex:0 ] ]
)
            return ;
        // se arrivo qui, e' cambiata la canzone
        // ma potrebbe essere gia' stata suonata in tempi recenti
        whereIs = [ listaCanzoni indexOfObject:curTrack] ;
        if ( whereIs == NSNotFound )
        {
            // no, proprio non c'e'
            // se ho gia' raggiunto il massimo visualizzabile
            if ( curNumOfTracks == MAX_NUM_OF_TRACKS )
            {
                // elimino l'ultima traccia
                curNumOfTracks -= 1 ;
                [ listaCanzoni removeObjectAtIndex: curNumOfTracks ] ;
            }
            // inserisco la nuova traccia in testa
            [ listaCanzoni insertObject: curTrack atIndex:0 ] ;
            curNumOfTracks += 1 ;
        }
        else
        {
            // c'e' gia', la sposto da dove si trova in testa
                [ listaCanzoni exchangeObjectAtIndex: 0 withObjectAtIndex:
whereIs ] ;
        }
    }
```

A questo punto, sono nella situazione in cui iTunes è vivo e lotta assieme a noi. Aggiorno la base di dati interna, ovvero il vettore `listaCanzoni`; distinguo vari casi.

Se il vettore è vuoto, aggiungo senz'altro la nuova canzona; se invece c'è già qualcosa, verifico se per caso la canzone in esame è già presente. In effetti, questo metodo è chiamato ogni 15 secondi: nella maggior parte dei casi la canzone in esecuzione è già stata inserita (a meno che non si stia ascoltando qualche disco piuttosto sperimentale di mia conoscenza), ed è proprio l'ultima ad essere stata inserita. In tutto questi casi, continua a non esserci alcunché da fare.

Finalmente arrivano i casi interessanti: la prima cosa che ci può capitare è che la canzone non è presente nella lista. Devo allora aggiungerla in cima, facendo attenzione a non superare il limite massimo di canzoni previste (che ho fissato a dieci, ma solo per

avere un numero). Se ho già raggiunto il limite, semplicemente elimino la canzone più vecchia (quella in fondo al vettore), per poi passare senz'altro ad aggiungere la nuova canzone.

Tuttavia, potrebbe capitare che la canzone in corso è stata già eseguita in precedenza (entro le precedenti dieci). In tal caso la porto in prima posizione, ed al suo posto ci infilo la canzone precedente (sarebbe stato meglio scalare tutte le altre canzoni, ma non avevo voglia di pensare).

Insomma, il risultato finale di questa serie di operazione è stata quella di avere modificato il vettore `listaCanzoni`. Adesso, bisogna modificare l'aspetto dell'applicazione: il menu e la lista presente nella finestra accessoria:

```
    // - - - - - - aggiorno l'aspetto visuale
    numMenu = [[theMenu itemArray] count ] ;
    // tolgo tutte le voci di traccia presenti
    for ( i = (numMenu-6) ; i >= 0 ; i -- )
    {
        [ theMenu removeItemAtIndex: i ] ;
    }
    // ed adesso lo riempio con le voci da listacanzoni
    for ( i = (curNumOfTracks-1) ; i >= 0 ; i -- )
    {
        // la spezzo nei vari componenti
        trackInfo = [ [ listaCanzoni objectAtIndex: i ]
            componentsSeparatedByString: @"\t" ];
        // produco una stringa autore/titolo
        curTrack = [ NSString stringWithFormat: @"%@ - %@",
                        [trackInfo objectAtIndex: 2 ],
                        [trackInfo objectAtIndex: 3 ] ] ;
        // inserisco la voce nel menu, con una action
                       item   =   [   [NSMenuItem   alloc]   initWithTitle:
curTrack
                    action:@selector(playThatSong:)
                    keyEquivalent:@""];
        // assegno il target e il tag
        [ item setTarget: self ] ;
        [ item setTag: i ] ;
        // inserisco effettivamente il menu
        [ theMenu insertItem:item atIndex:0];
    }
    // forzo un ricarico della lista
    [ theTable reloadData ];
}
```

Uso un approccio piuttosto drastico per aggiornare il menu: cancello tutte le voci del menu, con l'eccezione delle ultime cinque, che sono quelle standard previste da *Interface Builder*. Poi, procedo ad inserire man mano una serie di voci corrispondenti ai vari elementi del vettore `listaCanzoni`. La voce del menu è composta dalle sole informazioni di autore e titolo; ad ogni menu è associata l'azione `playThatSong:` avente come target la classe `self` (ancora `LastSongCtl`). Per distinguere di quale canzone si tratta, attribuisco alle varie voci anche un `tag` (l'indice del vettore `listaCanzoni`).

E con questo, il menu è servito.

L'ultima istruzione, infine, dice alla tabella di ricaricare i dati perché ci sono state delle

modifiche. Infatti, in sede di Interface Builder, avevo collegato la tabella alla classe `LastSongCtl` come sorgente di dati e come delegata. Tra le altre cose, significa che all'interno di `LastSongCtl` ci sono i metodi

```
- (int)    numberOfRowsInTableView: (NSTableView *)tableView;
- (id)      tableView:  (NSTableView  *)tableView  objectValueForTableColumn:
(NSTableColumn *)tableColumn row: (int)row ;
```

necessari appunto alla visualizzazione dei dati: sono piuttosto semplici.

```
- (int)
numberOfRowsInTableView:    (NSTableView *)tableView
{
    return [listaCanzoni count];
}

- (id)
tableView:            (NSTableView *)tableView
    objectValueForTableColumn:    (NSTableColumn *)tableColumn
    row:             (int)row
{
    NSString * colId ;
    NSArray        * trackInfo ;
    // recupero l'identificatore della colonna
    colId = [ tableColumn identifier] ;
    // e la informazioni relative alal riga da restituire
    trackInfo = [ [ listaCanzoni objectAtIndex: row ]
        componentsSeparatedByString: @"\t" ];
    // a seconda del nome della colonna, restituisco la stringa
    if ( [ colId isEqual: @"autore" ] )
    {
        return ( [trackInfo objectAtIndex: 2 ] );
    }
    if ( [ colId isEqual: @"titolo" ] )
    {
        return ( [trackInfo objectAtIndex: 3 ] );
    }
    if ( [ colId isEqual: @"album" ] )
    {
        return ( [trackInfo objectAtIndex: 4 ] );
    }
    if ( [ colId isEqual: @"durata" ] )
    {
        // la durata e' espressa in secondi
        int min, sec ;
        sec = [ [trackInfo objectAtIndex: 5 ] intValue ] ;
        // la spezzo in minuti e secondi
        min = sec / 60 ;
        sec = sec % 60 ;
        // per mettere uno zero se ci sono pochi secondi
        if ( sec < 10 )
            return ( [NSString stringWithFormat: @"%4d:0%1d", min, sec] );
        return ( [NSString stringWithFormat: @"%4d:%2d", min, sec] );
    }
    return ( [trackInfo objectAtIndex: 2] );
}
```

Il secondo metodo è un po' lungo, ma assolutamente lineare nel suo svolgimento; dopo aver diviso per l'ennesima volta la stringa nei vari pezzi, il metodo restituisce il campo adatto alla richiesta. Fa qualche lavoro in più nel caso del tempo, in quanto iTunes

restituisce il dato sotto forma di secondi, mentre la visualizzazione sarebbe meglio farla in qualche formato più riconoscibile.

OPERAZIONI DA MENU

Attraverso il menu si svolgono diverse operazioni, come già illustrato in sede di presentazioni. Le voci standard sono presto trattate.

Per visualizzare la finestra accessoria c'è il metodo seguente:

```
- (IBAction)
showLastSongs:(id)sender
{
    [ winctl showWindow: sender ];
    [ NSApp activateIgnoringOtherApps:YES];
    [ [ winctl window ] makeKeyAndOrderFront: sender ];
}
```

La finestra, che in Interface Builder mi sono premunito di non visualizzare fin da subito, deve essere per prima cosa costruita e visualizzata. Poi va portata davanti a tutte le altre con il classico `makeKeyAndOrderFront` (lasciate al momento perdere l'istruzione di mezzo: è necessaria in un secondo tempo).

Per arrestare o riprendere l'esecuzione, si deve eseguire lo script apposito:

```
- (IBAction)
playStop:(id)sender
{
    [ playStopScript executeAndReturnError:nil] ;
    // aggiorno tutto
    [ menuUpdater fire ];
}
```

Facile anche qui, adesso che ho visto come si fa. Mi devo ricordare anche di eseguire l'aggiornamento dei dati (soprattutto se faccio ripartire l'esecuzione).

Per portare iTunes davanti o lanciarlo in esecuzione nel caso non lo sia, si usa un unico meccanismo:

```
- (IBAction)
showiTunes:(id)sender
{
    [ [ NSWorkspace sharedWorkspace ] launchApplication: @"iTunes" ];
}
```

Utilizzo la classe `NSWorkspace`, che contiene metodi di utilità per l'interazione con il sistema operativo, per lanciare in esecuzione una applicazione di dato nome.

Per uscire dall'applicazione, c'è il comando *Quit*, che esegue un brutale uscita di scena:

```
- (IBAction)
quitMenu:(id)sender
{
```

```
    [ NSApp terminate: sender ] ;
}
```

La parte più difficile (e finale) è quando l'utente seleziona una delle voci di menu corrispondenti ad una delle canzoni memorizzate: l'idea è di eseguire nuovamente la canzone selezionata. La stessa cosa faccio accadere quando l'utente fa doppio clic su una delle righe della tabella all'interno della finestra accessoria. Il metodo è il già citato `playThatSong::`

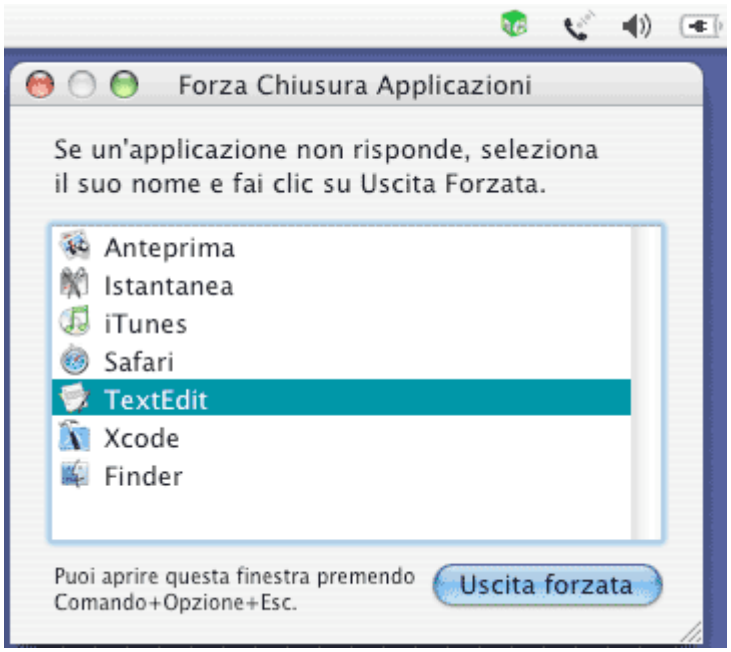
```
- (IBAction)
playThatSong:(id)sender
{
    NSArray        * trackInfo ;
    NSString    * playuotString ;
    NSAppleScript * playuotScript ;
    // per ora nessuna selezione
    int    idx = -1 ;
    // potrei aver selezionato tramite menu
    if ( [ sender isKindOfClass: [ NSMenuItem class ]] )
    {
        // ed allora l'indice la canzone la ottengo attraverso
        // il tag della voce di menu
        idx = [ sender tag ] ;
    }
    // oppure ho fatto doppio clic sulla lista
    if ( [ sender isKindOfClass: [ NSTableView class ]] )
    {
        // ed allora l'indice e' la riga selezionata
        idx = [ sender selectedRow ] ;
    }
    // giusto per paranoia
    if ( idx == -1 ) return ;
    // adesso recupero la canzone
    trackInfo = [ [ listaCanzoni objectAtIndex: idx ]
        componentsSeparatedByString: @"\t" ];
```

Per prima cosa riconosco l'origine del comando, se menu o tabella. Lo scopo delle istruzioni è di arrivare ad un buon valore per la variabile `idx`, che riporti in tutti i casi l'indice dell'elemento prescelto all'interno del vettore `listaCanzoni`. Una volta determinata la canzone, come al solito spezzo la stringa nelle sue varie componenti.

```
    // produco uno script al volo
    playuotString = [ NSString stringWithFormat:
        @"tell application \"iTunes\" to play track %d of playlist %d",
            [[trackInfo objectAtIndex: 1 ] intValue ],
            [[trackInfo objectAtIndex: 0 ] intValue ] ] ;
    // costruisco uno applescript
    playuotScript = [ [ NSAppleScript alloc ] initWithSource: playuotString
];
    // e lo eseguo
    [ playuotScript executeAndReturnError:nil ];
    // aggiorno tutto (a prescindere dal risultato)
    [menuUpdater fire];
}
```

Per suonare la canzone prescelta, utilizzo l'indice di playlist e l'indice di track per impartire un comando ad iTunes. Come al solito, devo passare attraverso uno script; questa volta, però, lo script dipende da alcune variabili: piuttosto che avere uno script

esterno, conviene produrre uno script direttamente all'interno di una stringa (la variabile `playuotString`), convertire la stringa in uno script ed eseguirlo brutalmente.



C'è infine una ultima operazione da fare, ovvero trasformare l'applicazione da una applicazione standard in una applicazione normalmente in background; così facendo, non compare nel *Dock* e nella lista delle applicazioni che si possono uccidere con la chiusura forzata. Si interviene all'interno del file `Info.plist`. Ci sono due possibilità; la prima è di aggiungere la seguente coppia chiave- valore (che confesso non ricordare da dove salti fuori):

```
    <key>NSBGOnly</key>
    <string>1</string>
```

Questo dice che l'applicazione funziona solo in background. Così facendo, però, quando si seleziona la voce *Show Last Songs*, la finestra corrispondente non c'è verso di portarla davanti a tutte le altre, ma occorre, una volta visibile, selezionarla esplicitamente.

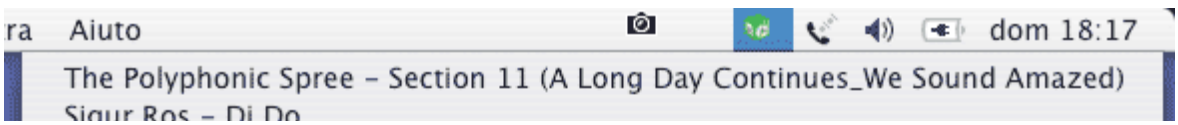
L'altra possibilità è di usare una coppia leggermente differente (nella documentazione sulle *Property List*, dopo che nell'esempio da me copiato se ne faceva uso):

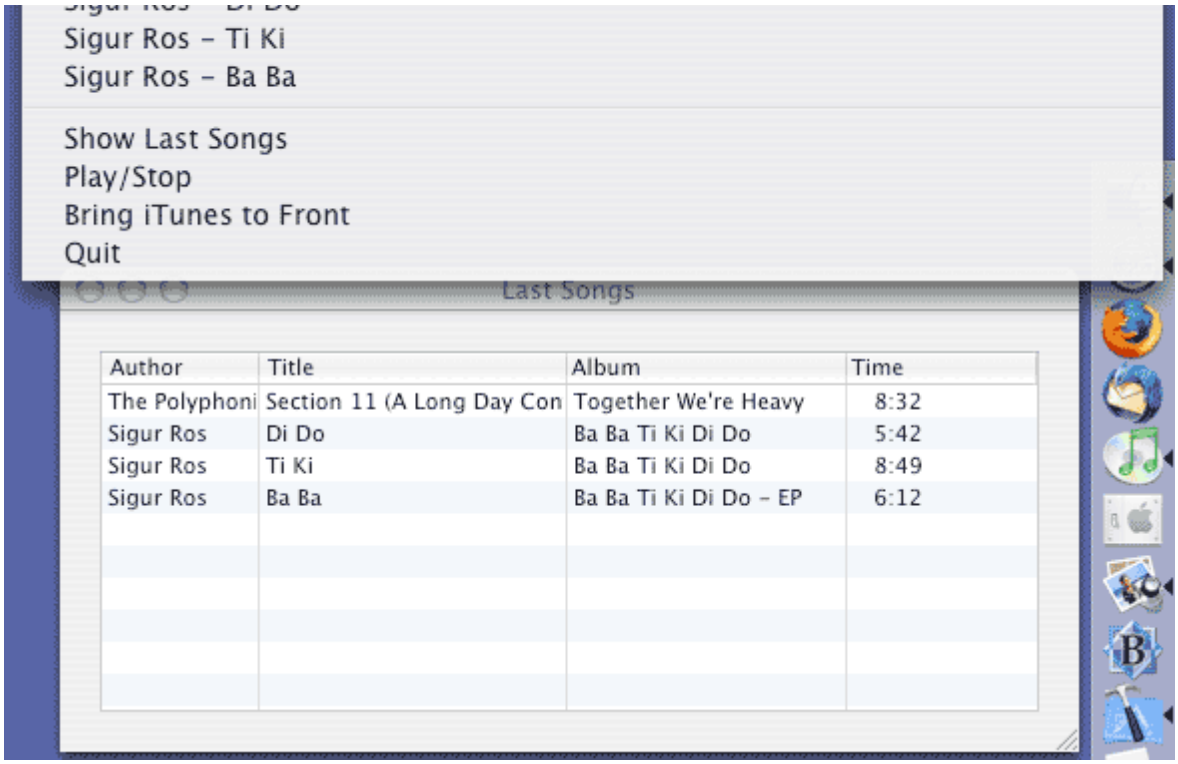
```
    <key>LSUIElement</key>
    <string>1</string>
```

il cui effetto è simile alla coppia precedente; qui però, con l'uso dell'istruzione prima non commentata

```
    [ NSApp activateIgnoringOtherApps:YES];
```

la finestra con la lista delle canzoni è portata direttamente in primo piano.





Insomma, il risultato finale, benché non del tutto professionale, con ampie possibilità di miglioramento, risponde alle mie esigenze.

Capitolo 060 - Ricerche approfondite

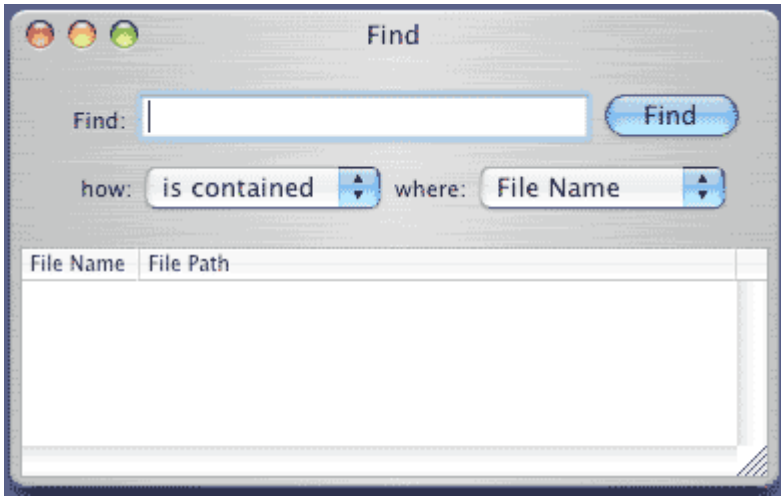
In questo capitolo riprendo in mano le funzioni di ricerca sugli elementi di un catalogo; le rendo più complete e sofisticate, e ci aggiungo qualche funzionalità. Il tutto, controllato da una finestra piuttosto esotica con alcuni punti degni di nota.

Sorgenti: nessuna, a parte la solita documentazione

Prima stesura: venerdì 24 settembre 2004

NUOVE FUNZIONALITÀ

Voglio introdurre nuove meccanismi per le funzioni di ricerca. Attualmente è infatti possibile eseguire ricerche solo basandosi sul valore di uno dei campi informativi di un elemento del catalogo.



Giusto per fare mente locale, la finestra dove inserire i parametri di ricerca presenta solamente due menu pop up con i quali scegliere il campo sul quale eseguire la ricerca e l'operatore di confronto; un terzo campo permette di inserire un testo che funziona come parametro della ricerca.

Questa strategia ha un difetto evidente ed uno meno evidente; il difetto evidente è che il *criterio di ricerca* è sempre piuttosto semplice. Il difetto meno evidente è che si confrontano sempre stringhe, e talvolta si ottengono risultati bizzarri. Poi c'è anche il fatto che la ricerca non è completa (la ricerca sulle date non funziona tanto bene).

La prima idea per migliorare le funzioni di ricerca è di introdurre più criteri di ricerca. Ogni criterio è formato da una terna *campo-operatore-valore*; più criteri possono essere combinati tra loro in *AND* (un elemento è valido se soddisfa a tutti i criteri contemporaneamente) oppure in *OR* (un elemento è valido se soddisfa ad almeno uno dei criteri).

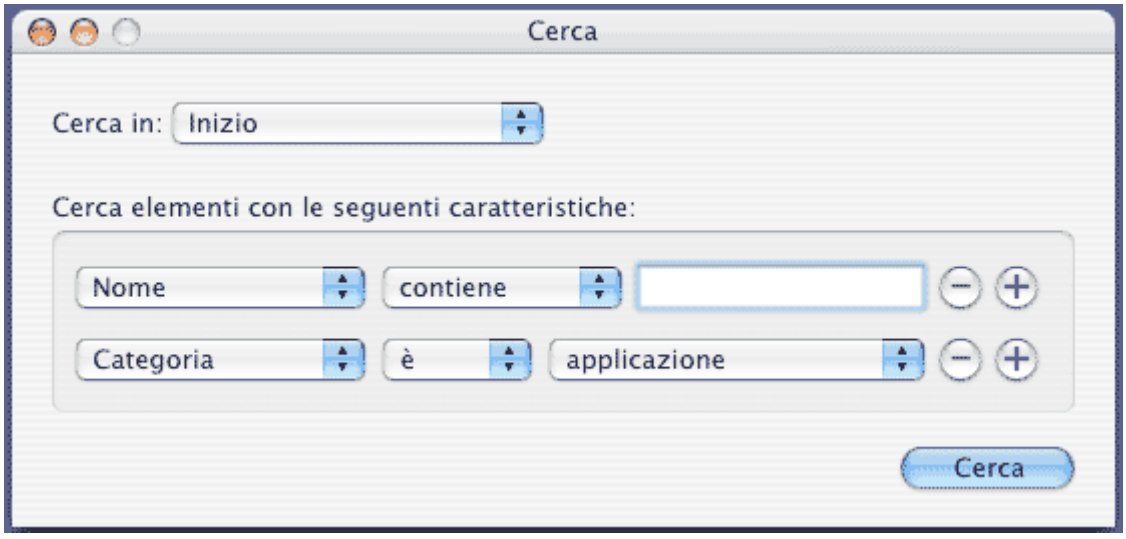
La seconda idea è di introdurre qualche criterio non basato esclusivamente sui campi

di informazione di un elemento. Ad esempio, un criterio che individui tutti i file che hanno discendenti (le cartelle) o che non ne hanno per nulla (i file semplici), oppure ancora un criterio che individui i *bundle* (applicazioni, framework, eccetera).

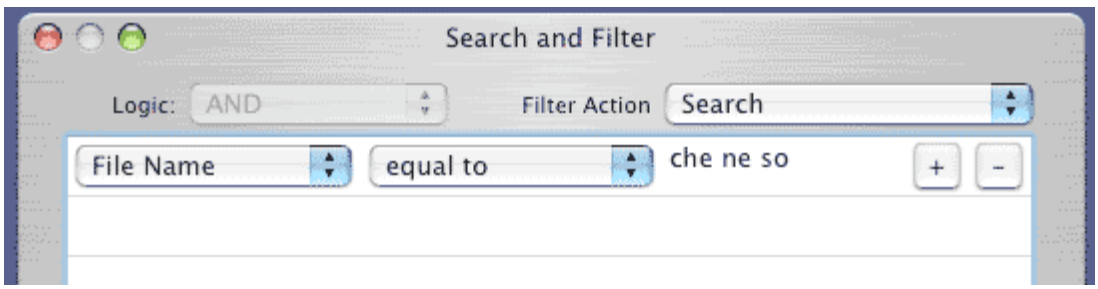
La terza idea è che le operazioni di ricerca rientrano nei più generali processi di *filtraggio*. Una procedura di ricerca su una base di dati non è altro che una operazione di filtraggio: si parte da un insieme di elementi, e di questi se ne fanno passare, se ne filtrano, una certa parte, che soddisfano a dei determinati criteri. Mostrare l'elenco risultante è l'unico effetto di questa ricerca. Una possibilità interessante che mi è venuta in mente è di effettuare delle semplici operazioni su questi elementi. Nella fattispecie, ciò che mi interessa è impostare il campo `is2Print` a *Vero* (`YES`) o *Falso* (`NO`) secondo determinati criteri. Ad esempio, catalogando CD di programmi, mi interessa nelle copertine stampare solo le applicazioni presenti; se invece ci sono CD di file MP3 ben ordinati (sorvoliamo sugli aspetti legali della faccenda), mi interessa stampare solo le cartelle, perché i file presenti sono ordinatamente raccolti per autore, all'interno di cartelle con nome acconcio.

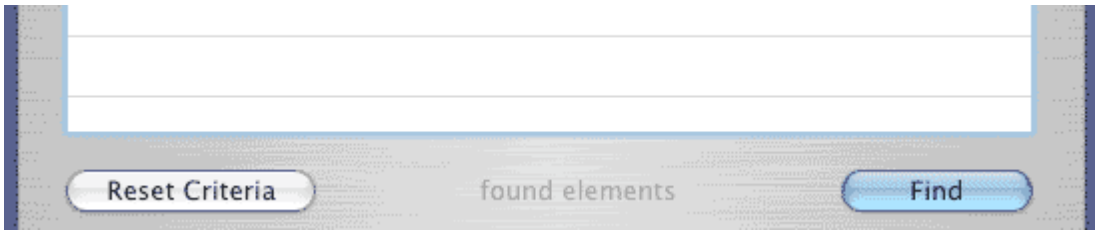
INTERFACCIA UTENTE

In base a questi grossolani requisiti, ho cominciato a progettare l'interfaccia utente. Qui ho il primo problema di come l'utente possa stabilire diversi criteri.



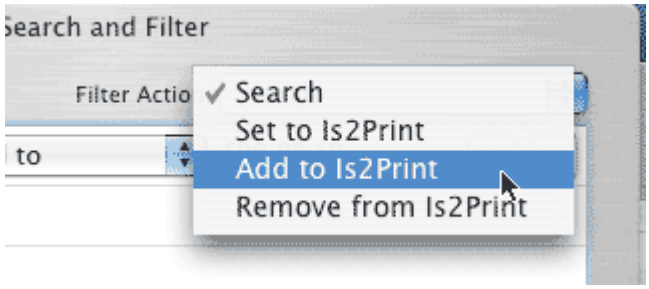
Mi sono ispirato alla finestra con la quale si svolge lo stesso compito nel Finder di Mac Os X. Senza raggiungere la bellezza e la (nascosta) complessità di quella finestra, ho pensato di raccogliere il meccanismo di specifica dei criteri all'interno di una `NSTableView`. Ogni riga della tabella specifica un criterio, mentre le varie colonne sono utilizzare per determinare meglio i parametri del criterio.





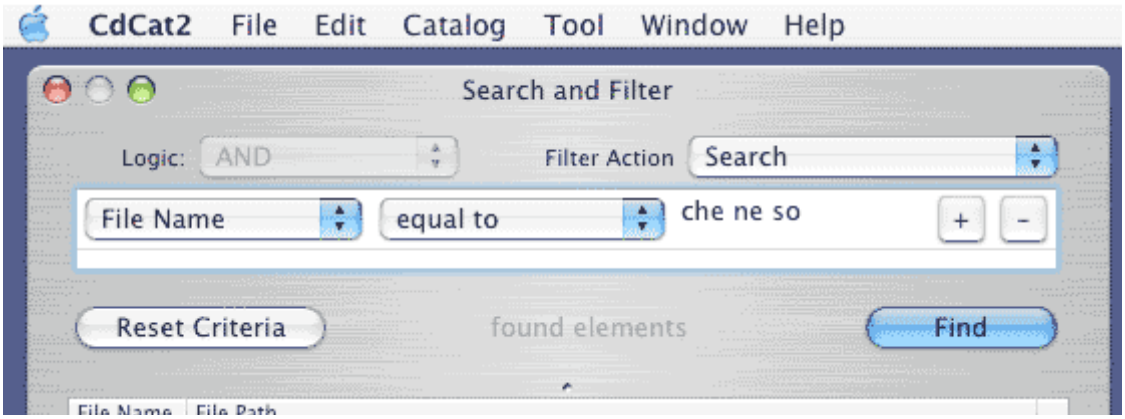
In particolare, la tabella ha cinque colonne. Nella prima colonna c'è un menu pop up con il quale specificare il campo dell'elemento del catalogo sul quale opera il criterio; la seconda colonna specifica l'operatore (maggiore, minore, uguale, eccetera) di confronto, mentre con la terza colonna si determina direttamente (tramite tastiera) il valore di confronto. La quarta e la quinta colonna le riservo per due pulsanti, in grado di aggiungere o eliminare criteri. Rispetto alla finestra proposta dal Finder, oltre alla bruttezza di quanto realizzato da me, c'è anche il fatto che il Finder presenta in alcuni casi un ulteriore menu pop up (o altro controllo speciale) come specifica del valore, mentre nel mio caso c'è sempre un campo di testo.

La finestra è completata da altri due menu pop up. Col primo menu si specifica se i criteri vanno considerati in AND o in OR (è evidente che nel caso di un solo criterio, la distinzione AND/OR cade); col secondo menu invece determino l'azione da eseguire sugli elementi trovati.



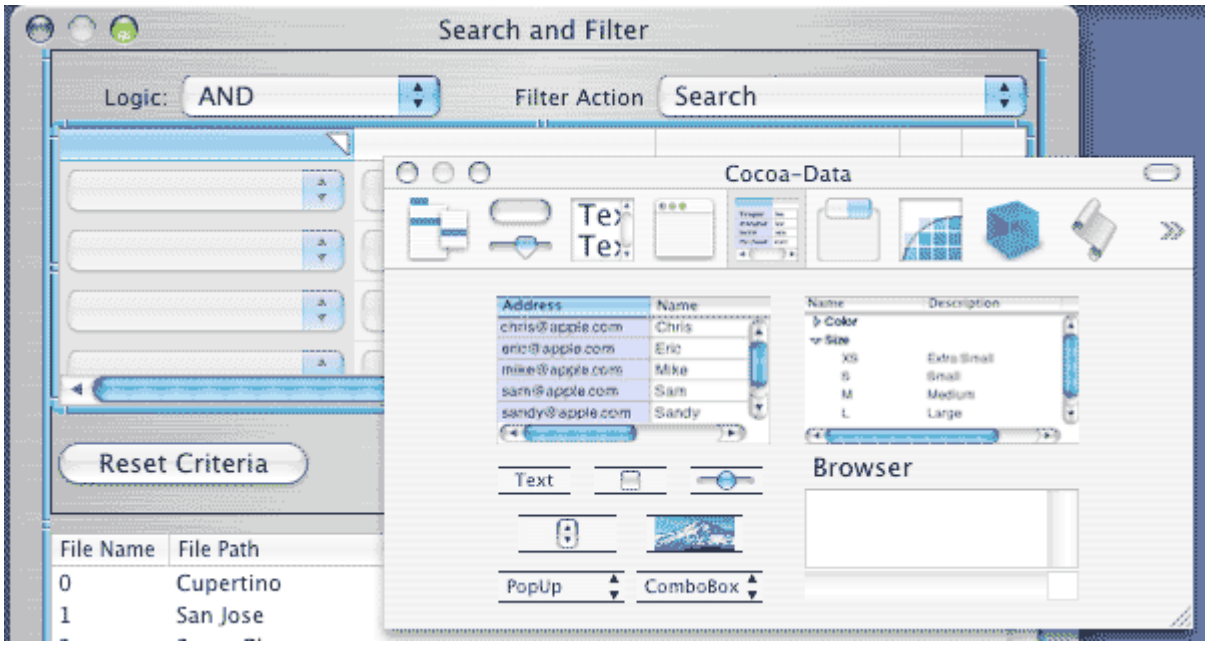
Ho previsto l'azione standard di ricerca, con la quale gli elementi sono semplicemente visualizzati in una lista; l'azione di impostazione del campo `is2Print`: il pulsante assume lo stato corrispondente al risultato della valutazione dei criteri per quell'elemento; l'azione di aggiunta agli elementi da stampare: se un elemento soddisfa ai criteri, il suo campo `is2Print` è posto a Vero; l'azione opposta di rimozione dalla lista degli elementi in stampa.

Ci sono poi due pulsanti, uno per cominciare le operazioni di ricerca e l'altro per riportare le impostazioni dei criteri ad un valore di reset.



Tutto ciò è racchiuso all'interno di un elemento `NSBox`, a suo volta parte di una `NSSplitView`. L'altra parte della `NSSplitView` è la tabella destinata a raccogliere i risultati della ricerca.

La parte più complicata dell'intera faccenda è la completa caratterizzazione della `NSTableView` destinata a contenere i criteri. Finora ho sempre utilizzato tabelle piuttosto semplici; la cosa più complicata è stato l'inserimento di una cella speciale per avere testo ed immagine (la classe `ImageAndTextCell`, per altro di terze parti), o di una cella sotto forma di pulsante di spunta (proprio la visualizzazione del campo `is2Print`). Ricordo che la visualizzazione dei dati all'interno di una tabella avviene tramite l'uso di oggetti della classe `NSCell` (o discendenti; normalmente si tratta di `NSTextFieldCell`). In entrambi questi casi per usare un tipo di cella differente ho dovuto utilizzare Cocoa ed una istruzione apposita. Ho scoperto che invece è possibile eseguire l'operazione anche in *Interface Builder*.



Si va nella palette *Cocoa-Data* e si trascina una (quella desiderata, ovviamente: NSPopUpButtonCell e `NSButtonCell`) delle piccole icone nella parte inferiore esattamente sullo header della colonna; l'operazione è segnalata dalla presenza di un triangolino in alto a destra. Selezionando il triangolino, si accede all'oggetto di tipo `NSCell`, per eventualmente manipolarne qualche parametro.

Con questa configurazione di file nib, ho avuto notevoli problemi: spesso e volentieri *Interface Builder* entrava in confusione e terminava la propria vita operativa senza salvare correttamente il file. Tuttavia, con infinita pazienza e ripetuti salvataggi, ne sono venuto fuori.

Ho così un nuovo file `FilterWin.nib` ed una nuova classe `FilterWinCtl` che vanno a rimpiazzare la precedente finestra (e la corrispondente classe) per effettuare ricerche.

I CRITERI DI RICERCA

Ho cercato di riutilizzare quanto più possibile le istruzioni già scritte in precedenza; in realtà poche cose si sono salvate, anche se l'impianto di base è rimasto lo stesso.

Per prima cosa, occorre spendere due parola sulla struttura dati destinata a contenere le specifiche dei criteri: è una buona e vecchia struttura C:

```
typedef struct    _filterElem {
    int            fieldId ;
    int            operId ;
    id            theValue ;
}
    FltElemStruct, * FltElemPtr ;
```

Il campo `fieldId` contiene un identificatore del campo (nome del file, data di creazione, eccetera); il campo `operId` contiene un identificatore che specifica il tipo di confronto (maggiore, minore, eccetera); il campo `theValue` contiene il valore con cui effettuare il confronto. Si tratta di un oggetto generico in quanto può contenere una stringa, un numero o una data.

Ho fatto in modo che un vettore di dieci di queste strutture sia compreso nelle variabili della classe `FilterWinCtl`:

```
    int                         numOfCriteri ;
    FltElemStruct            criteriFiltro[10] ;
```

Ho anche fatto in modo che i valori dei campi `fieldId` e `operId` corrispondano ai tag dei menu pop up.

C'è un po' di confusione riguardo i menu pop up: ce ne sono veramente tanti. I menu per la selezione AND/OR e per le azioni conseguenti alla ricerca sono di facile ed immediata gestione; in pratica impostano dei parametri che saranno letti al momento di effettuare la ricerca (quando si attiva il pulsante *Find*, ovviamente). Ben più complicata la gestione dei vari menu pop up all'interno della tabella. Qui c'è il menu della prima colonna, che è sempre lo stesso; il menu nella seconda tabella invece cambia a seconda della scelta effettuata nella prima colonna. Se il campo evidenziato dal primo menu è ad esempio il nome del file, il menu delle operazioni di confronto presenta le opzioni di uguale e diverso; se il campo è una data, il confronto si può fare per maggiore, minore o uguale. Del resto, lo stesso problema è già stato affrontato e risolto nella vecchia realizzazione. Qui le cose sono un po' più lunghe e noiose, perché più pignole. Ci sono diversi menu pop up relativi alle operazioni: uno per i confronti maggiore, minore, uguale, diversi, adatti per date e numeri; uno per uguale e diverso, adatti per campi come permessi di accesso, creatore, gruppo, eccetera; uno con uguale e contenuto in, per i classici confronti su stringhe.

Per far partire il meccanismo, ad ogni voce di menu è stato assegnato un diverso tag. Il menu della scelta dei campi fa riferimento al seguente metodo, lungo ma noioso:

```
- (IBAction)
popUpSelectField:(id)sender
{
    int        row = [listaCriteri selectedRow ] ;
    int        whois = [ sender tag ] ;

    if ( row >= numOfCriteri ) return ;
    switch ( whois ) {
    case MENUTAG_FILENAME:
    case MENUTAG_GROUPNAME:
    case MENUTAG_OWNERNAME:
        criteriFiltro[row].fieldId = whois ;
        criteriFiltro[row].operId = MENUTAG_STR_ISCONTAINED ;
        // devo convertire il valore in una stringa
        convert2string( & criteriFiltro[row] ) ;
        break ;
    case MENUTAG_MODDATE:
    case MENUTAG_CREATDATE :
        criteriFiltro[row].fieldId = whois ;
        criteriFiltro[row].operId = MENUTAG_NUM_EQ ;
        // devo convertire il valore in una data
        convert2date( & criteriFiltro[row] ) ;
        break ;
    case MENUTAG_POSIXPERM:
    case MENUTAG_OSCREATOR:
    case MENUTAG_OSTYPE:
    case MENUTAG_FILESIZE :
        criteriFiltro[row].fieldId = whois ;
        criteriFiltro[row].operId = MENUTAG_NUM_EQ ;
        // devo convertire il valore in un numero
        convert2number( & criteriFiltro[row] ) ;
        break ;
    case MENUTAG_SPECIAL :
        criteriFiltro[row].fieldId = whois ;
        criteriFiltro[row].operId = MENUTAG_NONE ;
        break ;
    // non dovrebbe mai succedere ...
    default :
        criteriFiltro[row].fieldId = MENUTAG_FILENAME ;
        criteriFiltro[row].operId = MENUTAG_STR_ISCONTAINED ;
        break ;
    }
    [ listaCriteri reloadData ] ;
}
```

Dopo aver recuperato la riga correntemente selezionata (che è anche l'indice del vettore dei criteri), si imposta il campo `fieldId` con il tag ricevuto; si predispone anche il campo `operId` perché visualizzi una operazione sensata sul tipo di dato. Il valore di confronto poi è convertito dal valore corrente ad un valore più adatto. Se ad esempio si passa da una stringa ad una data, è chiamata la funzione `convert2date`:

```
void
convert2date( FltElemPtr critPtr )
{
    if ( [ critPtr->theValue isKindOfClass:[NSString class] ] )
    {
        NSDate    * xx = [ NSDate dateWithString: critPtr->theValue ] ;
        [ critPtr->theValue release ];
        critPtr->theValue = [ xx copy ];
        [ critPtr->theValue retain ];
    }
```

```
    else if ( [ critPtr->theValue isKindOfClass:[NSNumber class] ] )
    {
        NSString    * xx = [ NSDate dateWithTimeIntervalSinceNow:
                [ critPtr->theValue floatValue] ] ;
        [ critPtr->theValue release ];
        critPtr->theValue = [ xx copy ];
        [ critPtr->theValue retain ];
    }
}
```

Così facendo, si evitano spiacevoli inconvenienti quando si cercano di confrontare mele (date) con pere (stringhe). Lo steso dicasi per gli altri casi.

Del tutto simile, ma molto più semplice, il metodo associato a tutti gli elementi dei menu pop up relativi alle operazioni:

```
- (IBAction)
popUpSelectOper:(id)sender
{
    int         row = [listaCriteri selectedRow ] ;
    int         whois = [ sender tag ] ;

    if ( row >= numOfCriteri ) return ;
    criteriFiltro[row].operId = whois ;
    [ listaCriteri reloadData ] ;
}
```

VISUALIZZARE I CRITERI

Per visualizzare i criteri, occorre scrivere i due classici metodi di una sorgente di dati:

```
- (int)    numberOfRowsInTableView:    (NSTableView *)tableView;
-       (id)                          tableView:       (NSTableView       *)tv
objectValueForTableColumn:    (NSTableColumn *)tc row: (int)row ;
```

Qui ci sono una pio di complicazioni semplici: la prima riguarda il fatto che ci sono due tabelle cui fornire dati, e la classe `FilterWinCtl` funziona da dataSource e delegata per entrambe. La seconda è che non si possono fornire i valori ai menu pop up. Il primo problema si risolve fornendo ogni tabella di un tag, e verificando quale tabella è interessata dalla fornitura dati; oltretutto, per fornire i dati alla tabella dei risultati, utilizzo pari pari lo stesso meccanismo della vecchia realizzazione (lo array `foundItems`). Il secondo problema è stato risolto con il metodo (delegato)

```
-     (void)     tableView:(NSTableView     *)tv     willDisplayCell:(id)ac
forTableColumn:(NSTableColumn *)tc row:(int)rowIndex ;
```

A questo punto, il numero di elementi presenti è semplice:

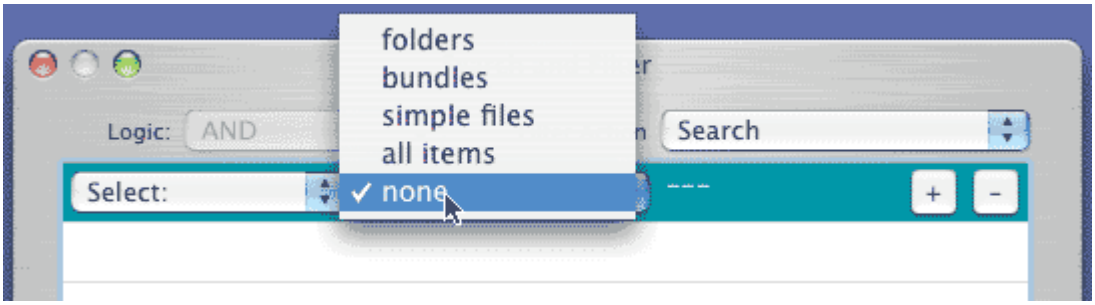
```
- (int)
numberOfRowsInTableView:    (NSTableView *)tableView
{
    if ( [ tableView tag ] == TABLEVIEWTAG_CRITERI )
            return (numOfCriteri);
    else    return [foundItems count];
```

```
}
```

Anche per fornire i valori alle singole celle ci vuole poco:

```
- (id)
tableView:            (NSTableView *)tableView
    objectValueForTableColumn:    (NSTableColumn *)tableColumn
    row:             (int)row
{
    if ( [ tableView tag ] == TABLEVIEWTAG_CRITERI )
    {
        // sono nella tabella criteri
        NSString * colId = [ tableColumn identifier] ;
        // l'unico valore sensato da restituire sta qui
        if ( [ colId isEqual: @"valueSpec" ] )
        {
            // se il campo specificato e' speciale
            if ( criteriFiltro[row].fieldId == MENUTAG_SPECIAL )
                return ( @" " );
            // altrimenti, restiuisco il valore nudo
            return ( criteriFiltro[row].theValue );
        }
        // giusto per ritornare qualcosa sempre
        return ( @"dummy" );
    }
    else
    {
        // sono nella tabella risultati
        CatFileInfo    * item = [foundItems objectAtIndex: row] ;
        NSString    * colId = [ tableColumn identifier] ;
        // ho al momento due colonne: il nome o l'intero percorso
        if ( [ colId isEqual: COLID_FILENAME ] )
            return ( [ item fileName] );
        return ( [ item fileFullPath] );
    }
}
```

È arrivato il momento di parlare della voce di menu *Select:* cui corrisponde il tag `MENUTAG_SPECIAL`.



Con questo menu introduco criteri non legati alla scelta campo-operazione-valore; i criteri introdotti riguardano la selezione di cartelle (o meglio, di file con qualche discendente), di bundle (applicazioni, framework, eccetera, cartelle caratterizzate dall'avere una data estesione (per riconoscere un bundle ho scritto una funzione apposta), di semplici file (che non hanno discendenti), la selezione di tutto ed anche di nessun elemento (poco utili nelle ricerche, ma più adatti alle impostazioni per caratterizzare la stampa).

Caratteristica di questo tipo di criteri è di non avere un valore di confronto, per cui la terza colonna della tabella non è utile; la riempio comunque con dei caratteri non significativi.

Torniamo alla visualizzazione della tabella dei criteri; devo ancora fare il grosso del lavoro, ovvero aggiustare la visualizzazione dei menu pop up; il metodo è piuttosto lungo, ma con molte parti poco interessanti; però in fondo c'è qualcosa di interessante.

Ricordo che il metodo è chiamato per ogni cella prima che questa sia disegnata; è utilizzato appunto per effettuare predisposizioni dell'ultimo momento:

```
- (void)
tableView:(NSTableView *)aTableView
    willDisplayCell:(id)aCell
    forTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    if ( [ aTableView tag ] == TABLEVIEWTAG_CRITERI )
    {
        // sono nella tabella criteri
        NSString * colId = [ aTableColumn identifier] ;
        // menu pop up della scelta campo
        if ( [ colId isEqual: @"fieldSpec" ] )
        {
            // ricavo quale item e' da evidenziare
                  int      qualemenu  =  (criteriFiltro[rowIndex].fieldId  -
MENUTAG_FILENAME) ;
            [ aCell selectItemAtIndex: qualemenu ] ;
        }
        // menu pop up delle operazione di confronto
        if ( [ colId isEqual: @"operSpec" ] )
        {
            // tag del menu da visualizzare
            // per avere l'indice, dovro' sottrarre il tag base
            int    newCode = criteriFiltro[rowIndex].operId ;
            // imposto menu e item selezionato in dipendenza
            // dalla scelta del campo
            switch ( criteriFiltro[rowIndex].fieldId ) {
            case MENUTAG_FILENAME:         // stringhe
            case MENUTAG_GROUPNAME:         // cfr: uguale
            case MENUTAG_OWNERNAME:         //    contenuto
                [ aCell setMenu: strMenu ] ;
                newCode -= MENUTAG_STR_ISCONTAINED ;
                break ;
            case MENUTAG_MODDATE:          // cfr: eguale, diverso
            case MENUTAG_CREATDATE:         //    maggiore, minore
            case MENUTAG_FILESIZE:
                [ aCell setMenu: operMenu ] ;
                newCode -= MENUTAG_NUM_EQ ;
                break ;
            case MENUTAG_OSTYPE:          // cfr: uguale
            case MENUTAG_POSIXPERM:     //    diverso
            case MENUTAG_OSCREATOR:
                [ aCell setMenu: compMenu ] ;
                newCode -= MENUTAG_NUM_EQ ;
                break ;
            case MENUTAG_SPECIAL :         // crf, nessuno
                [ aCell setMenu: typeMenu ] ;
                newCode -= MENUTAG_FOLDER ;
                break ;
            }
```

```
            [ aCell selectItemAtIndex: newCode ] ;
        }
```

Fino a qui non ci sono cose particolari degne di nota. Il trucco di tutto il meccanismo sta nell'avere associato ad ogni item dei menu tag con valori consecutivi; per passare dal tag all'indice dell'item all'interno del menu è quindi sufficiente sottrarre al tag il valore del tag del primo item del menu. Inoltre, a seconda del tipo di menu, alla cella è associato il menu corrispondente (`strMenu`, `operMenu`, eccetera, sono tutti outlet a menu predefiniti all'interno del file nib).

Adesso arriva il bello: il problema con la visualizzazione del campo dei valori sta nel fatto che uno stesso spazio è occupato da valori di natura diversa: stringhe, numeri, date, eccetera. Per aiutare l'utente a vedere questi dati, Cocoa introduce il concetto di Formatter. Ecco quindi che anche per queste celle assegno un formattatore adatto al tipo di dati da rappresentare:

```
        // aggiusto un formatter per visualizzare il dato
        // nella forma piu' acconcia
        if ( [ colId isEqual: @"valueSpec" ] )
        {
            TOS9TCForm        *myDF1 ;
            FileSizeForm     *myDF2 ;
            FilePosixPerm     *myDF3 ;
            NSDateFormatter *myDF4 ;

            switch ( criteriFiltro[rowIndex].fieldId ) {
            case MENUTAG_FILENAME:        // stringhe
            case MENUTAG_SPECIAL :
            case MENUTAG_GROUPNAME:
            case MENUTAG_OWNERNAME:
                [ aCell setFormatter: nil ];
                break ;
            case MENUTAG_OSTYPE:         // os type
            case MENUTAG_OSCREATOR:
                myDF1 = [[[ TOS9TCForm alloc ] init ] autorelease ];
                [ aCell setFormatter: myDF1 ];
                break ;
            case MENUTAG_FILESIZE:        // dim file
                myDF2 = [[[ FileSizeForm alloc ] init ] autorelease ];
                [ aCell setFormatter: myDF2 ];
                break ;
            case MENUTAG_POSIXPERM:        // posix perm
                myDF3 = [[[ FilePosixPerm alloc ] init ] autorelease ];
                [ aCell setFormatter: myDF3 ];
                break ;
            case MENUTAG_MODDATE:         // date
            case MENUTAG_CREATDATE:
                myDF4 = [ [NSDateFormatter alloc]
                    initWithDateFormat: @"%d %b %Y"
                    allowNaturalLanguage: YES];
                [ aCell setFormatter: myDF4 ];
                break ;
            }
        }
    }
    // niente da fare per la tabella risultati
}
```

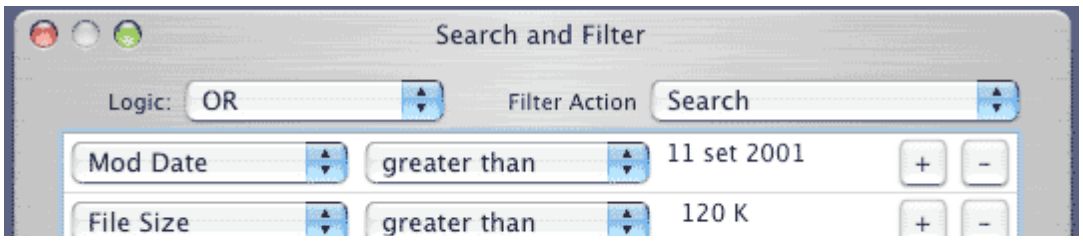
Prima di continuare, occorre fermarsi un momento su questi formattatori. Quando, a

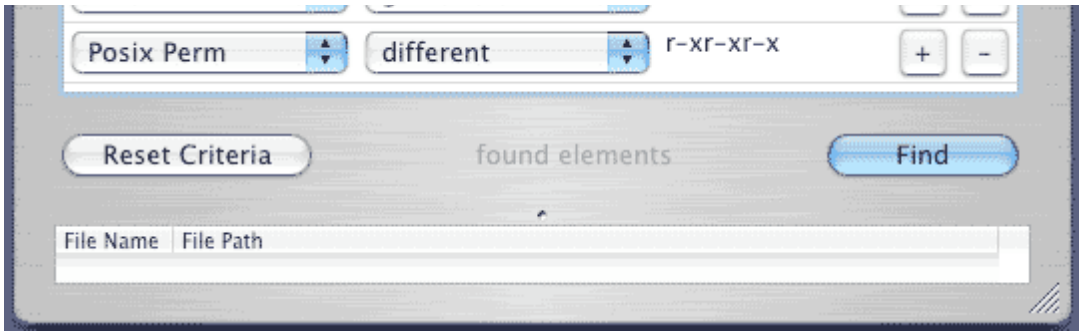
suo tempo, realizzai qualche classe di formattatori, mi ero limitato alla parte di visualizzazione (ovvero, come convertire un valore in una rappresentazione sotto forma di stringa). Ora, occorre anche il processo inverso, ovvero, partire da una rappresentazione sotto forma di stringa per ottenere un dato valore. Se questo processo è di serie con i formattatori standard (date, numeri), non lo è per i formattatori specifici della mia applicazione.

In effetti, ad esempio, nella realizzazione della classe `FileSizeForm`, ci sono sì due metodi alla bisogna, ma erano vuoti; occorre almeno scrivere il primo, che dovrebbe produrre un valore per l'oggetto di un dato tipo a partire da una stringa che lo rappresenta:

```
- (BOOL)
getObjectValue:             (id *)anObject
    forString:          (NSString *)string
    errorDescription:     (NSString **)error
{
    NSRange     rr ;
    float     xx, scale ;
    // dalla stringa estraggo l'eventuale numero presente
    xx = [ string floatValue ] ;
    // cerco di ricavare l'unita' di misura guardando se ci sono
    // sottostringhe di soli caratteri
          rr   =   [   string   rangeOfCharacterFromSet:   [   NSCharacterSet
letterCharacterSet ]
        options: NSCaseInsensitiveSearch ] ;
    // se non trovo sottostringhe letterali, dco niente unita'
    scale = 1 ;
    // se invece ho trovato qualcosa, calcolo la scala
    if ( rr.location != NSNotFound )
    {
        // vedo che razza di carattere potrei avere
        unichar cc = [ string characterAtIndex: rr.location ] ;
        // tra quelli sensati (byte, Kappa, Mega, Giga)
        if ( cc == 'b' )
            scale = 1 ;
        else if ( cc == 'K' )
            scale = 1024 ;
        else if ( cc == 'M' )
            scale = 1024 * 1024 ;
        else if ( cc == 'G' )
            scale = 1024 *1024 * 1024 ;
    }
    // calcaolo la dimensione in byte e costruisco un NSNumber
    * anObject = [ NSNumber numberWithLong: (long)(xx * scale) ] ;
    return ( YES );
}
```

Le operazioni qui svolte sono piuttosto grossolane e intolleranti verso l'errore da parte dell'utente; però così facendo si possono inserire nella terza colonna valori come *100 K* e *2 G* per specificare dimensioni di file.





Operazioni concettualmente similari sono svolte per le classi `TOS9TCForm` e `FilePosixPerm`.

Ancora due concetti: permettere la modifica della cella di testo per impostare il valore solo quando necessario, con il metodo seguente (delegato):

```
- (BOOL)
tableView: (NSTableView *)aTableView
    shouldEditTableColumn:    (NSTableColumn *)aTableColumn
    row: (int)rowIndex
{
    if ( [ aTableView tag ] == TABLEVIEWTAG_CRITERI )
    {
        NSString * colId = [ aTableColumn identifier] ;
        if ( [ colId isEqual: @"valueSpec" ] )
        {
            if ( criteriFiltro[rowIndex].fieldId == MENUTAG_SPECIAL )
                    return NO ;
        }
        return YES ;
    }
    else return NO ;
}
```

Se la tabella è quella dei criteri, normalmente si risponde che tutto si può editare; a meno che non si tratti della colonna in cui specificare il valore di confronto, e il criterio è speciale.

Corrispondentemente, c'è il metodo per assegnare i valori nella tabella alla struttura dati:

```
- (void)
tableView:(NSTableView *)aTableView
    setObjectValue:(id)anObject
    forTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
    if ( [ aTableView tag ] == TABLEVIEWTAG_CRITERI )
    {
        NSString * colId = [ aTableColumn identifier] ;
        if ( [ colId isEqual: @"valueSpec" ] )
        {
            [ criteriFiltro[rowIndex].theValue release ] ;
            criteriFiltro[rowIndex].theValue = anObject ;
            [ criteriFiltro[rowIndex].theValue retain ] ;
        }
    }
```

```
}
```

GESTIRE I CRITERI

All'inizializzazione della classe `FilterWinCtl` occorre predisporre la struttura dati dei criteri:

```
- (void)
resetCriteriFiltro
{
    numOfCriteri = 1 ;
    criteriFiltro[0].fieldId = MENUTAG_FILENAME ;
    criteriFiltro[0].operId = MENUTAG_STR_ISEQUAL ;
    criteriFiltro[0].theValue = [ NSString stringWithString: @"che ne so" ]
;
    [ criteriFiltro[0].theValue retain ] ;
}
```

In questo modo, all'apertura della finestra, si presenta un criterio di default (ricerca sul campo con il nome del file, per stringhe uguali). Questa è anche la situazione che si presenta quando si fa clic sul pulsante di *Clear All*.

Quando l'utente vuole aggiungere un criterio, fa clic sul pulsante presente nella quarta colonna di uno criteri presenti; questo provoca l'esecuzione del seguente metodo:

```
- (IBAction)
addAnotherCrit:(id)sender
{
    if ( numOfCriteri >= 10 ) return ;
    criteriFiltro[numOfCriteri].fieldId = MENUTAG_FILENAME ;
    criteriFiltro[numOfCriteri].operId = MENUTAG_STR_ISCONTAINED ;
    criteriFiltro[numOfCriteri].theValue =
        [ NSString stringWithString: @"che ne so" ] ;
    [ criteriFiltro[numOfCriteri].theValue retain ] ;
    numOfCriteri += 1 ;
    [ logicPop setEnabled: YES ] ;
    [ listaCriteri reloadData ] ;
}
```

Le operazioni sono semplici: si aggiunge un criterio, con valori più o meno a caso. C'è da abilitare il menu pop up AND/OR, che inizialmente è disabilitato (perché di default c'è un solo criterio presente).

Un po' più complicato, ma di poco, il processo di eliminazione di un criterio, che si ha quando l'utente fa clic sul pulsante presente nella quinta colonna della tabella.

```
- (IBAction)
removeCrit:(id)sender
{
    int         row = [listaCriteri selectedRow ] ;
    int         i ;

    if ( numOfCriteri <= 1 ) return ;
    for ( i = row ; i < numOfCriteri-1 ; i ++ )
    {
        criteriFiltro[i].fieldId = criteriFiltro[i+1].fieldId ;
```

```
        criteriFiltro[i].operId = criteriFiltro[i+1].operId ;
        if (criteriFiltro[i].theValue != criteriFiltro[i+1].theValue)
        {
            [ criteriFiltro[i].theValue release ];
            criteriFiltro[i].theValue =
                [ criteriFiltro[i+1].theValue copy ];
            [ criteriFiltro[i].theValue retain ];
        }
    }
    [ criteriFiltro[numOfCriteri-1].theValue release ];
    numOfCriteri -=1 ;
    if ( numOfCriteri <= 1 )
        [ logicPop setEnabled: NO ] ;
    [ listaCriteri reloadData ] ;
}
```

Bisogna gestire il vettore, spostando indietro tutti i valori presenti di un passo, a partire dall'elemento da cancellare; in fondo, se i criteri si sono ridotti ad uno solo, disabilito nuovamente il menu pop up per AND/OR.

LA NUOVA RICERCA

A questo punto non rimane altro che scrivere il metodo che effettua la ricerca vera e propria. Si tratta sostanzialmente della vecchia procedura, rivista per tenere conto della presenza di più criteri. Cambiano i metodi nel nome e negli argomenti, ma il concetto è lo stesso. Si parte dal metodo attivato quando si fa clic sul pulsante *Find*:

```
- (IBAction)
filterElem:(id)sender
{
    int        i ;
    // recupero il catalogo dove effettuare la ricerca
    // piglio tutti i documenti dell'applicazione, e scelgo quello che sta
    // davanti a tutti gli altri... non che questo meccanismo mi piaccia...
        [  self  setCurrCatalog:  (CdCatDoc*)[[  NSApp  orderedDocuments]
objectAtIndex: 0] ];
    // pulisco i risultati della ricerca precedente
    [ self setFoundItems: [ NSMutableArray array ]];
    // e li tolgo dalla finestra di ricerca
    [ resultFld reloadData ];
    [ resultMsg setStringValue: [ NSString
        stringWithFormat: @"Found %d elements", [ foundItems count] ]];
    // ripulisco criteriFiltro mettendo una guardia a fine vettore
    for ( i = numOfCriteri ; i < 10 ; i++ )
        criteriFiltro[i].fieldId = -1 ;
    [ [ [ self currCatalog ] dataSource ] critSelect: criteriFiltro
                withLogic: [ [ logicPop selectedItem ] tag ]
                oper:        [ [ operPop selectedItem ] tag ]
                placeIn:    foundItems];
        // forzo il rinfresco della finestra dei risultati
    [ resultFld reloadData ];
    [ resultMsg setStringValue: [ NSString
        stringWithFormat: @"Found %d elements", [ foundItems count] ] ];
}
```

Se confrontate il vecchio metodo, le istruzioni sono le stesse: c'è solo una predisposizione del vettore `criteriFiltro` e il nuovo metodo di ricerca `critSelect:...`

Questo si riferisce alla classe `CatDataSrc` ed ancora una volta si limita a spazzare i volumi presenti passando gli argomenti:

```
- (void)
critSelect: (FltElemPtr) fltCrit withLogic: (int) logic
        oper: (int) operaz placeIn: (NSMutableArray *) where
{
    // piglio un elenco di tutti i volumi presenti
    NSEnumerator     * enumerator = [[self startPoint] objectEnumerator];
    FileStruct    * item ;
    // li esamino uno ad uno
    while (item = [enumerator nextObject])
    {
        // e faccio una ricerca in profondita' sugli elementi contenuti
          [ item critSelect: fltCrit withLogic: logic oper: operaz placeIn:
where ];
    }
}
```

Come sempre, il lavoro sporco è svolto all'interno della classe `FileStruct`.

```
- (void)
critSelect: (FltElemPtr) fltCrit withLogic: (int) logic
        oper: (int) operaz placeIn: (NSMutableArray *) where
{
    FileStruct    * item ;
    NSEnumerator     * enumerator = [[self fileList] objectEnumerator];
    // in ogni caso, vedo se l'elemento corrente va bene
    BOOL    isok = checkCriterio( self, fltCrit, logic );
    // intanto inserisco l'elemento filtrato nel risultato
    // se la modalita' e' ricerca, ho anche finito
    if ( isok )
        [ where addObject: self ] ;
```

Fino a qui, a parte la differente funzione invocata per capire se l'elemento risponde ai criteri impostati, non c'è nulla di nuovo. Qui inserisco alcune istruzioni per gestire i casi relativi all'impostazione delle variabili `is2Print`:

```
    // se sono su filtro stampa, distinguo i casi
    if ( operaz == FILTER_OPER_SETPRINT )
    {
        // assegno lo stato secondo valutazione
        [ self setIs2Print: isok ] ;
    }
    if ( operaz == FILTER_OPER_ADDPRINT )
    {
        // aggiungo elementi alla stampa
        if ( isok || is2Print )
            [ self setIs2Print: YES ] ;
    }
    if ( operaz == FILTER_OPER_REMPRINT )
    {
        // rimuovi gli elmenti dalal stampa
        if ( isok && is2Print )
            [ self setIs2Print: NO ] ;
    }
    // esploro i figli
    while (item = [enumerator nextObject])
    {
```

```
          [ item critSelect: fltCrit withLogic: logic oper: operaz placeIn:
where ];
    }
}
```

Come si vede, si fa prima a leggere le istruzioni che a spiegare: se l'operazione scelta è di impostare il parametro `is2Print`, si esegue il semplice metodo setIs2Print. Se l'operazione è di aggiungere gli elementi individuati tramite la ricerca all'insieme degli elementi che saranno stampati, si esegue l'operazione logica di OR tra il valore corrente di `is2Print` e il risultato della selezione; se infine l'operazione è di togliere l'elemento dalla stampa, si esegue lo AND logica tra il valore corrente ed il risultato della selezione. Per convincervi del funzionamento delle operazione, non c'è altro che farvi da soli qualche esempio.

SCEGLIERE CON CRITERIO

Rimane da esaminare la funzione `checkCriterio;` questa esegue sull'oggetto passato come primo argomento la valutazione dei criteri (secondo argomento) adoperando la logica AND/OR specificata nel terzo argomento.

La funzione è piuttosto lunga, ma si divide facilmente in diversi blocchi, che si ripetono nel concetto con lievi differenze. Tuttavia, prima di cominciare, alcune nozioni di logica.

Se la logica è AND, un elemento si considera trovato (e la funzione restituisce YES) se tutti i criteri sono soddisfatti. Non appena uno dei criteri risulti NO, è inutile continuare: sono sicuro che l'elemento non fa parte del gruppo. Quindi, per lavorare in logica AND, si parte da un valore YES, e si continua a valutare i criteri finché uno risponde NO (ed allora restituisco NO) oppure sono finiti (ed allora rispondo YES).

Se la logica è OR, un elemento si considera trovato (e la funzione restituisce YES) se almeno un criterio è soddisfatti. Non appena uno dei criteri risulta YES, è inutile continuare: sono sicuro che l'elemento fa parte del gruppo. Quindi, per lavorare in logica OR, si parte da un valore NO, e si continua a valutare i criteri finché uno risponde YES (ed allora restituisco YES) oppure sono finiti (ed allora rispondo NO).

Come si può vedere, i due paragrafi precedenti, pur differenti, sono molto simili tra loro (un matematico direbbe che sono *duali*). Per questo userò lo stesso meccanismo di lavorazione nelle due logiche, nascondendo dentro funzioni apposta il fatto che la logica sia AND piuttosto che OR:

```
BOOL
initWithLogic ( int logic )
{
    if ( logic == FILTER_LOGIC_AND )
        return ( YES ) ;
    return ( NO );
}

BOOL
setWithLogic ( BOOL val1, BOOL val2, int logic )
{
    if ( logic == FILTER_LOGIC_AND )
```

```
        return ( val1 && val2 ) ;
    return ( val1 || val2 );
}

BOOL
stopWithLogic( BOOL val, int logic )
{
    if ( logic == FILTER_LOGIC_AND )
        return ( val == NO ) ;
    return ( val == YES );
}
```

Con queste tre funzioni ausiliarie, la funzione principale che decide sulla ricerca diventa:

```
BOOL
checkCriterio( FileStruct * item, FltElemPtr fltCrit, int logic )
{
    NSComparisonResult    xxx ;
    NSRange                xx ;
    NSString            *s1 ;
    NSNumber            * n1 ;
    long long            tmp1, tmp2 ;
    long                tmp3, tmp4 ;
    int                    i ;
    BOOL                tmpVal ;
    NSArray                * id_string = [NSArray arrayWithObjects:
        COLID_FILENAME, COLID_MODDATE, COLID_CREATDATE, COLID_FILESIZE,
        COLID_GROUPNAME, COLID_OWNERNAME, COLID_POSIXPERM,
        COLID_OSCREATOR, COLID_OSTYPE,nil ];
    // inizializzo il risultato con valore di partenza
    BOOL                select = initWithLogic( logic) ;
    // esamino ordinatamente tutti i criteri
    for ( i = 0 ; i < 10 ; i++ )
    {
        // sentinella
        if ( fltCrit[i].fieldId == -1 )
            break ;
```

Ricordate? Prima di scatenare le operazioni di ricerca avevo impostato qeuesto valore `-1` per indicare che non ci sono più criteri; con l'istruzione `break` esco dal ciclo `for`.

Ora cominciano diversi blocchi di operazioni, a seconda del tipo di confronti possibili; non li mostro tutti, ma su alcuni tiro via.

```
        // le operazioni dipendono dal campo prescelto
        switch ( fltCrit[i].fieldId ) {
        // - - - - - - - - - - - - - - - - - - - -
        case MENUTAG_FILENAME:            // sono tutte stringhe
        case MENUTAG_GROUPNAME :        // cfr: uguale e diverso
        case MENUTAG_OWNERNAME :
            // ricavo la striga
                    s1 = [ item valueForKey: [ id_string objectAtIndex:
fltCrit[i].fieldId] ];
            // a seconda delle opzioni, faccio il confronto
            switch ( fltCrit[i].operId ) {
            case MENUTAG_STR_ISEQUAL :
                tmpVal = [ s1 isEqualToString: fltCrit[i].theValue ] ;
                // confronto esatto tra stringhe
                break ;
```

```
            case MENUTAG_STR_ISCONTAINED :
                xx = [ s1 rangeOfString: fltCrit[i].theValue ];
                tmpVal = (xx.location != NSNotFound) ;
                break;
            default:    // non succede, ma in caso ...
                tmpVal = NO ;
            }
            select = setWithLogic( select, tmpVal, logic );
            break ;
```

Qui sto considerando un campo stringa. Alla variabile `select` assegno, con le opportune considerazioni, il risultato del confronto.

```
        case MENUTAG_POSIXPERM:             // sono numeri, ma con strana
        case MENUTAG_OSCREATOR :          // rappresentazione;
        case MENUTAG_OSTYPE :              // cfr: uguale e diverso
            ...
            break ;
        case MENUTAG_MODDATE :          // una data, confronto pieno
            ...
            break;
        case MENUTAG_CREATDATE :          // una data, confronto pieno
            xxx = ( [ [ item creatDate] compare: fltCrit[i].theValue] ) ;
            switch ( fltCrit[i].operId ) {
            case MENUTAG_NUM_EQ :
                tmpVal = (xxx == NSOrderedSame) ;
                break;
            case MENUTAG_NUM_DIFFERENT :
                tmpVal = (xxx != NSOrderedSame) ;
                break;
            case MENUTAG_NUM_GREAT :
                tmpVal = (xxx == NSOrderedDescending) ;
                break;
            case MENUTAG_NUM_LESS :
                tmpVal = (xxx == NSOrderedAscending) ;
                break;
            default:    // non succede, ma in caso ...
                tmpVal = NO ;
            }
            select = setWithLogic( select, tmpVal, logic );
            break;
        case MENUTAG_FILESIZE :          // un numero, confronto pieno
            ...
            break ;
```

Nel caso di un confronto di date, uso il metodo apposito, ed in base al criterio produco un corretto risultato per la variabile `tmpVal`, che poi genera ancora un valore per la variabile `select`.

```
        case MENUTAG_SPECIAL :          // confronti sparsi...
            switch ( fltCrit[i].operId ) {
            case MENUTAG_FOLDER :    // figli > 0
                tmpVal = [ item numOfFiles]> 0 ;
                break ;
            case MENUTAG_BUNDLE :    // funzione apposta
                tmpVal = checkIfBundleDirectory( [item fileName]) ;
                break ;
            case MENUTAG_SIMPLEFILE :    // figlio = 0
                tmpVal = [ item numOfFiles] == 0 ;
                break ;
```

```
            case MENUTAG_ALLFILES :     // sempre YES
                tmpVal = YES ;
                break ;
            case MENUTAG_NONE :         // sempre NO
                tmpVal = NO ;
                break ;
            default:    // non succede, ma in caso ...
                tmpVal = NO ;
                break ;
            }
            select = setWithLogic( select, tmpVal, logic );
            break;
```

Nei confronti di tipo speciale, si produce velocemente il risultato in `tmpVal`, e di qui, ancora una volta, il valore `select`.

Finalmente siamo arrivati alla fine:

```
        default:        // non succede, ma in caso ...
            select = NO ;
            break ;
        }
        // verifico se vale la pena continuare
        if ( stopWithLogic( select, logic ) )
            break ;
    }
    // restituisco la valutazione di tutti i criteri
    return ( select );
}
```

La funzione `stopWithLogic(.)` controlla se si deve continuare col prossimo criterio o risparmiare operazioni inutile in quanto il risultato è già stabilito. L'ultima istruzione, infine, restituisce il valore `select`, che è il risultato definitivo della valutazione dei criteri.

Questo è tutto. Sono piuttosto soddisfatto di ciò che ho realizzato, soprattutto dal punto di vista concettuale. Come detto, dal punto di vista estetico le cose lasciano piuttosto a desiderare; ed anche in quanto a completezza, ci sono molti punti delicati. Però ci sono molte funzionalità ottenute con poco sforzo (la ricerca sul Finder non ha le condizioni in OR!), cosa è sempre positiva.

Capitolo 061 - Meno finestre per tutti

L'interfaccia utente è sempre stato uno dei miei punti deboli (a vari livelli). Mi sono accorto che questa applicazione ha una quantità di finestre accessorie veramente eccessiva, ed ho pensato di ridurle un po'. In questo breve capitolo, pertanto, non ci sono discorsi eclatanti, ma solo la riduzione del numero delle finestre, ed un piccolo appunto sui menu.

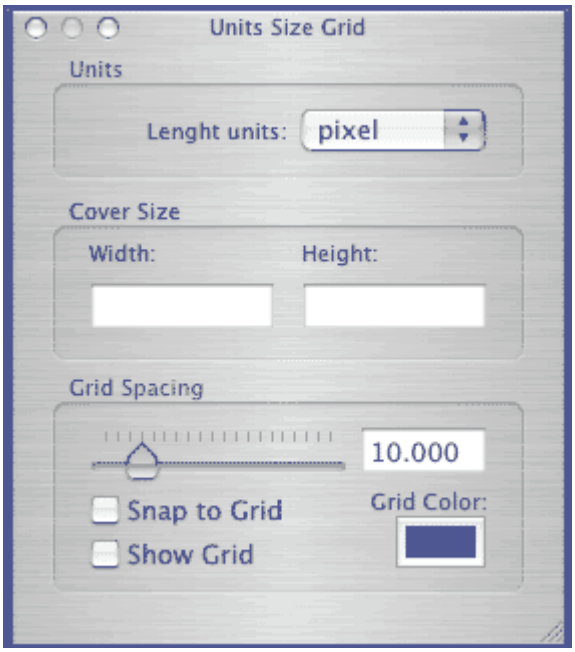
Sorgenti: nessuna

Prima stesura: 1 novembre 2004

FINESTRE AUSILIARIE

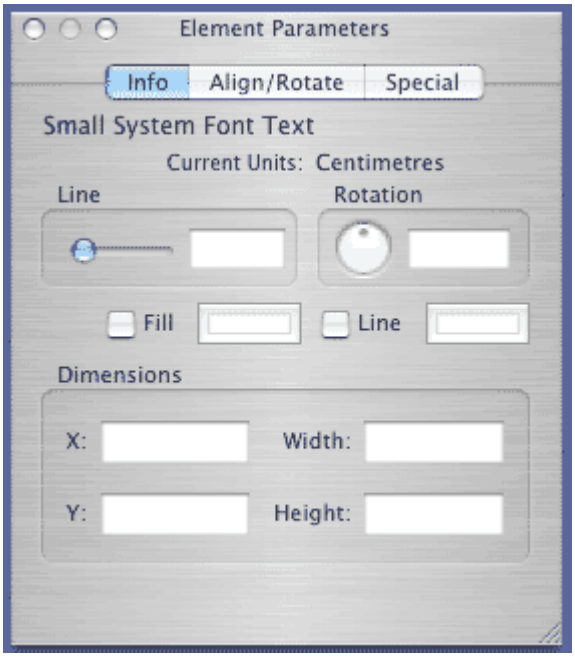
L'applicazione presenta una gran quantità di finestre ausiliare: vale a dire, tutti quei pannelli che sono utilizzati a diversi scopi e di ausilio per le due finestre principali, l'elenco dei file del catalogo ed il disegno di copertine.

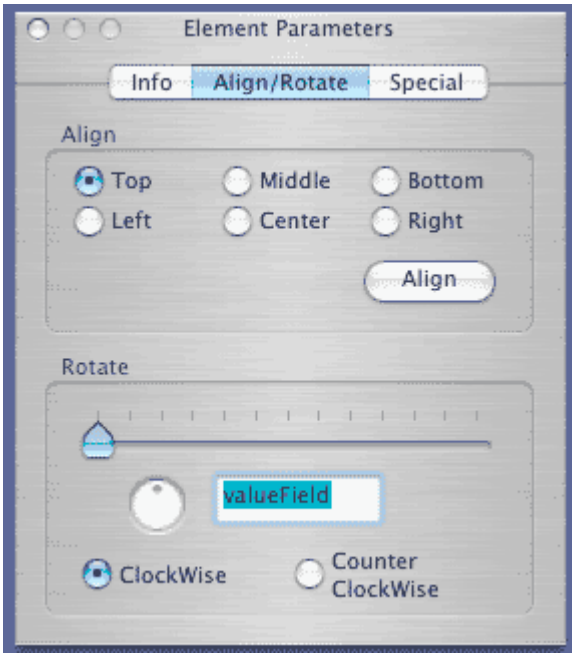
Alcune finestre sono autosufficienti e non saranno toccate, come la finestra *About* e quella delle *ricerche*. Lascio anche intonsa la finestra che mostra le informazioni relative ad un elemento del catalogo, e mi concentro sulle finestre di ausilio al disegno della copertina.



In primo luogo, unisco la finestra che permette di stabilire le dimensioni della copertina con la finestra per impostare unità di misura e griglia. L'unico file `UnitsSizeGrid.nib` piglia il posto di due precedenti file nib, e l'unica classe `UnitSizeGridWinCtl` comprende le precedenti classi `CvrSizeWinCtl` e `GridDlgWinCtl`.

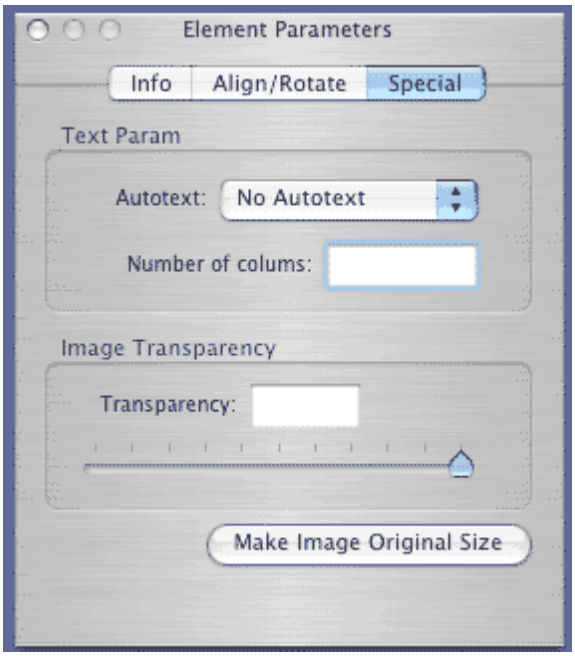
Questa classe non ha nulla di speciale: deriva infatti semplicemente dall'unione delle due classi precedenti, conglobando le variabili d'istanza e i metodi delle due classi da cui deriva. Ovviamente, ci sono alcune semplificazioni e modifiche che si ripercuotono sull'intera applicazione.





La seconda e fondamentale modifica è la riduzione ad una sola finestre della finestre accessorie legate alla manipolazione degli elementi grafici: in questo modo le classi `ElemInfoWinCtl` (informazioni e manipolazione delle caratteristiche comuni degli elementi), `AlignWinCtl` (gestione dell'allineamento), `RotateWinCtl` (rotazione di uno o più elementi), `TxtParWinCtl` (parametri di un campo di testo) e `TranspWinCtl` (gestione della trasparenza degli elementi immagine) sono tutte raccolte all'interno dell'unica classe `ElemParamWinCtl`. Corrispondentemente, scompaiono i vari file nib delle finestre gestite e costruisco una nuova finestra nel file `ElemParam.nib`. Per evitare di costruire una finestra di dimensioni enorme, divido i

vari controlli all'interno di una `NSTabView` tripartita. La prima tabview contiene le informazioni relative all'elemento, la seconda raccoglie i controlli per rotazione ed allineamento, la terza i parametri per i testi e le immagini.



La classe `ElemParamWinCtl` si occupa di gestire tutti i vari controlli; anche qui, le variabili d'istanza ed i metodi derivano direttamente dalle classi da cui deriva, per semplice accumulo. Solo qualche metodo è stato modificato per tenere conto delle mutate condizioni. L'unica parte degna di nota è quella che attiva o disattiva i controlli a seconda della selezione corrente. Le operazioni sono controllate dal metodo `setSelectedElem`, invocato dalla classe `AppDelegate` in seguito ad una notifica di cambio selezione (il metodo `updateCoverViewInfo`: la quantità di finestre da aggiornare si è ridotta di moto: sono sole le due finestre residue descritte).

```
- (void)
setSelectedElem: (CCE_BasicForm *) selElem
{
    unsigned long maskSts ;

    theSelectedElem = selElem;
    // se non ci sono eselementi selezionati
    if ( theSelectedElem == nil )
    {
        // dico che se ne fa nulla
        [ elemId setStringValue: @"No graphic element selected"];
        [ self setControlsEnabled: ENABCTL_NULLENAB ] ;
        return ;
    }
    // ... o ce ne sono troppi
    else if ( theSelectedElem == (CCE_BasicForm*)(-1) )
    {
        [ elemId setStringValue: @"Too many graphic elements selected"];
        // attivo solo rotazione ed allineamento
        [ self setControlsEnabled: (ENABCTL_ALIGN | ENABCTL_ROTATE) ] ;
        return ;
    }
    // se arrivo qui, c'e' un solo elemento selezionato
    // attivo tutti gli elementi della finestra
```

```
    maskSts = (ENABCTL_ELEMINFO | ENABCTL_ROTATE) ;
    // i parametri testo se il caso
    if ( [theSelectedElem isKindOfClass:[CCE_Text class]] )
        maskSts += ENABCTL_TXTPARAM ;
    // i parametri immagine se il caso
    if ( [theSelectedElem isKindOfClass:[CCE_Image class]] )
        maskSts += ENABCTL_TRANSPIMG ;
    // attivo o disattivo i controlli
    [ self setControlsEnabled: maskSts ] ;
    // aggiorno il contenuto dei vari campi
    [ self updateWindowInfo ];
}
```

Qui c'è una fondamentale modifica: il metodo `setControlsEnabled` non ha più un parametro booleano, ma un parametro intero. Questo parametro è in realtà un *campo di bit*: è un numero che deriva dalla sovrapposizione (or bit per bit o anche semplice somma) di più elementi, ciascuno caratterizzato dall'avere un singolo bit ad uno (si tratta di una tecnica standard del programmatore assembler e C per mettere in un unico parametro una serie di variabili booleane).

```
#define         ENABCTL_NULLENAB        0x0000
#define         ENABCTL_ELEMINFO        0x0001
#define         ENABCTL_ALIGN            0x0002
#define         ENABCTL_ROTATE            0x0004
#define         ENABCTL_TXTPARAM        0x0008
#define         ENABCTL_TRANSPIMG        0x0010
```

C'è quindi un bit per ogni porzione di finestra che deve essere attivata o meno a seconda della selezione corrente. Se ad esempio c'è più di un elemento selezionato, solo la porzione della finestra relativa all'allineamento ed alla rotazione sono attive, mentre tutte le altre sono tenute disattive.

Le operazioni sono svolte dal metodo seguente:

```
- (void)
setControlsEnabled: (unsigned long) statusMask
{
    BOOL    status ;
    // informazioni elemento
    status = ( statusMask & ENABCTL_ELEMINFO ) ? YES : NO ;
    [fillCheckbox setEnabled: status ];
    [fillColorWell setEnabled: status ];
    ...
    [heightTextField setEnabled: status ];
    // allineamento
    status = ( statusMask & ENABCTL_ALIGN ) ? YES : NO ;
    [ selectedAlign setEnabled: status ];
    [ execBtn setEnabled: status ];
    // rotazione
    status = ( statusMask & ENABCTL_ROTATE ) ? YES : NO ;
    [ crcSlider setEnabled: status ];
    [ linSlider setEnabled: status ];
    [ linSlider setEnabled: status ];
    [ dirRadioBtn setEnabled: status ];
    // parametri testo
    status = ( statusMask & ENABCTL_TXTPARAM ) ? YES : NO ;
    [ autoTxtMenu    setEnabled: status ];
    [ columnNum        setEnabled: status ];
    // trasparenza immagine
```

```
    status = ( statusMask & ENABCTL_TRANSPIMG ) ? YES : NO ;
    [ transpSlider setEnabled: status ];
    [ transpText setEnabled: status ];
    [ makeImgOriginalsize setEnabled: status ] ;
}
```

Prima di ogni sezione di controlli, si ricava lo stato corrispondente estraendolo con l'operazione di and bit per bit dal parametro; con questo valore si adegua l'aspetto dei controlli alla situazione.

ABILITAZIONE AUTOMATICA DEI MENU

Nell'operazione di riunificazione delle finestre, mi sono imbattuto del problema di come abilitare o disabilitare le voci di menu che permettevano di aprire queste finestre. Finora avevo collegato le voci di menu a dei metodi della classe `AppDelegate`, che a loro volta aprivano le finestre. Questa situazione, benché molto comoda, ha delle controindicazioni: permette di aprire delle finestre che in effetti non hanno significato nel contesto. Ad esempio, con la finestra del catalogo dei file in primo piano, era possibile aprire la finestra che imposta le dimensioni della copertina. Sarebbe più corretto che in questo contesto la voce sia disabilitata. Questo mi ha portato ad approfondire l'argomento del *First Responder* e dell'abilitazione automatica dei menu.

Quando costruisco una finestra all'interno di un file nib, ho sempre disponibili due classi: la classe *File's Owner* e la classe *First Responder*. La prima in genere diventa un'istanza della classe controllore della finestra, mentre la seconda identifica la classe destinata a ricevere in prima battuta gli eventi. Si tratta spesso della finestra, ma non appena l'utente si focalizza su di elemento (ad esempio, un campo testo), il primo risponditore diventa quel campo stesso. Ci sono due cose importanti da puntualizzare: il primo risponditore può essere impostato da programma o comunque indipendentemente dall'utente. Questo accade ad esempio se si valorizza l'outlet `initialFirstResponder` di una finestra con uno degli elementi presenti all'interno della finestra stessa, oppure se si usa il metodo `makeFirstResponder`. La seconda cosa da tenere presente è che il primo risponditore è solo il primo (appunto) di una lunga catena, che porta da un elemento dell'interfaccia al suo superiore gerarchico (ogni `NSView` è una sottoclasse di `NSResponder`, classe che appunto raccoglie tutti i meccanismi per rispondere agli eventi) fino ad arrivare alla finestra, al delegato della finestra, al controllore, al documento, all'applicazione.

Ora che questo argomento è chiaro (spero) passo a parlare dei menu. Ad ogni evento, ogni voce di menu valuta il proprio stato e la voce corrispondente si abilita o si disabilita a seconda delle condizioni operative. Nella situazione più semplice, la cosa avviene automaticamente. Ogni voce di menu che intende eseguire qualche operazione ha associato un target ed una action. Ad ogni evento, allora, verifica se il target esiste, e prova a vedere se il target stesso o una classe della catena dei risponditori appesa al target, è in grado di rispondere al metodo indicato dalla action. Se ciò accade, il menu è abilitato; se nessuno è in grado di rispondere, il menu è disabilitato.

Nel caso dei menu che aprivano le finestre accessorie, non sapendo quale target attribuire, avevo deputato la classe `AppDelegate`. Tuttavia, poiché questa classe è sempre presente e disponibile alla risposta, le voci di menu corrispondenti sono sempre abilitate.

Per avere voci di menu più corrette semanticamente, occorre spostare il target. Ho diviso le finestre accessorie in quattro categorie: quelle pertinenti al catalogo di file, quelle pertinenti al disegno della copertina, quelle relative a qualche caratteristica di un elemento grafico, e quelle globali dell'applicazione. Le finestre globali sono ad esempio la finestra di About e delle preferenze. Le voci di menu corrispondenti sono quindi rimaste collegate alla classe `AppDelegate`, che contiene anche i metodi per aprire le finestre. Le finestre relative al catalogo dei file sono quelle delle rierche e delle informazioni sul file; i metodi corrispondenti sono stati quindi spostati all'interno della classe `ListWinCtl`. Sempre qui ho portato anche il metodo per aprire la finestra che contiene la copertina, in modo che il menu per mostrare la copertina sia attivo solo se in primo piano c'è una finestra di catalogo. Infine, le finestre relative alla copertina nel suo complesso (la finestra unificata con dimensioni, unità e griglia e il pannello con gli strumenti) passano in carico alla classe `CoverWinCtl`, mentre la finestra unificata con informazioni, rotazione, eccetera rimane in carico alla classe `CoverView`.

Perché tutto questo meccanismo funzioni, occorre modificare il target delle voci di menu nel file `MainMenu.nib`. Il target non è più la classe `AppDelegate`, ma il generico *First Responder* (occorre aggiungere a tale classe i vari metodi per poter attribuire le action alle voci) presente nel file.

Per completare le operazioni, ci sono due istruzioni da aggiungere. La prima istruzione serve a selezionare il giusto tab nella finestra ElemParam; ho quindi aggiunto un metodo alla classe `ElemParamWinCtl` che selezioni il tab corretto a seconda del comando impartito; ad esempio:

```
- (IBAction)
showRotateWin: (id)sender
{
    // mostro la finestra invocando l'istanza condivisa
    [ [ ElemParamWinCtl sharedElemParamWinCtl ] showWindow: sender ] ;
          [   [   ElemParamWinCtl   sharedElemParamWinCtl   ]   selectPanel:
SECTSEL_ALIGNROTATE ] ;
}
```

La seconda istruzione serve a riportare il focus e il primo risponditore al posto giusto dopo qualche operazione sulla finestra della copertina. All'interno del metodo `handleMouseClick` aggiungo l'istruzione seguente:

```
    [[winCtl window] makeFirstResponder: self];
```

in modo che la catena di risponditori parta (nuovamente) dalla `CoverView`.

MANTENERE LA DIMENSIONE DELLE COLONNE

Una cosa che mi irritava nella finestra del catalogo era la dimensione delle colonne, sempre ballerina e che non si mantiene uguale tra una finestra ed un'altra. Il problema è dovuto al fatto che la `NSOutlineView` possiede due colonne fisse (nome del file e campo is2Print), mentre le altre colonne sono costruite di volta in volta quando necessario. In pratica questo significa che il meccanismo di persistenza, fornito di serie con Cocoa per conservare le dimensioni delle colonne, non funziona. Ed allora, lo realizzo io, salvando alla chiusura della finestra le larghezze delle varie colonne, ed impostandole

all'apertura.

Ovviamente, il posto più sensato dove conservare queste informazioni è il file delle preferenze.

Per prima cosa, arricchisco la classe `UserPrefs` con due metodi per recuperare ed impostare valori generici di preferenze:

```
+ (void)
setSinglePrefValue: (id) val forKey: (id) key
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [ defaults setObject: val forKey: key ] ;
}

+ (id)
getSinglePrefValue: (id) key
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    return ( [ defaults objectForKey: key ]) ;
}
```

Ora arricchisco il dizionario delle preferenze con un po' di valori: tutte le larghezze delle colonne, contraddistinte da una stringa che le identifica e che funziona da chiave nel dizionario.

Così facendo, all'apertura della finestra, nel metodo `windowDidLoad`, imposto la larghezza della colonna che contiene il nome del file:

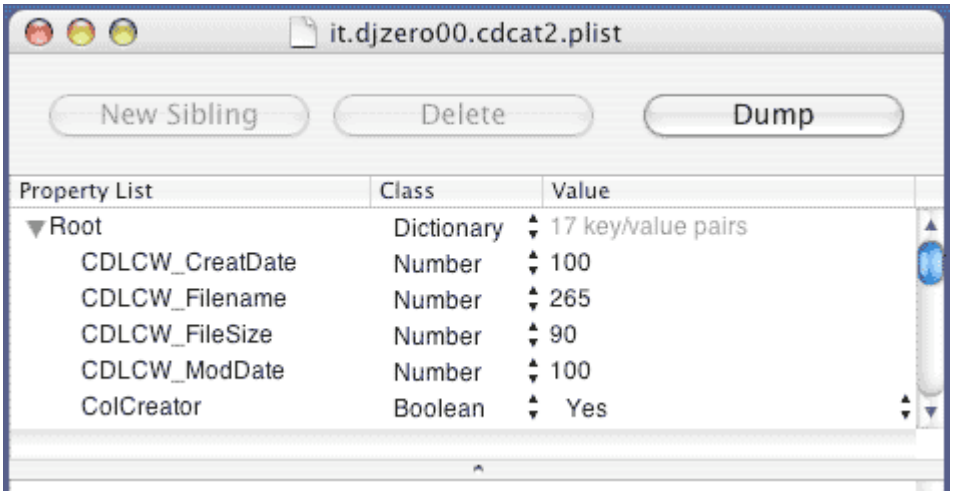
```
- (void)
windowDidLoad
{
    ...
    imageAndTextCell = [[[ImageAndTextCell alloc] init] autorelease];
    cw = [ [UserPrefs getSinglePrefValue: keyCDLCW_Filename ] intValue ];
    if ( cw <= 0 ) cw = 100 ;
    [ tableColumn setWidth: cw ];
    [ tableColumn setDataCell:imageAndTextCell ];
    ...
```

Ugualmente procedo all'interno della funzione `configureColumn` (riportata solo per alcuni casi):

```
void
configureColumn( NSTableColumn *tc, NSString * colId )
{
    ...
    if ( [ colId isEqual: COLID_POSIXPERM ] )
    {
            FilePosixPerm       *myDF3 = [[[ FilePosixPerm alloc ] init ]
autorelease ];
         int     cw = [ [UserPrefs getSinglePrefValue: keyCDLCW_PosixPerm ]
intValue ];
        if ( cw <= 0 ) cw = 100 ;
        [ tc setWidth: cw ];
        [[tc dataCell ] setFormatter: myDF3 ];
        return ;
```

```
    }
    ...
    if ( [ colId isEqual: COLID_OWNERNAME ] )
    {
         int     cw = [ [UserPrefs getSinglePrefValue: keyCDLCW_OwnerName ]
intValue ];
        if ( cw <= 0 ) cw = 100 ;
        [ tc setWidth: cw ];
    }
}
```

Ho preso l'accortezza di impostare un valore di 100 pixel nel caso in cui il valore per qualche motivo sia negativo o nullo (ad esempio, perché è la prima volta che parte l'applicazione, ed il file delle preferenze è ancora vuoto).



Rimane da determinare il posto in cui salvare questi valori; nella classe `ListWinCtl` c'è già un metodo che è chiamato alla chiusura della finestra, e che serve a restituire YES per indicare alla classe `CdCatDoc` che si intende chiudere il documento. Lo utilizzo anche per salvare le dimensioni delle colonne.

```
- (BOOL)
shouldCloseDocument
{
    NSTableColumn        * tc ;
    NSNumber            * cw ;
    // ne approfitto per mettere a posto le larghezze di default
    tc = [ fullList tableColumnWithIdentifier: COLID_FILENAME];
    cw = [ NSNumber numberWithInt: [ tc width ] ] ;
    [ UserPrefs setSinglePrefValue: cw forKey: keyCDLCW_Filename ];
    tc = [ fullList tableColumnWithIdentifier: COLID_MODDATE];
    if ( tc )
    {
        cw = [ NSNumber numberWithInt: [ tc width ] ] ;
        [ UserPrefs setSinglePrefValue: cw forKey: keyCDLCW_ModDate ];
    }
    ...
    tc = [ fullList tableColumnWithIdentifier: COLID_OSTYPE];
    if ( tc )
    {
        cw = [ NSNumber numberWithInt: [ tc width ] ] ;
        [ UserPrefs setSinglePrefValue: cw forKey: keyCDLCW_OSType ];
    }
    return ( YES );
```

```
}
```

Con questo, le dimensioni sono conservate all'interno del file delle preferenze e si mantengono tra un lancio dell'applicazione ed il successivo.

Capitolo 062 - Esportazione dati

Ancora un breve capitolo per esportare in un file di testo l'elenco dei file presenti all'interno del catalogo.

Sorgenti: nessuna

Prima stesura: 10 novembre 2004

I NOMI DELLE PREFERENZE

Prima di esportare dati, cambio i nomi delle preferenze. Fino ad ora mi ero trovato ad utilizzare delle `#define` per individuare delle stringhe con cui individuare alcune grandezze la cui specifica era a cura dell'utente. Ad esempio:

```
#define     keyColModDate     @"ColModDate"
...
#define     keyCDLCW_ModDate     @"CDLCW_ModDate"
```

specificano la prima la presenza o meno di una colonna nella finestra del catalogo, mentre la seconda specifica la dimensione in pixel nella visualizzazione di quella stessa colonna. Ogni volta che aggiunto preferenze riguardanti i singoli attributi di un file, devo aggiungere una diecina di `#define`, una per ogni attributo.

Trovo tutto ciò piuttosto scomodo, ed ho quindi deciso di lavorare in altro modo. Il punto di partenza è il nome di quell'attributo, che è anche l'identificatore della colonna. Sono un gruppo di `#define` in `CatFileInfo.h`:

```
#define         COLID_FILENAME          @"fileName"
#define         COLID_MODDATE         @"modDate"
...
```

Per ottenere un nome di preferenza, faccio precedere la stringa da un prefisso opportuno, costruendo così una nuova stringa, che uso appunto come identificatore della grandezza.

```
#define         keyDsplPrefPrefix         @"CDLDF_"
#define         keyExprPrefPrefix         @"CDLPF_"
#define         keyColWPrefPrefix         @"CDLCW_"
```

Ho definito quindi questi tre prefissi, per indicare il primo la visualizzazione o meno di una colonna, mentre il terzo specifica la larghezza di una colonna; il secondo prefisso sarà utilizzato poco sotto per determinare l'esportazione o meno di quell'attributo in un file di testo.

Ho poi definito una sintetica funzione in `djZeroUtils.m` per costruire i nomi:

```
NSString *
bldStrPref( NSString * pref, NSString * colid )
```

```
{
    return ( [ pref stringByAppendingString: colid] );
}
```

Infine, con infinita pazienza, sono andato a sostituire nell'intero codice tutte le occorrenze delle precedenti `#define` con nuove espressioni.

A titolo di esempio, mostro solo come cambia il metodo che salva le dimensioni delle colonne all'interno delle preferenze:

```
- (BOOL)
shouldCloseDocument
{
    NSTableColumn        * tc ;
    NSNumber            * cw ;

#define        SET_COLWIDTH_PREF( quale )                          \
    tc = [ fullList tableColumnWithIdentifier: quale ];         \
    if ( tc )                                                  \
    {                                                          \
        cw = [ NSNumber numberWithInt: [ tc width ] ] ;         \
        [ UserPrefs setSinglePrefValue: cw forKey:             \
            bldStrPref(keyColWPrefPrefix, quale) ];         \
    }

    // ne approfitto per mettere a posto le larghezze di default
    tc = [ fullList tableColumnWithIdentifier: COLID_FILENAME];
    cw = [ NSNumber numberWithInt: [ tc width ] ] ;
                [    UserPrefs    setSinglePrefValue:    cw    forKey:
bldStrPref(keyColWPrefPrefix, COLID_FILENAME) ];
    SET_COLWIDTH_PREF( COLID_MODDATE ) ;
    ...
    SET_COLWIDTH_PREF( COLID_MODDATE ) ;
    return ( YES );
}
```

Attenzione che c'è anche un altro importante file da modificare: si tratta di `defCodeValues.dict`, dove sono contenuti i valori di default. Occorre cambiare tutti i nomi:

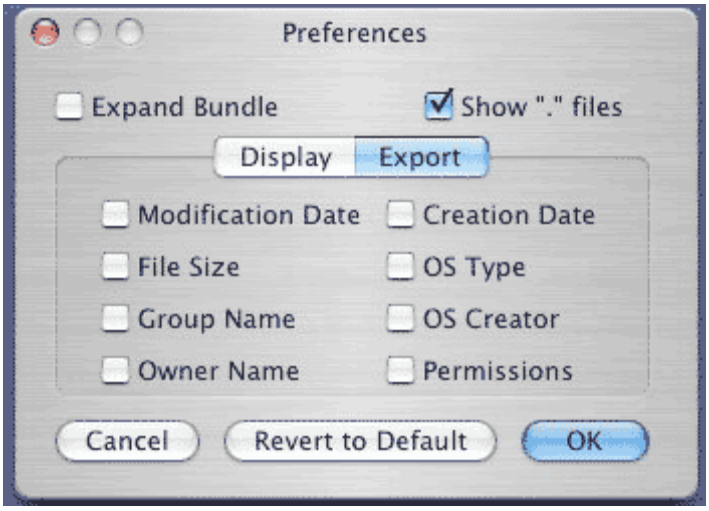
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE   plist   PUBLIC   "-//Apple   Computer//DTD   PLIST   1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CDLDF_creatDate</key>
    <true/>
    <key>CDLDF_creatorCode</key>
    <false/>
    ...
    <key>ExpandBundle</key>
    <false/>
    <key>ShowDotFiles</key>
    <true/>
</dict>
</plist>
```

Alla fine, sono rimaste due sole preferenze con un nome proprio, le grandezze booleane che indicano l'espansione o meno dei *bundle* e la visualizzazione o meno dei così detti

dot file.

ESPORTARE IN UN FILE DI TESTO

Accade talvolta di dover trasferire i dati di un programma verso altri programmi; spesso il formato di interscambio non è altro che un semplice file di testo. Mi prefiggo quindi di produrre un file di testo con al suo interno l'elenco dei file presenti nel catalogo prescelto. Per complicarmi la vita, decido di utilizzare ancora una volta il campo `is2Print` per determinare se un dato file (e sui discendenti se si tratta di una cartella) è esportato oppure no; inoltre, permetto all'utente di specificare quali informazioni relative al file sono presenti. L'utente può quindi determinare, all'interno delle preferenze, quali campi esportare, così come decide quali campi sono visualizzati nella finestra.



Allo scopo, predispongo una finestra delle preferenze modificata: ho aggiunto una `NSTabView` con due viste. Nella prima vista sono comprese le preferenze di quali colonne visualizzare; nella seconda vista, assolutamente uguale, si specificano quali colonne/attributi esportare. Ho aggiunto un outlet verso la `NSMatrix` con i nuovi pulsanti, e ho modificato alcuni metodi di `PrefWinCtl` per tenere conto dei nuovi valori.

A titolo di esempio, riporto il metodo che aggiorna le preferenze secondo i desideri dell'utente:

```
- (void)
userSelectUpdate:(id)sender
{

#define                              MATRIX_GET_STATE(    key,    row,    col
)                                                  \
                                                                [defDispValues
setObject:                                                                  \
          ([[colDspMatrix cellAtRow: row column:col] state ] ? lsPrefsYes :
lsPrefsNo)      \
        forKey:key]

    // devo recuperare i valori dalla finestra
    [defDispValues setObject:
```

```
        ([expandBundleButton state] ? lsPrefsYes : lsPrefsNo)
        forKey:keyExpandBundle];
    [defDispValues setObject:
        ([showDotFilesButton state] ? lsPrefsYes : lsPrefsNo)
        forKey:keyShowDotFiles];
    // recupero i valori secondo i controlli ed assegno le prefs
     MATRIX_GET_STATE( bldStrPref(keyDsplPrefPrefix, COLID_MODDATE),      0,
0 );
        MATRIX_GET_STATE(  bldStrPref(keyDsplPrefPrefix,  COLID_CREATDATE),
    0, 1 );
    MATRIX_GET_STATE( bldStrPref(keyDsplPrefPrefix, COLID_FILESIZE),     1,
0 );
                        MATRIX_GET_STATE(      bldStrPref(keyDsplPrefPrefix,
COLID_OSTYPE),        1, 1 );
        MATRIX_GET_STATE(  bldStrPref(keyDsplPrefPrefix,  COLID_GROUPNAME),
    2, 0 );
        MATRIX_GET_STATE(  bldStrPref(keyDsplPrefPrefix,  COLID_OSCREATOR),
    2, 1 );
        MATRIX_GET_STATE(  bldStrPref(keyDsplPrefPrefix,  COLID_OWNERNAME),
    3, 0 );
        MATRIX_GET_STATE(  bldStrPref(keyDsplPrefPrefix,  COLID_POSIXPERM),
    3, 1 );

#define                              MXEXPR_GET_STATE(    key,    row,    col
)                                            \
                                                            [defDispValues
setObject:                                                            \
         ([[colPrtMatrix cellAtRow: row column:col] state ] ? lsPrefsYes :
lsPrefsNo)     \
        forKey:key]

     MXEXPR_GET_STATE( bldStrPref(keyExprPrefPrefix, COLID_MODDATE),      0,
0 );
        MXEXPR_GET_STATE(  bldStrPref(keyExprPrefPrefix,  COLID_CREATDATE),
    0, 1 );
    MXEXPR_GET_STATE( bldStrPref(keyExprPrefPrefix, COLID_FILESIZE),     1,
0 );
                        MXEXPR_GET_STATE(      bldStrPref(keyExprPrefPrefix,
COLID_OSTYPE),        1, 1 );
        MXEXPR_GET_STATE(  bldStrPref(keyExprPrefPrefix,  COLID_GROUPNAME),
    2, 0 );
        MXEXPR_GET_STATE(  bldStrPref(keyExprPrefPrefix,  COLID_OSCREATOR),
    2, 1 );
        MXEXPR_GET_STATE(  bldStrPref(keyExprPrefPrefix,  COLID_OWNERNAME),
    3, 0 );
        MXEXPR_GET_STATE(  bldStrPref(keyExprPrefPrefix,  COLID_POSIXPERM),
    3, 1 );
    // recuperati i valori; li assegno ai correnti
    // salvo le preference su file
    [ UserPrefs savePrefsToDefault: defDispValues ];
    // e per finire nascondo la finestra
    [[self window] setIsVisible: FALSE ];
}
```

Non ho fatto altro che applicare le modifica sul nome delle preferenze, ed aggiungere nuove preferenze, molto simili alle precedenti.

ESPORTAZIONE

Sono a questo punto pronto per esportare i dati. Aggiungo al `MainMenu.nib` una voce di menu che scateni l'intero processo chiamando il metodo `textExport` della classe

`ListWinCtl`:

```
- (void)
textExport: (id)sender
{
    NSSavePanel    *sPanel = [ NSSavePanel savePanel ] ;
    // personalizzo il pannello di salvataggio
    [ sPanel setTitle:@"Export text file" ];
    // i file salvati avranno l'estensione lscat
    [ sPanel setRequiredFileType: @"txt" ];
    if ( [ sPanel runModal ] == NSOKButton )
    {
        // recupero il nome del file
        NSString    * aFile = [ sPanel filename ] ;
        NSString    * exportStr ;
        exportStr = [ [[ self document] dataSource] exportData ];
        if ( ! [ exportStr writeToFile: aFile atomically: YES ] )
        {
            // che ne so
        }
    }
}
```

Questo metodo chiede la specifica di un file di testo tramite il consueto pannello; se l'utente non cambia idea e fornisce un adeguato percorso, il metodo `exportData` produce una `NSString` con tutti i dati da esportare, e la semplice istruzione `writeToFile:` produce il file testo voluto. La potenza di Cocoa e soprattutto della classe `NSString` permette di sorvolare su molti dettagli.

Adesso non rimane che scrivere il metodo `exportData`, proprio della classe `CatDataSrc`. Ancora una volta, si tratta di eseguire una discesa ricorsiva della base di dati e di produrre, per ogni elemento attraversato, una stringa apposita. Questa volta utilizzo un metodo piuttosto pedestre:

```
- (NSString *)
exportData
{
    NSString    * expStr = [ NSString string ] ;
    int           jj ;
    for ( jj = 0 ; jj < [ startPoint count ] ; jj ++ )
    {
        FileStruct *    item ;
        item = [ startPoint objectAtIndex: jj ] ;
        expStr = [ item exportData: expStr withLevel: 0 ] ;
    }
    return ( expStr ) ;
}
```

Qui si esaminano tutti i volumi presenti, e per ognuno di essi si comanda l'esportazione dei dati. Qui occorre notare come la stringa sia passata come parametro e poi restituita dal metodo; si suppone che alla stringa siano state aggiunte le informazioni proprie di quell'elemento (e di tutti i discendenti dell'elemento). Inoltre, come già visto altrove, faccio in modo che gli elementi siano indentati in accordo alla profondità della struttura delle cartelle.

Al termine di questa cascata di metodi di esportazione c'è finalmente questo metodo,

proprio della classe `FileStruct`:

```
- (NSString *)
exportData: (NSString *) expStr withLevel: (int) lev
{
    if ( [ self is2Print ] )
    {
        int    i ;
        int    numFile = [ fileList count ] ;
        expStr = [ self writeExportToString: expStr withLevel: lev ] ;
        for ( i = 0; i < numFile; i++)
        {
            FileStruct    * tmpFile ;
            // costruisco l'albero
            tmpFile = [ fileList objectAtIndex: i ] ;
            expStr = [ tmpFile exportData: expStr withLevel: (lev+1) ] ;
        }
    }
    return ( expStr );
}
```

La prima cosa che il metodo si chiede è il valore di `is2Print`. Se Falso, restituisce intonsa la stringa passatagli, dal momento che non si devono aggiungere dati a quelli da esportare. Se invece `is2Print` è vero, aggiunge i propri dati alla stringa e poi passa il compito ordinatamente ai propri discendenti (se presenti). Alla fine, la stringa, ripetutamente aggiornata, è restituita al chiamante.

FORMATTARE IL TESTO ESPORTATO

Il lavoro finale, di produzione della stringa di testo da aggiungere alla stringa già disponibile, è svolta da un ulteriore metodo; questo non è nel suo impianto particolarmente difficile, ma ci sono alcuni piccoli accorgimenti da utilizzare se si intende ottenere un file testo leggibile in maniera decente.

```
- (NSString *)
writeExportToString: (NSString *) expStr withLevel: (int) lev
{
    NSString    * tmpStr ;
     // la stringa di prefisso con un po' di tabulazioni, ed il nome del
file
     tmpStr = [ expStr stringByAppendingFormat: @"%@%@\t", prefixLevString(
lev ), [ self fileName ] ];
    // esamino poi i vari attributi: data di modifica
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_MODDATE)] boolValue] )
    {
        // costruisco un formattatore per le date
        NSDate        * tmp = [ self modDate ] ;
        NSDateFormatter *dateFormat = [[NSDateFormatter alloc]
                                    initWithDateFormat:@"%d   %b   %Y   %H:%M"
allowNaturalLanguage:NO];
        NSString * localString = [ dateFormat stringForObjectValue: tmp ] ;
        tmpStr = [ tmpStr stringByAppendingFormat: @"%@\t", localString ];
    }
    // data di creazione
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_CREATDATE)] boolValue] )
    {
```

```
        // costruisco un formattatore per le date
        NSDate        * tmp = [ self creatDate ] ;
        NSDateFormatter *dateFormat = [[NSDateFormatter alloc]
                                    initWithDateFormat:@"%d   %b   %Y   %H:%M"
allowNaturalLanguage:NO];
        NSString * localString = [ dateFormat stringForObjectValue: tmp ] ;
        tmpStr = [ tmpStr stringByAppendingFormat: @"%@\t", localString ];
    }
    // dimensione del file
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_FILESIZE)] boolValue] )
    {
        // costruisco un formattatore per la dimensioen del file
        FileSizeForm    * numFmt = [ [ FileSizeForm alloc ] init ] ;
        long long    tmp = [ self fileSize ] ;
         NSString * localString = [ numFmt stringForObjectValue: [ NSNumber
numberWithLongLong: tmp ] ] ;
        tmpStr = [ tmpStr stringByAppendingFormat: @"%@\t", localString ];
    }
    // nome del gruppo e del proprietario
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_GROUPNAME)] boolValue] )
    {
        // e' una stringa, aggiungo brutalmente
             tmpStr  =  [  tmpStr  stringByAppendingFormat:  @"%@\t",  [  self
ownGroupName ] ];
    }
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_OWNERNAME)] boolValue] )
    {
        // e' una stringa, aggiungo brutalmente
             tmpStr  =  [  tmpStr  stringByAppendingFormat:  @"%@\t",  [  self
ownerName ] ];
    }
    // posix permissions
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_POSIXPERM)] boolValue] )
    {
        // formattatore per i permessi di lettura/scrittura/esecuzione
        FilePosixPerm    * numFmt = [ [ FileSizeForm alloc ] init ] ;
        long    tmp = [ self filePosixPerm ] ;
         NSString * localString = [ numFmt stringForObjectValue: [ NSNumber
numberWithLong: tmp ] ] ;
        tmpStr = [ tmpStr stringByAppendingFormat: @"%@\t", localString ];
    }
    // tipo e creatore
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_OSCREATOR)] boolValue] )
    {
        // formattatore per tipo/creatore in OS9
        TOS9TCForm    * numFmt = [ [ FileSizeForm alloc ] init ] ;
        long    tmp = [ self creatorCode ] ;
         NSString * localString = [ numFmt stringForObjectValue: [ NSNumber
numberWithLong: tmp ] ] ;
        tmpStr = [ tmpStr stringByAppendingFormat: @"%@\t", localString ];
    }
         if  (  [[  UserPrefs  getPrefValue:  bldStrPref(keyExprPrefPrefix,
COLID_OSTYPE)] boolValue] )
    {
        // formattatore per tipo/creatore in OS9
        TOS9TCForm    * numFmt = [ [ FileSizeForm alloc ] init ] ;
        long    tmp = [ self typeCode ] ;
         NSString * localString = [ numFmt stringForObjectValue: [ NSNumber
numberWithLong: tmp ] ] ;
```

```
        tmpStr = [ tmpStr stringByAppendingFormat: @"%@\t", localString ];
    }
    return ( [ tmpStr stringByAppendingString: @"\n" ] ) ;
}
```

In effetti, finché l'attributo è ben rappresentato da una stringa, uso l'attributo così come si trova? Negli altri casi, utilizzo invece (come al solito) un formattatore per poterne dare una rappresentazione più leggibile.

Capitolo 063 - Multithread

Ho avuto il mio primo riscontro sull'applicazione (grazie a Paolo e a Mauro). È stata criticata l'assenza del pulsante di cancellazione delle operazioni di catalogazione (o meglio, il pulsante c'era, ma non funzionava). Ne ho approfittato allora per rendere l'applicazione multithread, e di permettere la catalogazione in background.

Sorgente: ho guardato un esempio di Apple, ma poi ho fatto a modo mio.

Prima stesura: 16 novembre 2004.

TASK E THREAD

Prima di cominciare, un po' di teoria (semplifico e traviso, ma cerco di spiegarmi). Molti dei miei lettori (forse tutti) usano un sistema operativo *multitasking*. Cosa significa? Che il PC è in grado di svolgere contemporaneamente diversi programmi, l'utente è in gradi saltare tranquillamente da un programma ad un altro, e le attività di un programma non sono influenzate (o lo sono ad alto livello) dalle attività di un altro programma (detta in termini crudi: se un programma va in crash, gli altri continuano bellamente a lavorare). L'unità minima di esecuzione è detta *task*; un task è quindi un programma in grado di lavorare in autonomia, e capace di interagire con altri task secondo protocolli ben definiti. Oltre al multitasking, è possibile un'altra suddivisione delle operazioni di un programma, un po' meno drastica e quindi più leggera: i *thread*.

Due task diversi operano in contesti diversi, in aree di memoria differenti, e per accedere alla stessa risorsa utilizzano meccanismi ben definiti del sistema operativo; crudamente, un task non ha accesso alle variabili o risorse proprie di un altro task. Per comunicare tra loro, i task devono accordarsi su un dato protocollo, e scambiarsi informazioni in maniera precisa e strutturata. Il sistema operativo poi esegue l'uno o l'altro task a seconda delle condizioni operative correnti, ne permette la convivenza pacifica e li sorveglia benignamente dall'alto.

I thread invece condividono la memoria, le risorse e quant'altro. I thread sono unità di esecuzione che possono essere svolte in parallelo (in particolare, su due processori diversi), ma in cui l'interazione comune deve essere ben gestita da parte del programmatore; sono insomma un meccanismo che permette di eseguire operazioni in parallelo, ma la cui robustezza dipende essenzialmente dalla buona programmazione piuttosto che dal sistema operativo.

Normalmente, un programma diviso in task o thread è meno efficiente di un programma monolitico: bisogna tenere conto del carico aggiuntivo dovuto allo scambio di informazioni tra i vari task/thread. Di contro, la suddivisione in task o thread rende il programma più adatto alla vita in un contesto mutante, come ad esempio nell'interazione con l'utente. In tal caso si dice che il programma è più *responsivo*, cioè risponde con più prontezza all'utente (perché ad esempio c'è un thread che si occupa di interagire con l'utente, mentre le operazioni di lunga durata sono svolte da un altro thread).

Cocoa mette a disposizione due classi per realizzare i task e i thread. Il loro nome è,

sorprendentemente e rispettivamente, `NSTask` e `NSThread`. Con `NSTask` si lancia un processo indipendente dall'applicazione in esecuzione (come se si lanciasse da Terminale), mentre con `NSThread` si lancia appunto un thread all'interno dell'applicazione corrente. Per comunicare tra task, occorre mettere in piede un meccanismo di comunicazione (*pipe*, oppure oggetti distribuiti); per comunicare tra thread, ci si potrebbe limitare a scambiarsi messaggi tra oggetti, come se niente fosse.

Il problema sorge quando diversi thread tentano di accedere contemporaneamente alle stesse risorse (diciamo alla stessa locazione di memoria). Avere thread che si disinteressano della possibilità è un metodo molto semplice per creare danni ed inserire errori difficili da trovare. Cocoa fornisce meccanismi di protezione, in modo che thread diversi non possano accedere contemporaneamente alla stessa risorsa, ma rimane compito del programmatore scrivere bene il codice.

Di più, non tutte le librerie fornite da Apple sono *thread-safe*, ovvero, funzionano correttamente se chiamate da diversi thread. Ad esempio, le classi della *Application Kit* (in pratica, tutte le classi dell'interfaccia grafica) non lo sono. Questo significa che non posso avere diversi thread che gestiscono l'interfaccia grafica, pena la confusione massima. Le librerie standard C e quelle per la gestione dello I/O sono generalmente sicure, ma in generale bisogna fare parecchia attenzione.

IL SERVER DI CATALOGAZIONE

Una idea che mi girava in testa da molto tempo era appunto quella di rendere la procedura di catalogazione in un thread separato da quello principale dell'applicazione. In questo modo è possibile lanciare un thread per ogni procedura di aggiunta volumi; così facendo, l'utente non deve aspettare necessariamente il termine della catalogazione, ma può proseguire le operazioni sull'applicazione (ad esempio, operando su un altro catalogo). Poiché la parte di catalogazione non ragiona fondamentalmente sull'interfaccia, ma si interessa principalmente di ricevere informazioni dai volumi tramite chiamate Carbon, dovrebbe essere thread-safe. In effetti, finora, ho lavorato tranquillo senza problemi (questo non significa che non ci siano errori).

Ho estratto quindi le istruzioni che eseguono la catalogazione ed ho costruito una nuova classe `CatServer`.

```
@interface CatServer : NSObject
{
    NSAutoreleasePool    * pool;
    CdCatDoc            * theCat ;
}

// metodo per l'inizializzazione
- (void) makeVolCat: (NSArray *) theArgs ;
- (void) stopCatalog ;
```

La classe possiede solo due metodi: il primo esegue le operazioni di catalogazione utilizzando i meccanismi messi a disposizione da `VolInfo`, `FileStruct` e `CatFileInfo`. Il secondo metodo serve per terminare le operazioni di catalogazione e sistemare le cose.

Il metodo `makevolCat:` ha un solo argomento; si tratta tuttavia di un `NSArray`, che contiene una serie di informazioni. La presenza di un unico argomento è imposta dal meccanismo di creazione di un thread (ci arrivo tra un po'). L'oggetto `NSArray` contiene le due informazioni necessarie per effettuare una catalogazione. In primo luogo un riferimento all'oggetto `CdCatDoc` destinato a contenere le informazioni; il secondo elemento è a sua volta un `NSArray` con la lista dei nomi dei volumi da catalogare.

Detto questo, il metodo è presto presentato: non ho fatto altro che estrarre la maggior parte delle istruzioni dal metodo `performAddFilesModal` di `CdCatDoc`:

```
- (void)
makeVolCat: (NSArray *) theArgs
{
    unsigned long long    totalSpace, freeSpace, usedspace ;
    NSEnumerator     * enumerator ;
    NSString        * volPath ;
    NSDictionary    * fsattrs ;
    NSArray            * volList = [ theArgs objectAtIndex: 1 ];
    // recupero il catalogo
    theCat = [ theArgs objectAtIndex: 0 ];
    // devo costruire un pool di memoria per le attivita' correnti
    pool = [[NSAutoreleasePool alloc] init];
    // spazzolo tutti i vari volumi interessati
    enumerator = [volList objectEnumerator];
    // finche' ci sono volumi da trattare
    while ( volPath = [enumerator nextObject] )
    {
        // il volPath ho il nome completo del volume
        VolInfo            * fInfo ;
        FSSpec            myFSSpec ;
        FSRef            myFSRef;
        FSVolumeInfo    myInfo;
        FSCatalogInfo    catinfo ;
        OSStatus        status ;
        NSArray            * stsresp;
        // recupero informazioni specifiche di volume
             status  =  FSPathMakeRef  ([volPath  fileSystemRepresentation],
&myFSRef, NULL);
        if (status == noErr)
            status = FSGetCatalogInfo (&myFSRef, kFSCatInfoNone, & catinfo,
NULL,
                                       & myFSSpec, NULL);
        // adesso in myFSSpec ho il mio FSSpec
                 status  =  FSGetVolumeInfo  (  myFSSpec.vRefNum,  0,  NULL,
kFSVolInfoFileCount,
            & myInfo, NULL, NULL );
                            fsattrs    =    [[NSFileManager    defaultManager]
fileSystemAttributesAtPath: volPath];
                  totalSpace   =   [[fsattrs   objectForKey:NSFileSystemSize]
unsignedLongLongValue] ;
                freeSpace   =   [[fsattrs   objectForKey:NSFileSystemFreeSize]
unsignedLongLongValue] ;
        usedspace = totalSpace - freeSpace ;
        // aggiorno la finestra di attesa
        stsresp = [NSArray arrayWithObjects:
                        [ NSNumber numberWithLong: 0 ],
                        [ NSNumber numberWithLong: myInfo.fileCount ],
                        [NSString stringWithFormat: @"Cataloging: %@",
                                        [[ NSFileManager  defaultManager]
displayNameAtPath: volPath] ],
                        nil];
```

```
        [ theCat updateInterfaceStr: stsresp ];
        // costruisco l'alberatura dei file a partire dalla scelta
           fInfo = [[ VolInfo alloc ] initTreeFromPath: volPath withCat:
theCat ];
        // metto a posto le altre informazioni
        [ fInfo setFileSize: usedspace ];
        [ fInfo setVolSize: totalSpace ];
        [ fInfo setVolFreeSize: freeSpace ];
        // aggiungo la cosa al catalogo
        [ theCat updateCatalog: fInfo ];
    }
    [ self stopCatalog ] ;
}
```

Devo giustificare due istruzioni che coinvolgono il catalogo; c'è il problema di evitare chiamate dell'Application Kit dal momento che sono dichiarate non sicure nei confronti dei thread. Allora, per aggiornare il contenuto della finestra di avanzamento lavori `WaitPanCtl`, invece che invocare direttamente i metodi, passo attraverso la classe `CdCatDoc` (che è eseguita nel thread principale), utilizzando i due metodi `updateInterfaceStr` e `updateCatalog`. Il primo metodo è direttamente responsabile dell'aggiornamento della finestra, mentre il secondo aggiunge i dati realtivi al volume al catalogo (ed aggiorna l'interfaccia di conseguenza...).

L'altra istruzione aggiunta è la costruzione di un oggetto della classe `NSAutoreleasePool`. Si tratta della lista degli oggetti che hanno ricevuto un messaggio di autorelease e sono destinati ad essere distrutti alla fine delle operazioni. Ogni thread e task e applicazione deve possederne uno; finora la cosa era gestita nascostamente dalla classe `NSApp`; qui, costruendo un thread, va scritto esplicitiamente.

Ho dovuto modificare i metodi `initTreeFromPath` di `FileStruct` e `initWithPath` di `CatFileInfo` aggiungendo un riferimento al catalogo come argomento. Così facendo è possibile, dall'interno di questi metodi, effettuare una chiamata di aggiornamento alla finestra con il metodo `updateInterfaceStr`. Ad esempio:

```
- (id)
initWithPath: (NSString*) aFile withCat: (CdCatDoc*) theCat
{
    ...
    // aggiorno la barra di attesa
    if ( theCat )
    {
        NSArray * stsresp = [NSArray arrayWithObjects:
                    [ NSNumber numberWithLong: CATSRV_UPDATECOUNT ],
                    [ NSNumber numberWithLong: CATSRV_NOVALUE ],
                    nil, nil];
        [ theCat updateInterfaceStr: stsresp ];
    }
    return ( self ) ;
}
```

Rimane da descrivere il metodo che chiude le operazioni di catalogazione:

```
- (void)
stopCatalog
{
```

```
    // faccio mettere a posto le cose al catalogo
    [ theCat endOfCatalog ];
    // rilascio il pool di memoria
    [ pool release ];
    // uccido il thread (servira'?)
    [ NSThread exit ] ;
}
```

Semplicemente, si segnala al catalogo che si ha finito, si distruggono tutti gli oggetti residui per poi suicidare il thread.

LANCIARE UN THREAD

Pensavo di dover fare molto lavoro per lanciare un thread in esecuzione, invece è di una semplicità disarmante:

```
- (void)
performAddFilesModal: ( NSArray *)volList
{
    // costruisco un vettore con gli argomenti di lancio del thread
     NSArray          * theArgs = [NSArray arrayWithObjects: self, volList,
nil];
    // costruisco la finestra di attesa
    waitSheet = [[ WaitPanCtrl alloc] init ] ;
    // apro la finestra e la rendo modale per la finestra
    [ NSApp beginSheet: [ waitSheet window ]
        modalForWindow: [ NSApp mainWindow ]
        modalDelegate: self
        didEndSelector: @selector( endOfCatalog )
        contextInfo: waitSheet
    ];
    // costruisco il server di catalogazione
    theCatServer = [ [[CatServer alloc] init] autorelease ];
    // lancio il server come nuovo thread
    [NSThread detachNewThreadSelector:@selector(makeVolCat:)
                    toTarget: theCatServer withObject: theArgs ];
    // e non ho piu' nulla da fare...
}
```

Si tratta di utilizzare la classe `NSThread` ed il metodo di costruzione di un thread. Occorre specificare tre argomenti: quale oggetto è il tenutario del thread, quale metodo deve essere eseguito e l'unico argomento di questo metodo (ecco giustificato lo `NSArray` visto in precedenza).

Ho anche trasformato la finestra di avanzamento lavoro da modale per tutta l'applicazione a modale per la sola finestra del catalogo, dal momento che non occorre più bloccare tutte le altre operazioni. Ho inserito un riferimento alla finestra costruito come variabile d'istanza, dal momento che possono esserci più catalogazioni in corso.

Infatti, per aggiornare il catalogo e la finestra ci sono i due metodi seguenti:

```
- (void)
updateCatalog: (VolInfo*) theCatVol
{
    // aggiungo il valume al catalogo
    [ dataSource addFileEntry: theCatVol ];
```

```
    // rinfresco le finestre
    [ coverWin refreshWindow ] ;
}

- (void)
updateInterfaceStr: (NSArray*) theArgs
{
    // ci sono fino a tre valori nello array: il valore della barra,
    // il valore massimo delal barra, la stringa da visualizzare
    int            numargs = [ theArgs count ] ;
    long        cv1 = [ [ theArgs objectAtIndex: 0] longValue] ;
    long        cv2 = [ [ theArgs objectAtIndex: 1] longValue] ;
    // aggiornamento del valore corrente della barra
    if ( cv1 == CATSRV_UPDATECOUNT )
    {
        // il valore speciale CATSRV_UPDATECOUNT indica
        // incremento del valore della barra
        double xx = [ waitSheet localDoubleValue ] ;
        [ waitSheet localSetDoubleValue: (xx+1) ];
    }
    // il valore speciale CATSRV_NOVALUE indica nessuna attivita'
    else if ( cv1 != CATSRV_NOVALUE )
        // impostazione diretta del valore delal barra
        [ waitSheet localSetDoubleValue: cv1 ];
    // aggornamento del massimo valore della barra
    // il valore speciale CATSRV_NOVALUE indica nessuna attivita'
    if ( cv2 != CATSRV_NOVALUE )
        [ waitSheet localSetMaxValue: cv2 ];
    // eventuale aggiornamento della stringa nella finestra
    if ( numargs > 2 )
    {
        // solo se ci sono almeno tre argomenti
        NSString    * cv3 = [ theArgs objectAtIndex: 2] ;
        [ waitSheet setMainText: cv3 ];
    }
    // verifico se si deve fermare la catalogazione
    if ( [ waitSheet stopMe ] )
    {
        [ theCatServer stopCatalog ] ;
    }
}
```

Il secondo metodo utilizza qualche accorgimento; avevo bisogno di tre argomenti (valore corrente e massimo della barra di avanzamento, e la stringa da visualizzare), ma li ho raccolti tutti in un `NSArray`; il terzo argomento può non essere presente, mentre ho utilizzato due valori illegali per discriminare operazioni specifiche sulla barra di avanzamento. In particolare mi occorreva un meccanismo per fare avanzare di un passo la barra dall'interno del thread di catalogazione (senza fare troppi giri di variabili), e quindi ho utilizzato un valore particolare (-1) per indicare questo tipo di aggiornamento. Se invece non aveva senso specificare parametri sulla barra, ma solo la stringa da visualizzare, utilizzavo un altro valore particolare (-2) per indicare nessuna operazione.

C'è infine il metodo chiamato alla chiusura delle operazioni di catalogazione:

```
- (void)
endOfCatalog
{
    // nascondo la finestra di attesa
    [[waitSheet window] setIsVisible: FALSE ];
    [ waitSheet setStopMe: NO ] ;
```

```
    // ho finito di lavorare modale con la sheet
    [NSApp endSheet: [waitSheet window] returnCode: nil ];
    // aggiorno il documento
    [ self updateChangeCount: NSChangeDone ] ;
    [ self refreshWinsWithListSts: @"" coverSts: @"" ];
}
```

Si chiude la finestra di avanzamento, si distrugge l'oggetto che ha eseguito la catalogazione, e si aggiorna l'aspetto delle finestre rimaste.

FERMATE QUELLA CATALOGAZIONE

Staccare la parte di catalogazione dal resto dell'applicazione consente la gestione responsiva dell'interfaccia utente, ed in particolare permette di poter intervenire ed interrompere il processo.

Ho così abilitato il pulsane di cancellazione della catalogazione sulla finestra di avanzamento, e ho associato una azione molto semplice, che si limita ad impostare una variabile d'istanza:

```
- (void)
stopThatTrain:(id)sender
{
    stopMe = TRUE ;
}
```

In questo modo, nel momento più opportuno, si può verificare se l'utente ha fatto clic sul pulsante di arresto della catalogazione, e terminare le operazioni in corso. Ho scelto di effettuare l'eventuale arresto in corrispondenza di un aggiornamento della finestra di avanzamento stessa, quindi all'interno del metodo `updateInterfaceStr` di `CdCatDoc`. In questo modo evito interruzioni anomale del thread e gestisco con tranquillità la cessazione delle operazioni.

Capitolo 064 - Progettare un salvaschermo

Ancora una deviazione dal consueto flusso di Macocoa (ho l'impressione che ce ne saranno sempre più; la catalogazione ha un po' smesso di stimolarmi...). Questa volta produco un salvaschermo, e non è per nulla difficile.

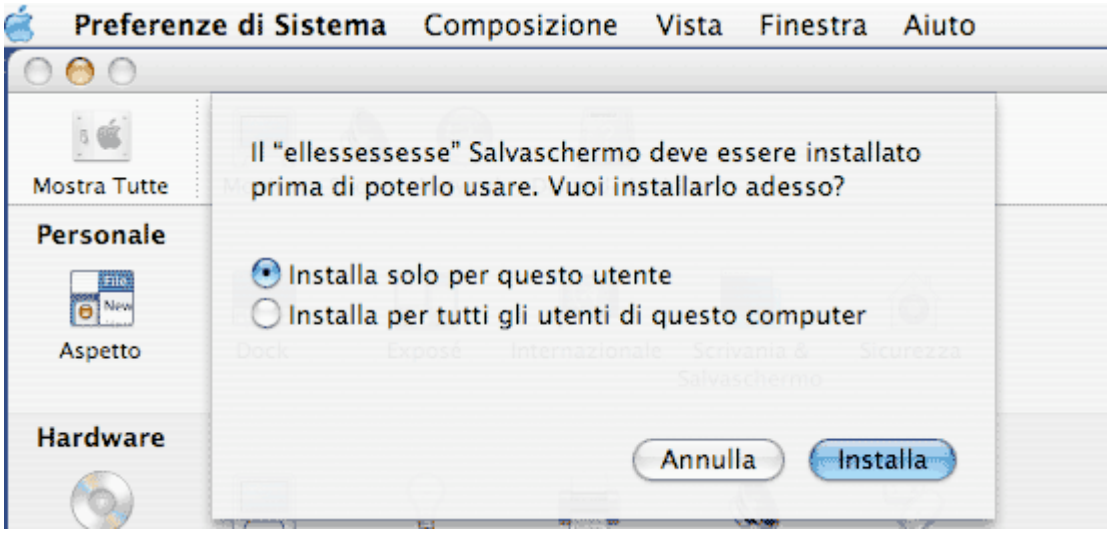
Sorgenti: documenti ed esempi apple

Stesura: 28 novembre 2004

SALVASCHERMO

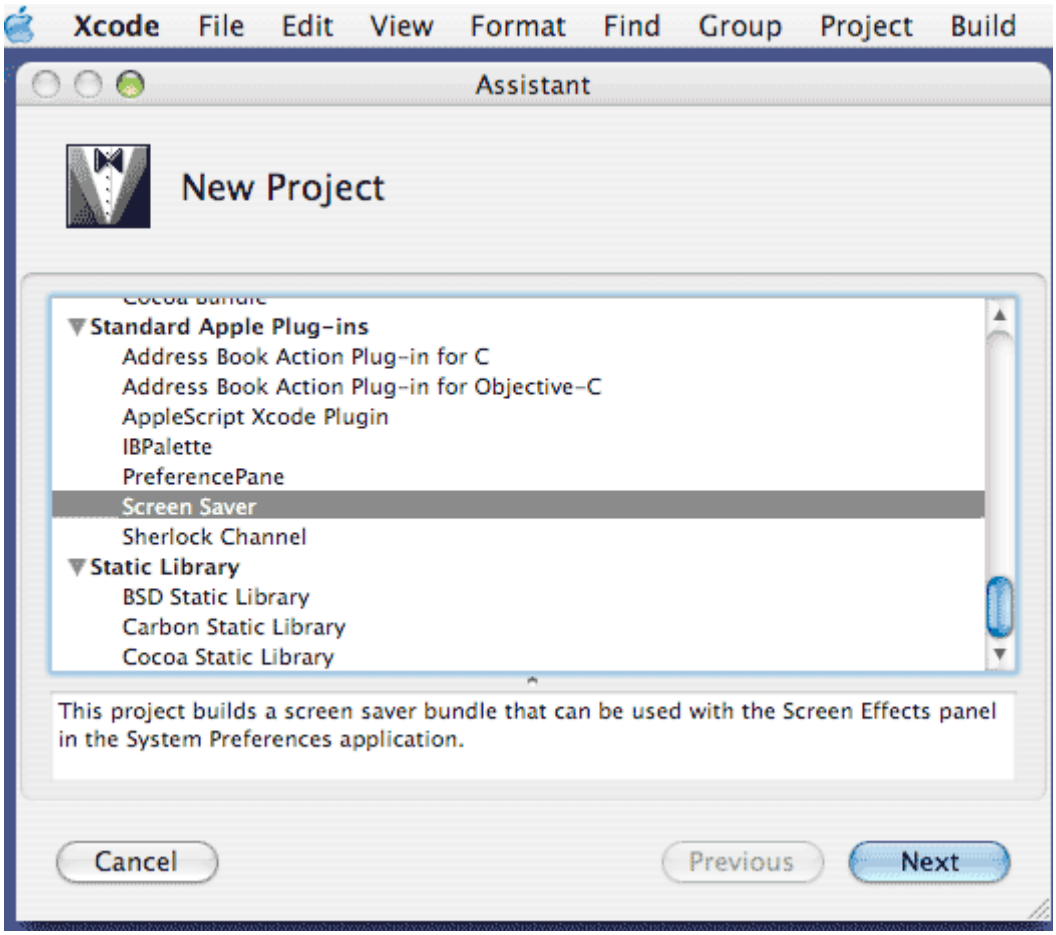
Un salvaschermo è un programma che prende possesso dello schermo dopo un certo periodo di inattività del computer. A parte le funzioni di preservare il funzionamento dei componenti fisici di uno schermo (cosa sempre meno utile, dal momento che spesso i monitor ormai si spengono da soli anche per conservare energia), forniscono talvolta intrattenimento e condivisione di risorse. In Mac OS X ci sono due tipi di salvaschermo: il primo è molto semplice, e consiste in uno *slide show*. Chiunque può produrne uno indicando una cartella colma di immagini. Qui mi interessa produrre un salvaschermo più sofisticato (si fa per dire), utilizzando Cocoa.

Dal punto di vista pratico, un salvaschermo è un *bundle*, ovvero un conglomerato di file, racchiusi all'interno di una cartella di estensione `.saver`. Questo bundle deve essere installato all'interno di una apposita cartella perché possa funzionare correttamente (`/Library/Screen Savers` oppure `~/Library/Screen Savers`, dove il carattere *tilde* indica la propria home directory o cartella d'inizio). Tuttavia, facendo doppio clic su di un bundle salvaschermo, il sistema operativo è così gentile da installarlo al posto giusto.



Cocoa mette a disposizione del programmatore un *framework* (collezione di oggetti, funzioni e quant'altro) per la progettazione di salvaschermi. si tratta di `ScreenSaver.framework`, che si trova ovviamente all'interno di

`/Library/Frameworks`. In realtà, non occorre preoccuparsi più di tanto di questi dettagli, perché XCode provvede da solo a fornirci tutto il necessario.



È infatti sufficiente generare un nuovo progetto, e scegliere il template opportuno nella lista sottoposta. Così facendo, XCode produce quasi tutto ciò che è necessario alla programmazione di un salvaschermo.

LA CLASSE SCREENSAVERVIEW

Il framework *ScreenSaver* consiste in una classe (`ScreenSaverView`) che fornisce i servizi di base; per scrivere un salvaschermo, se ne deve definire una sottoclasse e realizzare alcuni metodi necessari all'animazione ed alla configurazione del salvaschermo. Di fianco a questa classe, c'è una specifica sottoclasse di `NSUserDefaults` adatta alla gestione dei default dei salvaschermi (`ScreenSaverDefaults`) ed altre funzioni utili alla realizzazioni di salvaschermi (che infatti userò nel mio codice).

Per scrivere un salvaschermo, occorre dunque realizzare una sottoclasse di `ScreenSaverView`, ed in particolare sovrascrivere alcuni metodi. Fondamentalmente, i metodi da scrivere sono due:

```
- (id)            initWithFrame:(NSRect)frame isPreview:(BOOL)isPreview ;
- (void)        animateOneFrame ;
```

Il primo metodo è ovviamente chiamato quando il sistema operativo decide di far partire il salvaschermo. L'argomento booleano indica se il salvaschermo parte nella finestra di anteprima piuttosto che a tutto schermo; può essere utilizzato per differenziare il comportamento nei due casi.

Il secondo metodo è chiamato periodicamente ed è utilizzato per disegnare il contenuto dello schermo. Normalmente, il salvaschermo è una animazione, e quindi il metodo è utilizzato per disegnare un fotogramma di questa animazione.

A corredo di questi due metodi, ce ne sono molti altri per gestire le operazioni di contorno. Ad esempio, c'è un metodo per impostare la frequenza di aggiornamento, per eseguire operazioni poco prima o subito al termine delle operazioni, e due metodi per poter gestire una finestra di configurazione del salvaschermo.

L'INIZIALIZZAZIONE

Il salvaschermo che mi accingo a realizzare è piuttosto semplice, e ricalca un tipico salvaschermo che inspiegabilmente è assente da Mac OS X. L'idea è che il salvaschermo presenti a video una scritta mobile, definita dall'utente oppure che riporta l'ora corrente. Il movimento della scritta può essere scelto lineare, con rimbalzo sulle pareti laterali dello schermo, oppure può essere una passeggiata casuale (o dell'ubriaco). Per poter scegliere tra le varie opzioni, avrò bisogno di una finestra di configurazione, e di poter conservare le opzioni utente in un file di defaults.

Comincio con il lungo metodo di inizializzazione.

```
- (id)
initWithFrame:(NSRect)frame isPreview:(BOOL)isPreview
{
    // chiamo il metodo ereditato
    self = [super initWithFrame:frame isPreview:isPreview];
    // aggiusto le particolarita' di questo salvaschermo
    if (self)
    {
        NSSize                          requiredSize ;
        NSRange                          glyphRange ;
        int                              version ;
        ScreenSaverDefaults              * defaults ;
        // prelevo i default, se presenti
                                    defaults     =     [ScreenSaverDefaults
defaultsForModuleWithName:DEFAULTFILENAME];
```

Per conservare i valori di default tra successivi lanci del salvaschermo, si utilizza una sottoclasse specifica di `NSUserDefaults`. La specializzazione si rende necessaria per poter indicare il nome del file (che normalmente piglia il nome direttamente dall'applicazione, recuperandolo all'interno del file `Info.plist`). Per il resto, si utilizza in maniera identica, recuperando e salvando valori di default coi classici metodi.

Per essere sicuro dell'esistenza del file (e quindi evitare di caricare i valori da un file vuoto) verifico che uno dei valori estratti dal file abbia un valore sensato. Utilizzo allo scopo un campo di versione, utile anche in vista di successiva aggiornamenti del salvaschermo.

Gli altri valori conservati nelle preferenze sono il tipo di movimento (lineare o casuale), natura e colore del font utilizzato per la visualizzazione del testo, il tipo di testo da visualizzare (l'ora corrente piuttosto che un testo definito dall'utente) ed il testo stesso; infine, la massima velocità di movimento, in pixel per unità di tempo (il valore vero e proprio non sarà visualizzato all'utente, ma potrà qualitativamente sceglierlo attraverso uno slider).

```
        // uso un default 'versione' per verificare l'esistenza del file
        // oppure, un eventuale aggiornamento
        version = [defaults integerForKey: DEF_FILEPRESENT ];
        // se versione non e' zero, il file esiste gia'
        if ( version > 0 )
        {
            // leggo i parametri di default
            // tipo di movimento, font, tipo di testo, stringa e velocita'
            userSSType = [defaults integerForKey: DEF_USERSSTYPE ];
                    [ self setUserFont: [ NSFont fontWithName: [defaults
stringForKey: DEF_USERFONT ]
                                                size: [defaults floatForKey:
DEF_USERFONTSIZE ] ] ];
            [ self setUserColor: [ NSUnarchiver unarchiveObjectWithData:
                                    [defaults objectForKey: DEF_USERCOLOR ] ]
];
            userTextType = [defaults integerForKey: DEF_USERTEXTTYPE ];
            userString = [defaults stringForKey: DEF_USERSTRING ];
            userMaxSpeed = [defaults integerForKey: DEF_USERMAXSPEED ];
        }
        else
        {
            // predispongo valori di default cablati
            userSSType = SSTYPE_RANDOMWALK ;
             [ self setUserFont: [ NSFont fontWithName: @"Georgia" size: 48
] ];
            [ self setUserColor: [ NSColor redColor] ];
            userTextType = SSTXT_CURRTIME;
            userString = @"Macocoa Screen Saver";
            userMaxSpeed = 10 ;
            // predispongo i valori iniziali nelle preferenze
            [defaults setInteger: 1 forKey: DEF_FILEPRESENT];
            [defaults setInteger: userSSType forKey: DEF_USERSSTYPE];
              [defaults setObject: [userFont fontName] forKey: DEF_USERFONT
];
                        [defaults setFloat: [userFont pointSize] forKey:
DEF_USERFONTSIZE ];
              [defaults setObject: [ NSArchiver archivedDataWithRootObject:
userColor]
                forKey: DEF_USERCOLOR ];
            [defaults setInteger: userTextType forKey: DEF_USERTEXTTYPE ];
            [defaults setObject: userString forKey: DEF_USERSTRING ];
            [defaults setInteger: userMaxSpeed forKey: DEF_USERMAXSPEED];
            [defaults synchronize];
        }
```

Nel caso in cui il file dei default non sia presente, seleziono alcuni valori adatti alla prima partenza del salvaschermo, scrivendoli anche all'interno del file stesso. Ne segue che questa parte `else` del codice è eseguita una volta sola, al primo lancio in assoluto del salvaschermo per un dato utente (oppure, ogni volta che qualcuno cancella le preferenze...).

Adesso è tempo di costruire la struttura del salvaschermo.

La prima istruzione imposta l'intervallo di tempo tra due animazioni successive. Decido che tale tempo sarà di un decimo di secondo, per non caricare troppo il processore con attività poco redditizie. Per quanto riguarda il testo, decido di utilizzare per intero l'architettura di Cocoa, piuttosto che limitarmi alla singola `NSAttributedString`, in modo da sfruttare tutte le capacità di layout. Infatti, dopo tutta la sequenza di costruzione delle classi necessari, calcolo espressamente il rettangolo che racchiude il testo (o meglio, le sue dimensioni); queste dimensioni sono poi usate successivamente per cancellare solo la zona dello schermo interessata dal testo. Altrimenti, avrei dovuto cancellare tutta l'area dello schermo prima di disegnare nuovamente il testo in un'altra posizione. Poiché le operazioni non grafiche sono molto più veloci delle operazioni grafiche, cancellare solo il minimo indispensabile è più elegante.

```
        // predispongo i valori interni del salvaschermo
            // frequenza di aggiornamento: ogni freqValue secondi sara'
chiamato
        // il successivo metodo animateOneFrame
        [self setAnimationTimeInterval: freqValue ];
          // costruisco un textStorage (e conseguenti) per la gestione del
testo
        textStorage = [ [ NSTextStorage alloc ] initWithAttributedString:
            [ self prepareDspString ] ] ;
        layoutManager = [ [ [ NSLayoutManager alloc ] init ] autorelease ];
        [ textStorage addLayoutManager: layoutManager ];
        textContainer = [ [ [ NSTextContainer alloc ] init ] autorelease ];
        [ layoutManager addTextContainer:textContainer ];
        // calcolo una buona dimensione per il textContainer; prima dico
        // che la sua dimensione e' piuttosto grande
        [ textContainer setContainerSize: NSMakeSize( 10000, 10000 ) ];
        // poi effettuo il primo rendering della stringa
                glyphRange  =  [layoutManager  glyphRangeForTextContainer:
textContainer ];
         // estraggo l'effettivo rettangolo necessario per il disegno del
testo
                                        requiredSize    =    [layoutManager
usedRectForTextContainer:textContainer].size;
        // e lo attribuisco al textcontainer
        [ textContainer setContainerSize: requiredSize ];
        // aggiusto i parametri iniziali del movimento
        [ self newStartPoint ] ;
        // predispongo due filtri per velocita' ed angolo
        // utilizzati solo nel caso di random walk
        // per ridurre il nervosismo del movimento
        // filtro sulla velocita', taglia a 10 Hz
        InitIIR2Filter( & spdFlt, freqValue );
        spdFlt.filtType = FLTTYPE_DBLPOLE ;
        SetIIR2FilterParam( & spdFlt, 2, 1, 20 );
        SetIIR2FilterParam( & spdFlt, 2, 2, 1 );
        ConfigIIR2Filter( & spdFlt, ZERO );
        ResetIIR2Filter( & spdFlt, ZERO );
        // filtro sull'incremento dell'angolo, taglia a 20Hz
        InitIIR2Filter( & angFlt, freqValue );
        angFlt.filtType = FLTTYPE_DBLPOLE ;
        SetIIR2FilterParam( & angFlt, 2, 1, 20 );
        SetIIR2FilterParam( & angFlt, 2, 2, 1 );
        ConfigIIR2Filter( & angFlt, ZERO );
        ResetIIR2Filter( & angFlt, ZERO );
    }
```

```
    return self;
}
```

L'ultima parte del metodo inizializza i parametri del movimento. Il metodo `newStartPoint` calcola appunto i parametri iniziali del movimento (ci arrivo tra un attimo); seguono poi una serie di istruzioni piuttosto criptiche e che eviterò di spiegare. Mi limito a dire che qui costruisco due filtri numerici per smussare i movimenti casuali (quando il salvaschermo usa la passeggiata casuale); trovate il codice in due file `filter.c` e `filter.h`, scritti in puro C e che derivano da altri miei lavori in tutt'altro ambito. Poiché li avevo già pronti, puliti e funzionanti, li ho utilizzati senza troppe modifiche. La loro comprensione richiede troppo cultura che possa essere descritta in queste povere pagine, ma sono disposto a spiegazioni ulteriori a richiesta.

Il metodo `newStartPoint` utilizza altre funzioni specifiche per i salvaschermi:

```
- (void)
newStartPoint
{
    // zona occupata dal salvaschermo
    NSRect        vRect = [ self bounds ];
    // zona occupata dal testo
    NSSize        tSize = [ textContainer containerSize ] ;
    // una posizione casuale in x e y
    lastPosition.x = SSRandomFloatBetween( 0,
            vRect.origin.x + vRect.size.width - tSize.width ) ;
    lastPosition.y = SSRandomFloatBetween( 0,
            vRect.origin.y + vRect.size.height - tSize.height ) ;
    // un angolo casuale, evitando angoli troppo vicini a zero o a novanta
    // o multipli, che darebbero movimenti troppo banali
      lastAngle = M_PI * (SSRandomFloatBetween( 0.05, 0.45 ) + 0.5F *
SSRandomIntBetween(-2,1));
}
```

Spesso infatti si ha bisogno di avere dei numeri (pseudo) casuali per poter variare l'aspetto di un salvaschermo. Le funzioni `SSRandomFloatBetween` e `SSRandomIntBetween` producono appunto numeri casuali compresi tra i due argomenti della funzione. Utilizzo queste due funzioni per calcolare un punto iniziale di visualizzazione all'interno della vista, e per avere una direzione iniziale di movimento.

L'espressione che calcola lastAngle appare piuttosto complicata; in realtà cerca di evitare angoli troppo vicini a zero e 90 in modo da non avere movimenti (lineari) troppo banali.

IL MOVIMENTO

Prima di dettagliare il metodo di animazione, occorre spiegare come avviene il movimento.

Il primo tipo di movimento è lineare. Si parte con una direzione (un angolo compreso tra 0 e 360 gradi, o meglio, in radianti, tra meno pi greco e più greco) ed una velocità (cioé, un incremento di posizione nell'unità di tempo, dove l'unità di tempo è l'intervallo di aggiornamento del salvaschermo). Dalla posizione corrente ci si sposta nella direzione prescelta dell'ammontare della velocità, fino a raggiungere o superare

uno dei quattro bordi dello schermo. In questo caso, si rimbalza elasticamente indietro, come se la scritta fosse una palla di biliardo.

Il secondo tipo di movimento è la passeggiata casuale o, come preferisco dire, da ubriaco. Si parte da una posizione e con una certa direzione; da qui si generano due numeri casuali (sempre con le funzioni viste sopra). Il primo numero è l'incremento di posizione (un numero compreso tra 1 e la massima velocità specificata dall'utente); il secondo numero è la variazione della direzione, un numero compreso tra -90 e 90 (l'ubriaco cerca di mantenere un percorso più o meno in avanti), che si somma alla direzione corrente.

Ecco il metodo che esegue un passo dell'animazione:

```
- (void)
animateOneFrame
{
    float    newAngle, themargin ;
    float    newSpeed, tmpInc ;
    NSRect     rect;
    NSSize    limit ;
    NSRange glyphRange ;
    // in primo luogo, ripulisco la zona precedente
    [ self cleanLastText ];
    // ricalcolo le dimensioni della stringa da visualizzare
    [ textStorage setAttributedString: [ self prepareDspString ] ] ;
    [ textContainer setContainerSize: NSMakeSize( 10000, 10000 ) ];
     glyphRange = [layoutManager glyphRangeForTextContainer: textContainer
];
    limit = [layoutManager usedRectForTextContainer:textContainer].size;
    [ textContainer setContainerSize: limit ];
    // questo e' il rettangolo occupato dal salvaschermo
    rect = [self bounds] ;
```

Una buona animazione procede in questo modo: cancello il frame precedente, e poi disegno il nuovo frame. Meglio ancora, cancello solo la porzione del frame precedente che è cambiata, e poi scrivo il nuovo frame. La prima istruzione del metodo serve quindi a cancellare la zona di schermo occupata dal testo nel frame precedente. Di seguito, si produce la nuova stringa (il codice è inefficiente, in quanto la maggior parte delle volte la stringa non è cambiata: cambia solo nel caso di visualizzazione dell'ora corrente, e solo una volta su dieci. Ma non avevo voglia di complicarmi la vita), e si ricalcola lo spazio occupato da questa stringa (utile al passo di animazione successivo).

Adesso il codice si biforca per tenere conto delle due possibilità di movimento:

```
    // calcolo una nuova posizione, in dipendenza dal tipo di movimento
    switch ( userSSType ) {
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    case SSTYPE_RANDOMWALK :    // passeggiata da ubriaco
        // variazione dell'angolo, piu' o meno in avanti
        tmpInc = M_PI * SSRandomFloatBetween( -0.5, 0.5 );
        // filtro le asperita' del movimento
        newAngle = CalcIIR2Filter( & angFlt, tmpInc ) ;
        // calcolo la nuova direzione di movimento
        lastAngle += newAngle ;
        // spostamento, casuale tra 1 e massimo
        tmpInc = SSRandomFloatBetween( 1, (float) userMaxSpeed );
        // evito accelerazioni troppo brusche
```

```
        newSpeed = CalcIIR2Filter( & spdFlt, tmpInc ) ;
        // calcolo la nuova posizione aggiungendo lo spostamento
        lastPosition.x += newSpeed * cos ( lastAngle ) ;
        lastPosition.y += newSpeed * sin ( lastAngle ) ;
        // costringo la posizione all'interno del rettangolo massimo
        // se la posizione supera i limiti, rimango a contatto col margine
        // (l'ubriaco incontra il muro, e ci rimane attaccato...)
        if ( lastPosition.x + limit.width > rect.origin.x + rect.size.width
)
             lastPosition.x = rect.origin.x + rect.size.width - limit.width
;
        if ( lastPosition.x < rect.origin.x )
            lastPosition.x = rect.origin.x ;
         // credo che si debba tenere conto anche delle caratteristiche del
font
        tmpInc = [ userFont ascender ] + [ userFont descender ] ;
        themargin = lastPosition.y + tmpInc + limit.height ;
        if ( themargin > (rect.origin.y + rect.size.height) )
            lastPosition.y = rect.origin.y + rect.size.height
                - limit.height - tmpInc ;
        themargin = lastPosition.y + tmpInc ;
        if ( themargin < rect.origin.y )
            lastPosition.y = rect.origin.y - tmpInc ;
        break ;
```

Siamo nel caso della passeggiata casuale; genero un incremento di direzione (un angolo compreso tra -90 e 90 gradi), lo filtro per evitare cambi di direzione troppo bruschi, e poi ottengo la nuova direzione di movimento. Produco poi un nuovo spostamento, compreso tra un pixel ed il massimo numero previsto, filtro nuovamente per evitare accelerazioni eccessive; finalmente, arrivo a calcolare una nuova posizione del testo. Questa posizione però potrebbe essere tale da portare la rappresentazione del testo al di fuori dello spazio ammissibile. Vado quindi a correggere la posizione, facendo in modo che il testo sbatta contro il bordo, e lì rimanga finché la casualità non l'allontana nuovamente in direzione opposta. Nel correggere la posizione, devo tenere conto dello spazio occupato dalla scritta: uso quindi le dimensioni calcolate dalla classe `NSLayoutManager` per i margini destro e superiore. Devo tenere conto anche delle caratteristiche del font (fatto riconosciuto dopo lungo penare, come al solito). Non sono per nulla convinto della correttezza del mio codice (anzi, secondo me ho sbagliato in pieno); tuttavia, produce risultati accettabili (e, come si dice, quando va che tanto basta, non toccare, ché si guasta). Ad ogni modo, alla posizione corrente, a partire dalla quale verrà disegnato il testo, aggiungo un paio di contributi (ascent e descent) tipiche del font.

```
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    case SSTYPE_LINEAR :    // movimento lineare con rimbalzo
        // mantengo direzione e spostamento
        lastPosition.x += userMaxSpeed * cos ( lastAngle ) ;
        lastPosition.y += userMaxSpeed * sin ( lastAngle ) ;
        // gestione del rimbalzo
        // se vado troppo a destra, rimbalzo indietro verso sinistra
        themargin = lastPosition.x + limit.width ;
        if ( themargin > rect.origin.x + rect.size.width )
        {
            // spazio che manca da percorrere
             float    delta = themargin - (rect.origin.x + rect.size.width)
;
            // aggiorno la posizione
                    lastPosition.x  =  (rect.origin.x  +  rect.size.width  -
```

```
limit.width) - delta ;
            // cambia l'angolo di movimento
            lastAngle = M_PI - lastAngle ;
        }
        // se vado troppo a sinistra, rimbalzo verso destra
        // (c'e' una metafora politica in tutto cio'?)
        if ( lastPosition.x < rect.origin.x )
        {
            // spazio che manca da percorrere
            float    delta = rect.origin.x - lastPosition.x ;
            // aggiorno la posizione
            lastPosition.x = rect.origin.x + delta ;
            // cambia l'angolo di movimento
            lastAngle = M_PI - lastAngle ;
        }
        // se vado troppo in alto, rimbalzo in basso
         // credo che si debba tenere conto anche delle caratteristiche del
font
        tmpInc = [ userFont ascender ] + [ userFont descender ] ;
        themargin = lastPosition.y + tmpInc + limit.height ;
        if ( themargin > (rect.origin.y + rect.size.height) )
        {
             float    delta = themargin - (rect.origin.y + rect.size.height)
;
                    lastPosition.y  =  rect.origin.y  +  rect.size.height  -
limit.height
                   - delta -tmpInc ;
            lastAngle = - lastAngle ;
        }
        // se vado troppo i nbasso, rimbalzo in alto
        themargin = lastPosition.y + tmpInc ;
        if ( themargin < rect.origin.y )
        {
            float    delta = rect.origin.y - themargin ;
            lastPosition.y = rect.origin.y + delta - tmpInc ;
            lastAngle = - lastAngle ;
        }
        break ;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    default :    // non dovrei mai capitare qui...
        NSLog( @"Non dovrei mai scrivere questo, ma non si sa mai - 1");
        break ;
    }
    // riporto la direzione all'interno dell'angolo fondamentale
    while ( lastAngle < - M_PI )    lastAngle += 2 * M_PI ;
    while ( lastAngle > M_PI )    lastAngle -= 2 * M_PI ;
    // scrivo di nuovo la stringa, nella nuova posizione
            [   layoutManager   drawGlyphsForGlyphRange:glyphRange   atPoint:
lastPosition ];
}
```

A questo punto l'altro tipo di movimento, quello lineare, è solo leggermente più complicato. Aggiorno la posizione aggiungendo lo spostamento previsto, e poi verifico se ho raggiunto e superato uno dei bordi. Nel caso, devo cambiare la direzione in modo da simulare un rimbalzo elastico. Allo scopo, ci sono due elementi da tenere in considerazione: lo spazio e l'angolo. Comincio con l'angolo che è più facile. In un rimbalzo elastico l'angolo di incidenza deve essere uguale all'angolo di riflessione; a seconda del bordo il nuovo angolo si ottiene pigliando l'angolo piatto e sottraendo l'angolo di incidenza, oppure semplicemente rovesciando il segno. Per lo spazio rimasto da percorrere, piglio la differenza tra la nuova posizione ed il bordo; questa differenza va sottratta alla coordinata del bordo per ottenere la posizione riflessa. La cosa è

complicata vieppiù dal fatto che bisogna considerare lo spazio occupato dal testo e le onnipresenti caratteristiche del font.

Finalmente, l'ultima istruzione del metodo disegna il testo nella nuova posizione.

Mancano un paio di metodi per completare il disegno del testo.

Il metodo per cancellare lo spazio occupato in precedenza:

```
- (void)
cleanLastText
{
    NSRect     rect;
         NSSize         limit   =   [layoutManager   usedRectForTextContainer:
textContainer ].size;
        //  calcolo  il  rettangolo  da  cancellare;  parto  dalla  posizione
precedente
    // ma devo tenere conto delle caratteristiche del font
    rect = NSMakeRect( lastPosition.x,
        lastPosition.y + [ userFont ascender ] + [ userFont descender ],
        limit.width, limit.height );
    [[NSColor blackColor] set];
    NSRectFill( NSInsetRect( rect, -2, -2) );
}
```

Qui utilizzo la posizione corrente e le dimensioni del testo per costruire un rettangolo, che riempio di nero. Ancora una volta, le caratteristiche del font aiutano a calcolare meglio il rettangolo.

Infine, il metodo che prepara il testo:

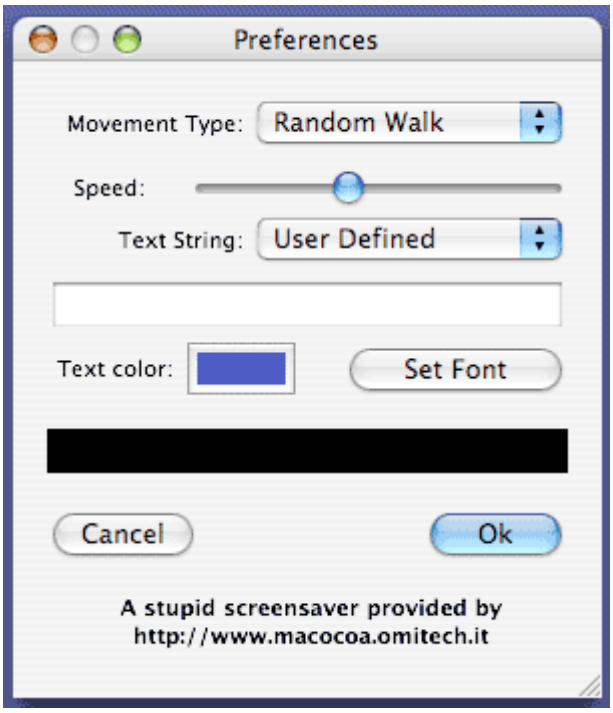
```
- (NSMutableAttributedString *)
prepareDspString
{
    NSMutableAttributedString    * thestring ;
    NSMutableDictionary        * txtAtt ;

    switch ( userTextType ) {
    case SSTXT_CURRTIME :    // ora corrente
        thestring = [ [NSMutableAttributedString alloc] initWithString:
                [  [NSDate  date]  descriptionWithCalendarFormat:@"%H:%M:%S"
timeZone:nil locale:nil] ];
        break ;
    case SSTXT_USERTEXT :    // testo stabilito dall'utente
          thestring = [ [NSMutableAttributedString alloc]  initWithString:
userString ];
        break ;
    default :    // non dovrei mai capitare qui...
        NSLog( @"Non dovrei mai scrivere questo, ma non si sa mai - 2");
        break ;
    }
    // inizializzo un dizionario per contenere gli attributi
          txtAtt   =   [[[NSMutableDictionary   alloc]   initWithCapacity:2]
autorelease];
    [ txtAtt setObject: userFont forKey: NSFontAttributeName ];
    [ txtAtt setObject: userColor forKey: NSForegroundColorAttributeName ];
      [  thestring  setAttributes:  txtAtt  range:  NSMakeRange(0,  [thestring
length])];
    return ( thestring );
```

```
}
```

Qui non c'è molto di nuovo rispetto a quanto visto nei precedenti capitoli di Macocoa. A seconda del tipo di testo da visualizzare, costruisco una `NSMutableAttributeString`, attribuisco font e colore, e la restituisco indietro.

CONFIGURAZIONE DEL SALVASCHERMO

La parte disegno del salvaschermo è adesso completa. C'è da scrivere tutta la sezione di configurazione. A parte un paio di dettagli, non è che ci siano grandi concetti. Si devono sovrascrivere due metodi della classe `SceenSaverView`.



Il primo metodo serve ad indicare al gestore dei salvaschermo se è presente o meno un pannello di configurazione. La versione di default di questo metodo dice di no, e quindi tutto si ferma lì. Qui invece il pannello è presente. Occorre allora scrivere un secondo metodo che restituisca la finestra che appunto presenta l'interfaccia di configurazione.

Ecco i due metodi necessari

```
- (BOOL)hasConfigureSheet
{
    // c'e' la finestra, ritorno YES
    return YES;
}

- (NSWindow*)configureSheet
{
    NSString    * thestr ;
    // se la finestra non e' presente, la carico dal file nib
    if (! prefSheet)
        [NSBundle loadNibNamed:@"prefSheet" owner:self];
    // questo e' un buon posto per predisporre la finestra in modo
    // che mostri i valori correnti
```

```
    [ ssType selectItemAtIndex: (userSSType - SSTYPE_BASEVAL4MENU) ] ;
    [ textType selectItemAtIndex: (userTextType - SSTXT_BASEVAL4MENU) ] ;
    if ( userTextType == SSTXT_USERTEXT )
        [ displayText setEnabled: YES ] ;
    else
        [ displayText setEnabled: NO ] ;
    [ displayText setStringValue: userString ] ;
    [ textColor setColor: userColor ];
    [ travelSpeed setIntValue: userMaxSpeed ] ;
    // aggiusto la visualizzazione del font di esempio
    thestr = [ NSString stringWithFormat: @"%@ %4.1f",
                    [ userFont displayName ], [ userFont pointSize ] ] ;
    // il nome del font lo imposto con font e col colore prescelti
    [ choosenfont setTextColor: userColor ] ;
    [ choosenfont setStringValue: thestr ];
    return ( prefSheet );
}
```

Il secondo metodo, come si può facilmente capire leggendo il codice, carica la finestra dal nib che mi sono premunito di costruire. Utilizzando poi i valori di default presenti, predispone i vari controlli della finestra in maniera da rispecchiare lo stato corrente.

Un punto interessante è che il proprietario di questo nib (e della finestra contenuta) non è come al solito una classe `NSWindowController`, ma una classe `NSView`; in effetti non ci sono particolari differenze di trattamento, basta aggiungere gli opportuni outlet verso i controlli, realizzare le action che i controlli attivano, proprio come una normale classe controllore di finestre.

Per il resto, è la solita ordinaria amministrazione. Quando si chiude la finestra facendo clic sul pulsante di *Ok*, si devono effettuare le modifiche ai valori di default, ed aggiornarli nelle preferenze:

```
- (IBAction)
prefSheetSave:(id) sender
{
    ScreenSaverDefaults            * defaults ;
    // recupero la stringa utente
    [ self setUserString: [ displayText stringValue ]] ;
    // chiudo la finestra
    [NSApp endSheet:prefSheet];
    // aggiorno i defaults
                               defaults        =        [ScreenSaverDefaults
defaultsForModuleWithName:DEFAULTFILENAME];
    if ( [defaults integerForKey: DEF_FILEPRESENT ] <= 0 )
        [defaults setInteger: 1 forKey: DEF_FILEPRESENT];
    [defaults setInteger: userSSType forKey: DEF_USERSSTYPE];
    [defaults setObject: [userFont fontName] forKey: DEF_USERFONT ];
    [defaults setFloat: [userFont pointSize ] forKey: DEF_USERFONTSIZE ];
         [defaults  setObject:  [  NSArchiver  archivedDataWithRootObject:
userColor]
        forKey: DEF_USERCOLOR ];
    [defaults setInteger: userTextType forKey: DEF_USERTEXTTYPE ];
    [defaults setObject: userString forKey: DEF_USERSTRING ];
    [defaults setInteger: userMaxSpeed forKey: DEF_USERMAXSPEED ];
    [defaults synchronize];
}
```

Se invece si fa clic sul pulsante *Cancel*, ci si limita a chiudere la finestra

```
- (IBAction)
prefSheetCancel:(id) sender
{
    [NSApp endSheet:prefSheet];
}
```

In entrambi i casi la finestra va chiusa esplicitamente, proprio perché il suo proprietario non è un `NSWindowController`, e quindi non eredita alcune funzionalità.

Tutti i controlli che hanno un effetto diretto sul salvaschermo chiamano il seguente metodo come action, metodo che aggiusta i valori interni dei parametri:

```
- (IBAction)
updateWindow:(id) sender
{
    int        thetag = [ sender tag ] ;
    switch ( thetag ) {
    case 1 :    // tipo di movimento
          userSSType = [[ ssType selectedItem ] tag] - SSTYPE_BASETAG4MENU
;
        break ;
    case 2 :    // velocita' di movimento
        userMaxSpeed = [ travelSpeed floatValue ] ;
        break ;
    case 3 :    // cambio tipo di testo
         userTextType = [[ textType selectedItem ] tag] - SSTXT_BASETAG4MENU
;
        if ( userTextType == SSTXT_USERTEXT )
        {
            [ displayText setEnabled: YES ] ;
            [ displayText setStringValue: userString ] ;
        }
        else
        {
            [ displayText setEnabled: NO ] ;
        }
        break ;
    case 5 :    // colore del testo
        [ self setUserColor: [ textColor color] ];
        [ choosenfont setTextColor: userColor ] ;
        break ;
    default :
        break ;
    }
    [ self cleanLastText ];
}
```

Un controllo che non passa per questo metodo è il campo di testo dove l'utente specifica il testo a visualizzare, che è gestito direttamente dal metodo prefSheetSave.

Un altro controllo è il pulsante che permette di selezionare il font, che apre direttamente il pannello di scelta font:

```
- (IBAction)
selectFont:(id) sender
{
    [ [ NSFontManager sharedFontManager] orderFrontFontPanel: nil ];
}
```

A sua volta, il pannello attiva come action il metodo `changeFont`, che ho così realizzato:

```
- (void)
changeFont:(id)sender
{
    NSString    * thestr ;
    // ottengo il nuovo font
    NSFont        * newFont = [sender convertFont:userFont] ;
    // lo imposto internamente
    [ self setUserFont: newFont ];
    // aggiorno la stringa visualizzata
        thestr  =  [  NSString  stringWithFormat:  @"%@  %4.1f",  [  newFont
displayName ], [ newFont pointSize ] ] ;
    // e cambia la visualizzazione della stessa in accordo al nuovo font e
colore
    newFont = [ NSFont fontWithName: [userFont fontName] size: 12 ] ;
    [ choosenfont setFont: newFont ] ;
    [ choosenfont setStringValue: thestr ];
    // pulisco il testo attuale
    [ self cleanLastText ];
    return;
}
```

Ho infatti pensato che fosse cosa carina (ed educativa) fare in modo che una stringa con il nome del font prescelto fosse rappresentata autoreferenzialmente (cioé, con lo stesso font che indica). Così facendo, ho scoperto la differenza tra il metodo `displayName` (produce una stringa adatta alla visualizzazione verso un utente umano) e `fontName` (produce una stringa più adatta ad un utente elettronico).

Infine, un trucco per permettere all'utente di interagire con il salvaschermo, e cambiarne alcune proprietà mentre è in esecuzione:

```
-(void)
keyDown:(NSEvent *)theEvent
{
    // recupero i caratteri associati all'evento
    NSString *characters = [theEvent characters];
    // se ci sono effettivamente caratteri
    if ([characters length])
    {
        // ricavo il primo carattere
        unichar character = [characters characterAtIndex:0];
        // ed eseguo quanto richiesto
        switch (character) {
        case 'r':    // riparto con nuovi valori di movimento
            [ self cleanLastText ];
            [ self newStartPoint ];
            break ;
        case 't':    // cambio tipo di salvaschermo
            if ( userSSType == SSTYPE_RANDOMWALK )
                userSSType = SSTYPE_LINEAR ;
            else if ( userSSType == SSTYPE_LINEAR )
                userSSType = SSTYPE_RANDOMWALK ;
            // ...
            [ self cleanLastText ];
            [ self newStartPoint ];
            break ;
        default :
```

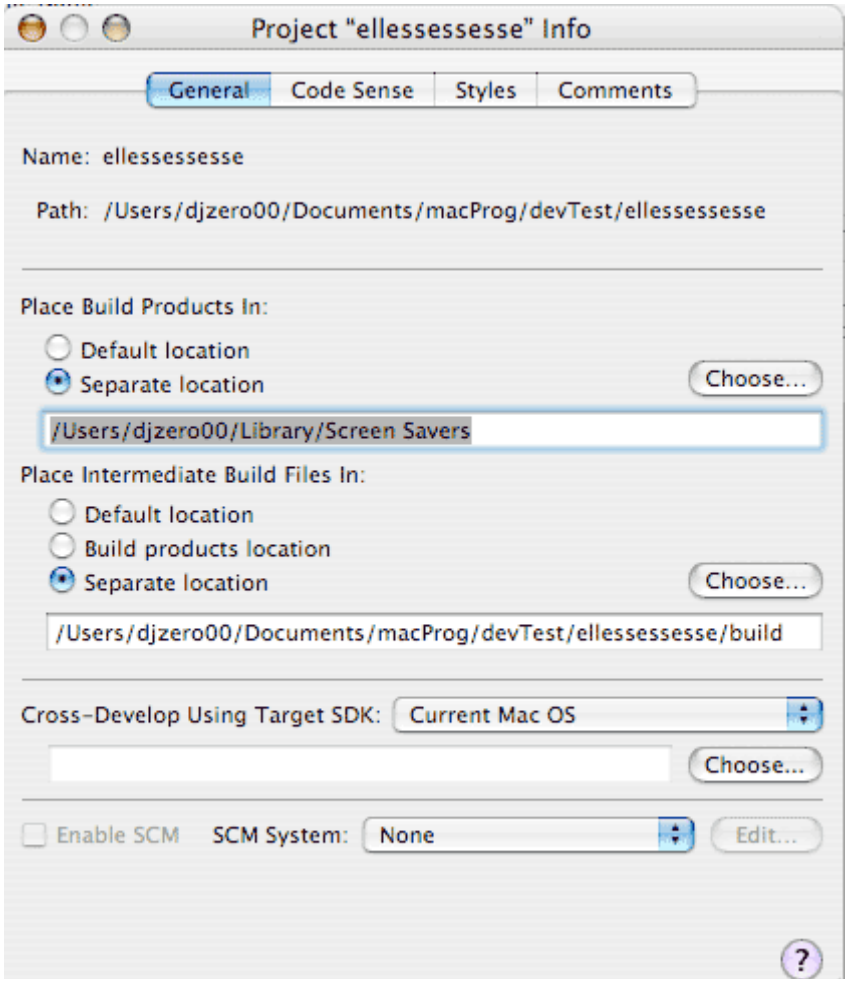
```
                break ;
            }
        }
}
```

Il metodo intercetta ogni pressione di tasto, e tratta in maniera particolare un paio di caratteri; se l'utente preme il tasto 'r', il salvaschermo riparte daccapo, ma con lo stesso tipo di movimento. Con il tasto 't', invece, il movimento cambia natura (da casuale diventa lineare e viceversa), ed ovviamente riparte daccapo.

DEBUG DI UN SALVASCHERMO

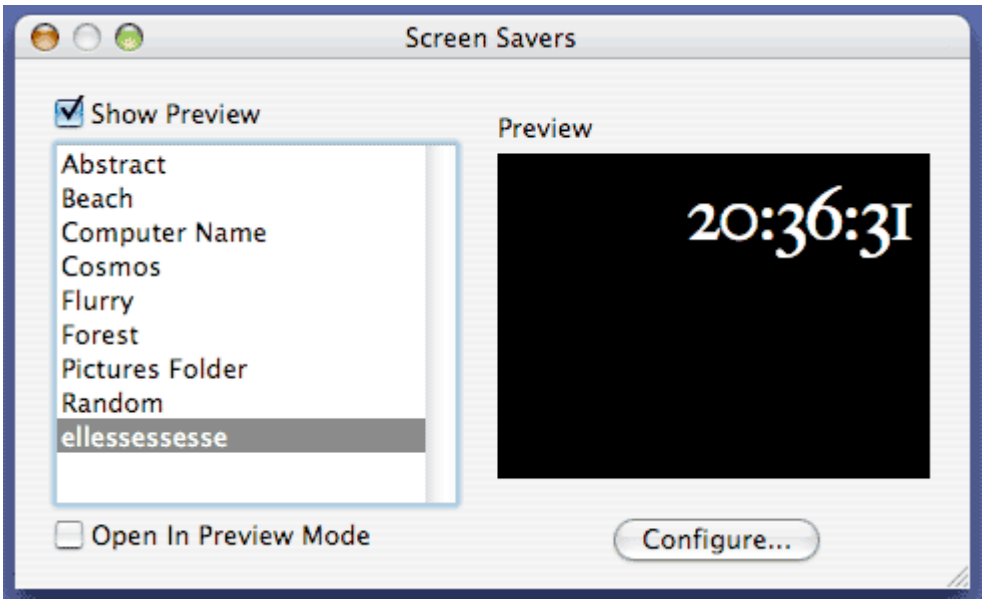
Un problema che si è presentato abbastanza presto nello sviluppo di questo salvaschermo è stato come eseguire il debug. Infatti, il processo di compilazione produce un file `.saver`, che non è una applicazione. Per provare il funzionamento, non è possibile utilizzare i comandi *Run* e *Debug* presenti in XCode. Il mio primo tentativo era allora di compilare, passare da XCode al Finder, fare doppio clic sul risultato della compilazione, installare il salvaschermo, e finalmente, all'interno dell'applicazione *Preferenze di Sistema*, eseguire il salvaschermo.

Un primo passo è stata l'eliminazione del passo di installazione. Si può fare in modo che XCode costruisce il salvaschermo in una directory che non è quella di default (all'interno della cartella del progetto), ma a scelta. Ho quindi scelto come cartella destinazione quella dei salvaschermi utente.



Si tratta d modificare il percorso di default nel tab *General* della finestra delle informazioni del progetto. Poi, sfruttando quella immensa mole di informazioni che è la mailing list degli sviluppatori Cocoa, ho trovato una brillante soluzione, anzi due.

In effetti è possibile eseguire le operazioni di Run e Debug di un salvaschermo direttamente all'interno di XCode (in altre parole: si può utilizzare il debugger!); si tratta di costruire un nuovo *Custom Executable* (c'è una voce apposita nel menu *Project* di XCode) e di stabilire l'applicazione Preferenze di Sistema come applicazione da utilizzare. Così facendo, il comando Run di Xcode fa partire appunto questa applicazione (e così salto un ulteriore passo). Addirittura, inserendo dei breakpoint nel codice del salvaschermo, con il comando Debug è possibile eseguire passo passo il codice del salvaschermo (ovviamente, solo la parte specifica del salvaschermo in via di costruzione, non essendo disponibile il sorgente di Preferenze di Sistema).



C'è addirittura un passaggio ulteriore; esiste una applicazione dal nome di SaverLab, open source, che fornisce un ambiente dove eseguire sperimentazioni con i salvaschermi. Tra le cose interessanti, la possibilità di rallentare o accelerare l'animazione, utilizzare finestre di varie dimensioni, avere più istanze contemporanee del salvaschermo, produrre un filmato quicktime con l'animazione eseguita dal salvaschermo. A parte qualche idiosincrasia minore (cose che funzionano con Preferenze di Sistema e non funzionano con SaverLab), è decisamente un programma da avere quando si sviluppano salvaschermi.

Capitolo 065 - Cocoa e Unix

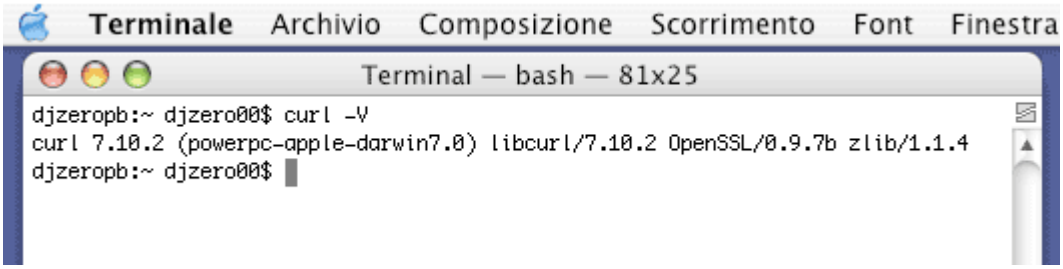
In questo capitolo mi ingegno a far interagire un programma Cocoa con Unix. Nella fattispecie, esamino un comando per scaricare file da internet, e lo incorporo all'interno di una interfaccia grafica in Cocoa (piuttosto brutta, va detto).

Sorgente: copio un esempio Apple (Moriarity), e non solo.

Prima stesura: 10 dicembre 2004.

CURL

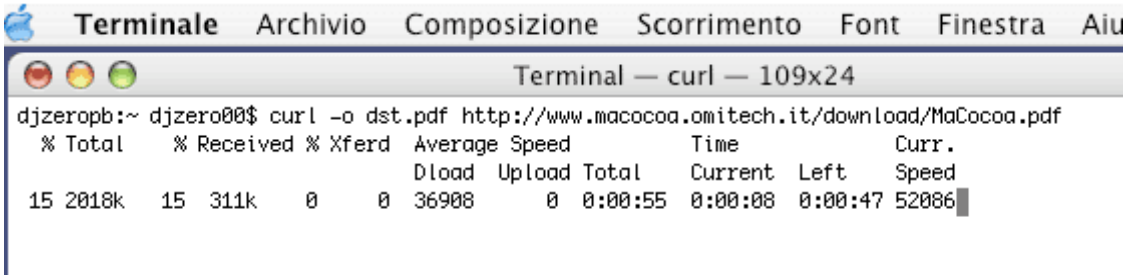
Curl, che i più esperti scrivono cURL, è un programma per effettuare trasferimenti di file tramite internet, utilizzando diversi protocolli quali HTTP, FTP, e molti altri. Si tratta di un programma a linea di comando, ed è disponibile per molti sistemi operativi. In particolare, è già installato all'interno di Mac OS X (il sito afferma a partire dalla versione 10.1); è un progetto open source, e sono quindi disponibili i sorgenti.



Per poterne vedere gli effetti, occorre normalmente aprire una finestra con l'applicazione Terminale, e scrivere il comando apposito, `curl`, corredato da varie opzioni.

La cosa più semplice che si può fare con curl è scaricare un file da internet, ad esempio con il protocollo http. La sintassi in questo caso è la seguente:

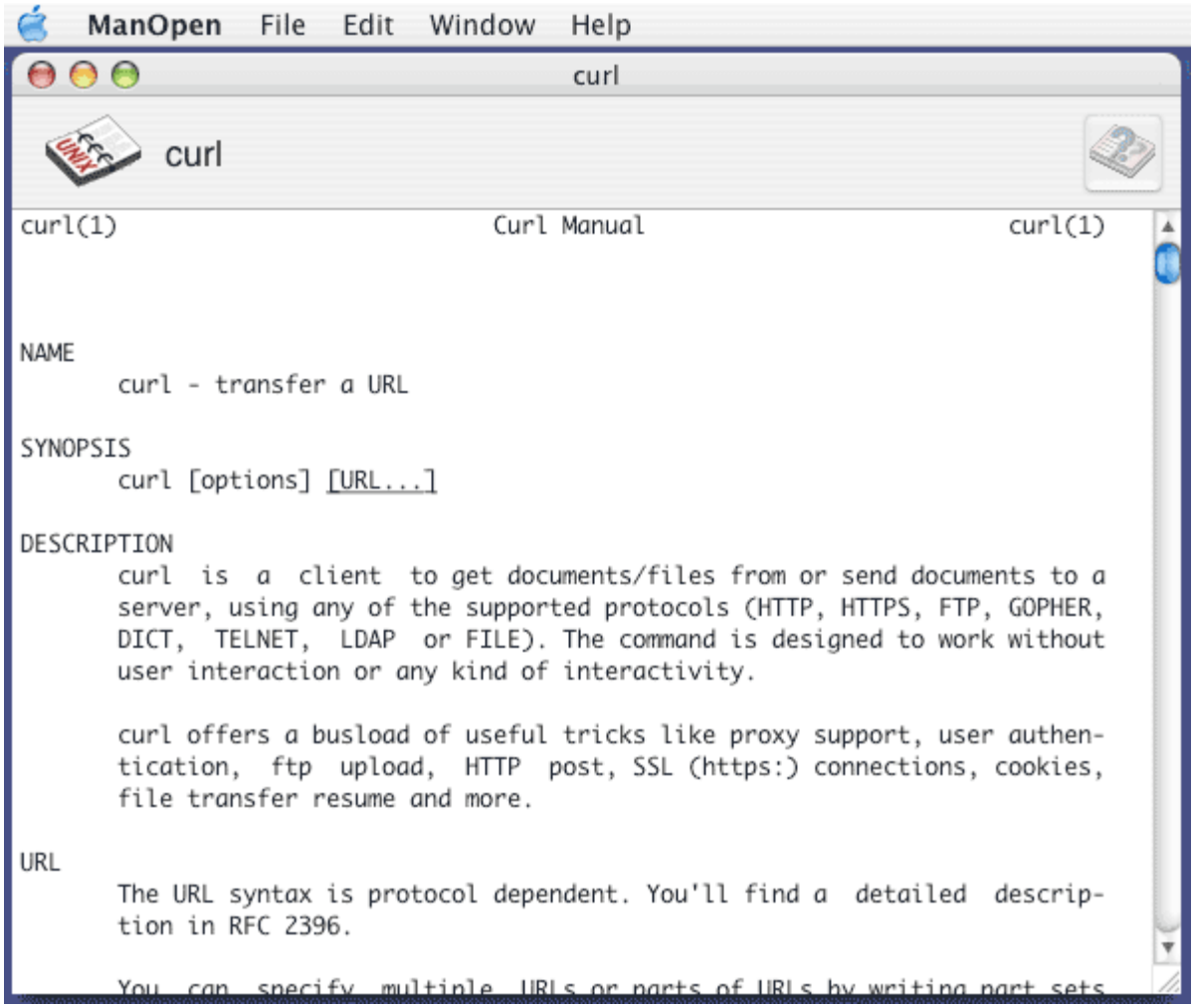
```
curl -o file-destinazione http://url-file-da-scaricare
```



Ad esempio, volendo scaricare i capitoli 1-31 di Macocoa, si può scrivere:

```
curl -o dst.pdf http://www.macocoa.omitech.it/download/MaCocoa.pdf
```

Tante e variegate sono le opzioni e le funzioni messe a disposizione da cURL.

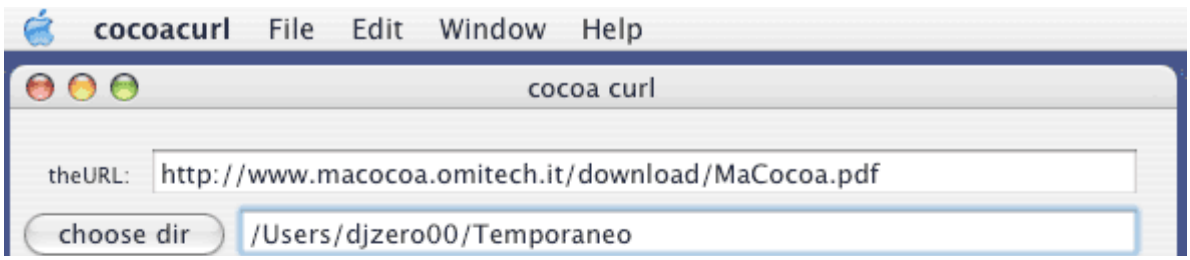


Si possono imparare leggendo banalmente il manuale messo a disposizione in linea, tramite il comando unix man o più comodamente con programmi quali ManOpen, o direttamente dall'interno di XCode (c'è una apposita voce sotto il menu *Help*) ; per il momento, mi interessano solo le funzioni base, visto che intendo focalizzare l'attenzione altrove.

L'idea è infatti di fornire una interfaccia grafica per questo programma, in grado di evitare ad un utente poco smaliziato l'apertura del Terminale e la scrittura dei comandi necessari.

L'INTERFACCIA

Definisco un nuovo progetto; è una semplice *Cocoa-Application*, dal momento che non saranno gestiti documenti. Per il momento, l'interfaccia è di una bruttezza unica, tanto da far rimpiangere la linea di comando.



C'è un capo di testo dove scrivere l'indirizzo del file da trasferire, un campo dove indicare la directory locale dove salvare il file; la directory di destinazione può essere scelta attivando il pulsante *Choose Dir*, che attiva un pannello di selezione. Quando i due campi sono pronti, si può attivare il trasferimento con il pulsante *Get File*. Un ulteriore campo di testo mostra il risultato del comando.

Ho definito una classe per la gestione dell'interfaccia, chiamandola `curlCtrl`. Qui sono presenti gli outlet verso i vari controlli ed i vari campi di testo, ed i metodi che rappresentano le action attivate dai controlli stessi. Effettuo tutti i collegamenti necessari in *Interface Builder*, e poi comincio a scrivere un po' di codice.

All'apertura della finestra è eseguito il metodo `awakeFromnib`

```
- (void)
awakeFromNib
{
    // url di default
    [ theUrl setStringValue: THEFILENAME ] ;
    [ self setUrl2download: [ theUrl stringValue ]];
    // directory destinazione di default
     [ self setSaveFilePath: [ NSHomeDirectory( ) stringByAppendingString:
STARTDIR ]];
    [ theDst setStringValue: saveFilePath ] ;
}
```

Con questo metodo predispongo dei valori di default per i campi che contengono lo URL e la directory di destinazione. In questo secondo caso utilizzo la funzione `NSHomeDirectory` per ottenere una `NSString` che rappresenta il percorso della *home directory* dell'utente (cui poi aggiungo una sotto directory giusto per mia comodità).

Curioso il metodo per selezionare la directory (lanciato da un clic sul pulsante *Choose Dir*) dove salvare il file trasferito, che utilizza, contrariamente alle credenze popolari, un pannello di apertura anziché di salvataggio.

```
- (IBAction)
nameDstFile:(id)sender
```

```
{
    // per scegliere uan directory, paradossalmente si usa il pannello
    // per scegliere un file... opportunamente configurato
    int             risposta ;
    NSOpenPanel     *sPanel = [ NSOpenPanel openPanel ] ;
    // preparo il pannello di salvataggio file
    [ sPanel setTitle:@"Directory destinazione" ];
    [ sPanel setPrompt:@"Scegli la directory di destinazione" ];
    [ sPanel setCanChooseDirectories: YES ];
    [ sPanel setCanChooseFiles: NO ];
    risposta = [ sPanel runModal ] ;
    if ( risposta == NSOKButton )
    {
        // se e' andato bene, prelevo il nome del file
        NSString *aFile = [ sPanel filename ] ;
        // e lo assegno alla variabile interna
        [ self setSaveFilePath: aFile ];
        [ theDst setStringValue: saveFilePath ] ;
    }
}
```

Fino a qui, però, non ci sono concetti nuovi. Ma ora, conviene rispolverare i Thread ed alcuni concetti di base di Unix.

WRAPPING E PIPE

L'idea di avere una interfaccia grafica attorno ad un comando Unix non è nuova, anzi, è talmente diffusa da avere un nome proprio: *wrapping* (*to wrap* significa in inglese, incartare, avvolgere). L'idea stessa di eseguire un wrapping di cURL non è per nulla nuova. Ci sono almeno due programmi, decisamente più belli del mio: cURL GUI e iCurl. Ovviamente il mio scopo non è di fare concorrenza a questi programmi, ma di imparare ad utilizzare i comandi Unix all'interno di Cocoa. Quando ho parlato di Thread, avevo accennato anche all'esistenza dei *Task*, programmi del tutto autonomi ma in grado di comunicare con altri task secondo metodi ben definiti. Bene: l'interfaccia grafica è un task, e cURL un altro task. Tutto molto semplice e chiaro.

I problemi cominciano quando il task cURL deve essere lanciato dall'interno del task di interfaccia grafica, e quando i due task devono scambiarsi informazioni.

Come esiste la classe `NSThread` per la gestione dei thread, esiste la classe `NSTask` per la gestione dei task. Questa classe fornisce tutta una serie di funzionalità adatte al lancio ed alla gestione di task.

All'interno di Unix, lanciare un task equivale a scrivere una linea di comando: occorre conoscere il nome del programma, la directory in cui lanciare il programma, gli argomenti di lancio. Automaticamente, ad un comando sono associati tre file: lo standard input (dove il programma legge eventualmente ulteriori caratteri per operare), lo standard output (dove scrive i risultati delle operazioni svolte) e lo standard error (dove scrive le informazioni di errore). Questo è vero sempre, con alcuni piccoli accorgimenti. Ad esempio, lo standard input può essere la tastiera, in modo che il programma possa operare interattivamente (ma lo standard input non è un file! Invece sì, in Unix, qualsiasi cosa è un file, oppure è truccato per apparire tale...); per la applicazioni che non sono lanciate da linea di comando, si utilizzano file di default (`console.log`, ad esempio, visibile con l'omonima applicazione *Console*). Oppure, e

finalmente, questi tre file possono essere dei *pipe*, ovvero, tipi speciali di file che servono per scambiare informazioni tra task (in inglese, pipe è un tubo). Con un pipe, lo standard output di un programma può essere utilizzato come standard input di un altro programma (come appunto un tubo mette in comunicazione due serbatoi, dove il primo pompa liquido verso il secondo): questo è il metodo tipico di Unix per far parlare tra loro due task. Per avere comunicazione bidirezionale, sono necessari due pipe (uno di andata e l'altro di ritorno), ma spesso uno solo è sufficiente. Ancora una volta, Cocoa definisce una classe, `NSPipe`, per rendere disponibile questa caratteristica ad alto livello.

CurlWrapper

Riassumendo: mi occorrono due concetti. Il primo concetto è la classe `NSTask`, che incapsula i meccanismi per lancio e gestione di un task; il secondo concetto è la classe `NSPipe`, che nasconde al suo interno i meccanismi di pipe per la comunicazione tra due task.

Comincio allora a scrivere una classe, `CurlWrapper`, che gestisca il lancio e la gestione del comando cURL.

Il metodo di inizializzazione è il seguente:

```
- (id)
initWithCaller: (id <CurlWrapperController>) aCaller inDir: (NSString* )
workDir
{
    self = [ super init ];
    // mi metto da parte un riferimento al chiamante
    theCaller = aCaller ;
#if    0
    // ricavo il path al programma curl all'interno del bundle
     curlPath = [ [ NSBundle mainBundle ] pathForResource: @"curl" ofType:
@"" ] ;
#else
    // uso il path di default del comando nel sistema
    curlPath = [ NSString stringWithString: @"/usr/bin/curl" ];
#endif
    [ curlPath retain ];
    // metto da parte la directory di lavoro
    workingDir = workDir ;
    [ workingDir retain ];
    return self;
}
```

Discuto dopo lo strano costrutto `<curlwrappercontroller>`; per ora basti sapere che `aCaller` è un riferimento alla classe che intende utilizzare il comando cURL (nella fattispecie, `curlCtrl`). All'interno del metodo metto da parte due stringhe importanti; la prima è il percorso completo del comando. Nel caso, il comando cURL si trova all'interno della directory */usr/bin*, come si può verificare con il Terminale ed il comando *whereis curl*. Ora, il comando cURL è già presente nell'installazione standard di Mac OS X, quindi sono ragionevolmente sicuro di trovarlo. Se invece il programma Unix non è comunemente disponibile, ci sono due possibilità. La prima consiste nell'installazione separata dell'applicazione interfaccia grafica e del comando Unix (e questo, in una posizione opportuna). Tuttavia, conviene sfruttare il fatto che una

applicazione Cocoa è in realtà una cartella speciale, al cui interno sono presenti tutte le risorse necessarie all'esecuzione.

Si potrebbe quindi aggiungere a XCode l'eseguibile all'interno del progetto, come una risorsa tra le altre. In questo modo, durante le operazioni di linking, il programma curl è copiato all'interno dell'applicazione finale `cocoacurl.app`. Così facendo, il suo path è facilmente noto: basta utilizzare la classe `NSBundle` ed i suoi metodi per recuperare il path completo, indipendentemente dalle particolarità dell'installazione dell'applicazione da parte dell'utente.

Nel caso di cURL, tuttavia, si pone un problema ulteriore, dovuto al fatto che di default il comando è costruito in modo da utilizzare librerie dinamiche piuttosto che statiche... insomma è un po' un pasticcio fare il tutto, per cui mi sono ridotto ad utilizzare l'installazione standard di cURL ed andare avanti.

Stabilito il programma da eseguire, occorre indicare la directory corrente; è lo scopo dell'argomento `workDir` del metodo, che è copiato in una variabile d'istanza per averlo disponibile successivamente.

A questo punto, mi trovo nella situazione di conoscere quale programma lanciare, e dove lanciarlo. L'effettivo lancio del programma avviene però in corrispondenza di un ulteriore metodo (perché? per nessuna ragione specifica; avrei in effetti potuto raccogliere tutto nel metodo di inizializzazione...):

```
- (void)
startDownloadWithArgs: (NSArray *) args
{
    NSPipe            * pipe= [ NSPipe pipe ] ;
    NSFileHandle    * handle;

    // dico al chiamante che stiamo partendo
    [ theCaller downloadStarted ];
    // costruisco il task
    curltask = [[NSTask alloc] init];
    // predispongo i parametri di lancio del task
    [ curltask setLaunchPath: curlPath ];
    // la directory di lavoro e' quella di destinazione
    [ curltask setCurrentDirectoryPath: workingDir ];
    [ curltask setArguments: args ];
    // dico che l'uscita del comando va su una pipe (tipo speciale di file)
    // ridirigo sia lo standard output che lo standard error
    [ curltask setStandardOutput: pipe ] ;
    [ curltask setStandardError: pipe ];
    // piglio un riferimento al file - pipe
    handle = [ pipe fileHandleForReading ] ;
    // e dico al sistema operativo di dirmi quando ci sono dati all'interno
del file
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(getData:)
        name: NSFileHandleReadCompletionNotification
        object: handle ];
    // e di farlo in ackground, con calma
    [ handle readInBackgroundAndNotify];
    // lancio il download
    [ curltask launch];
}
```

Qui risiede il cuore di tutto il meccanismo. Le operazioni svolte consistono sostanzialmente nella creazione e configurazione di una istanza di `NSTask`, o, in parole più brutali, nella costruzione di una linea di comando da mandare in esecuzione. Il metodo `setLaunchPath` indica, tramite la stringa che avevo già inizializzato, quale programma utilizzare; il metodo `setCurrentDirectoryPath:` specifica in quale directory il comando va lanciato, il metodo `setArguments:` specifica quali sono gli altri argomenti sulla linea di comando. A questo punto, invocando il metodo `launch`, potrei far partire il comando (e quindi il trasferimento dei file specificato dai parametri). Ho però il problema di scambiare informazioni con questo task, o meglio, mi piacerebbe che il comando cURL mi restituisse qualche informazione su ciò che sta facendo. Insomma, ho bisogno di far parlare tra loro due task: ecco il cuore del problema, dove entra il gioco il concetto di pipe.

Poiché nella configurazione del task ho già specificato ogni parametro, non ho bisogno di immettere ulteriori informazioni tramite lo standard input; mi interessa solamente trasferire informazioni dal task cURL al task di interfaccia grafica. Costruisco quindi un unico oggetto della classe `NSPipe` e gli faccio fare da ponte tra i due task. Per farlo, dico che sia lo standard output che lo standard error di cURL siano ridiretti sul pipe, piuttosto che utilizzare i valori di default. Su questo pipe (che, ribadisco, come quasi ogni cosa in Unix, è un file), definisco e configuro un meccanismo di notifica. Utilizzo la classe `NSFileHandle`, che è un meccanismo di comodo fornito per gestire file aperti o canali di comunicazione (appunto). Se il file è un file normale, ci sono metodi di più alto livello per gestirlo; ma se il file è, come in questo caso, un pipe, è meglio scendere a più basso livello. La cosa interessante che si può fare con questo `NSFileHandle` è di ricevere una notifica quando sono presenti nuovi dati all'interno del file stesso.

In pratica, faccio in modo che il File Manager di Mac OS X mi costringa ad eseguire il metodo `getData:` ogni volta che il pipe presenta dei dati.

```
- (void)
getData: (NSNotification *)aNotification
{
    // recupero i dati presenti
                    NSData     *data     =     [[aNotification     userInfo]
objectForKey:NSFileHandleNotificationDataItem];
    // se ci sono effettivamente i dati...
    if ([data length])
    {
        // li estraggo e li passo intonsi al chiamante
           NSString      * newdata = [[[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding] autorelease] ;
        [ theCaller appendOutput: newdata ];
    }
    else
    {
        // quando non ci sono piu' dati, il processo e' finito
        [self stopDownload];
    }
    // rifaccio ripartire la notifica
    [[aNotification object] readInBackgroundAndNotify];
}
```

Inoltre, il File Manager è così gentile da fornirmi, all'interno della notifica, un dizionario che contiene proprio i dati che sono presenti all'interno del file. Prendo allora

questi dati (che poi sono una serie di caratteri) e li rispedisco indietro alla classe chiamante `curlCtrl`. Da notare che il meccanismo di notifica è a colpo singolo, ovvero, la notifica viene data una volta sola. Per poter proseguire le operazioni di lettura dati dal pipe, occorre far ripartire il meccanismo con il metodo `readInBackgroundAndNotify`.

L'ultimo metodo della classe `CurlWrapper` deve essere eseguito al termine del trasferimento, e serve a fare le pulizie di casa:

```
- (void)
stopDownload
{
    NSData *data;
    NSFileHandle    * handle;
    // ripiglio un handle alla pipe
    handle = [ [curltask standardOutput] fileHandleForReading ] ;
    // cancello le notifiche
    [[NSNotificationCenter defaultCenter] removeObserver:self
            name:NSFileHandleReadCompletionNotification
            object: handle ];
    // termino il task
    [ curltask terminate ];
    // estraggo gli eventuali dati presenti nel file
    while ((data = [handle availableData]) && [data length])
    {
            [ theCaller appendOutput: [[[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding] autorelease]];
    }
    // dico che ho finito le operazioni
[ theCaller downloadFinished ];
theCaller = nil;
}
```

Qui è infatti rimossa l'osservazione delle notifiche sul `NSFileHandle`, e si estraggono gli eventuali ultimi dati pendenti all'interno del file (nel caso in cui il trasferimento del file sia stato interrotto prima dell'effettivo completamento; occorre lasciare il pipe pulito).

QUESTIONI DI PROTOCOLLO

A questo punto ritorno alla classe `curlCtrl` ed al fatto che questa classe deve rispettare un protocollo. Infatti (all'interno del file `CurlWrapper.h`) ho definito un protocollo cui devono rispondere tutte le classi che vogliono interagire con la classe `CurlWrapper`:

```
@protocol    CurlWrapperController
// quando c'e' qualcosa da riportare indietro al task principale
- (void)appendOutput:(NSString *)output;
// callback alla partenza di un download
- (void)downloadStarted ;
// callback al termine di un download
- (void)downloadFinished ;
@end
```

Con un protocollo indico esplicitamente quali metodi una classe deve realizzare per

poter effettuare con successo degli scambi di dati o di funzioni con un'altra classe. In questo caso, per poter lavorare con il comando cURL sono indicati tre metodi. Il metodo `downloadStarted` è utilizzato per notificare l'avvenuta partenza di un trasferimento, il metodo `downloadFinished` per indicare la conclusione (eventualmente anticipata) del trasferimento, mentre il metodo `appendOutput` trasferisce indietro una stringa di informazioni.

In altre parole, se una classe A intende utilizzare la classe `CurlWrapper`, conosce i metodi propri di `CurlWrapper` descritti nella sua interfaccia pubblica:

```
@interface CurlWrapper : NSObject
...
// inizializzazione della classe con gli argomenti
-   (id)initWithCaller:   (id   <CurlWrapperController>)   aCaller   inDir:
(NSString*) workdir ;
// partenza delel operazioni
- (void) startDownloadWithArgs: (NSArray *) args ;
// fine delle operazioni
- (void) stopDownload ;
@end
```

Tuttavia, la classe A deve anche adeguarsi al protocollo `CurlWrapperController`, realizzando i tre metodi `appendOutput, ...` sopra descritti.

In effetti, per questo semplice esempio, tutta l'architettura impostata è piuttosto sovrabbondante: si poteva conglobare in una unica classe tutti i metodi di `curlCtrl` e `CurlWrapper`, semplificando di parecchio la programmazione. Ma il buon programmatore orientato agli oggetti deve lavorare per il futuro; scrivendo la classe `CurlWrapper`, ho separato la parte di gestione del comando cURL dal resto dell'applicazione, con la quale interagisco in modi ben definiti (l'interfaccia pubblica ed il protocollo). Oggi il comando cURL è utilizzato dalla semplice interfaccia grafica sopra vista, ma nulla impedisce inserire la classe `CurlWrapper` all'interno di applicazioni più complicate.

RITORNO A CURLCTRL

C'è da descrivere come ho realizzato il protocollo `CurlWrapperController` all'interno della classe `curlCtrl`.

Utilizzo il metodo `downloadStarted` per predisporre l'interfaccia in modo che indici attività di trasferimento:

```
- (void)
downloadStarted
{
    NSMutableAttributedString    * appStr ;
    NSDictionary    * dd ;
    NSTextStorage    * txtSto = [messageField textStorage] ;
    curStyle = [ NSMutableDictionary dictionaryWithCapacity: 2 ] ;
    [ curStyle retain ];
     dd = [ txtSto fontAttributesInRange: NSMakeRange(0, [txtSto length]) ]
;
    [ curStyle addEntriesFromDictionary: dd ] ;
```

```
    dd = [ txtSto rulerAttributesInRange: NSMakeRange(0, [txtSto length]) ]
;
    [ curStyle addEntriesFromDictionary: dd ];
    downloadInProgress = YES;
    [theGetButton setTitle:@"Stop"];
          appStr   =   [[NSMutableAttributedString   alloc]   initWithString:
@"Downloading...\n"] ;
        [  appStr  setAttributes:  curStyle  range:  NSMakeRange(0,  [appStr
length])];
    [ txtSto setAttributedString: appStr ];
}
```

La parte più corposa consiste nel leggere gli attributi del campo di testo e salvarli in una variabile d'istanza; lo stile del testo è poi utilizzato per scrivere una stringa che segnala l'inizio lavori. Le altre due istruzioni presenti impostano l'una una variabile interna per indicare un trasferimento in corso, e l'altra modifica l'aspetto del pulsante in modo che presenti l'etichetta *Stop*.

L'etichetta è riportata al suo valore originario dal metodo `downloadFinished`

```
- (void)
downloadFinished
{
    downloadInProgress=NO;
    [ curStyle release ];
    [theGetButton setTitle:@"Get File"];
}
```

Il metodo reimposta anche le altre variabili d'istanza.

Ho lasciato per ultimo il metodo più importante, quello che è attivato dal clic dell'utente sopra il pulsante *Get File* (o *Stop* durante il trasferimento): risulta adesso di molta più semplice comprensione.

```
- (IBAction)
getFile:(id)sender
{
    // se il download e' in corso, lo fermo
    if ( downloadInProgress )
    {
        // fermo il task di download
        [ curlWrapper stopDownload];
        // lo distruggo del tutto
        [ curlWrapper release ];
        // e metto a nil per sicurezza
        curlWrapper=nil;
        // fine delle operazioni
        return;
    }
    else
    {
        NSString        * theNewUrl = [ theUrl stringValue ];
        // qui bisogna lanciare le operazioni di download
        NSString        * dstFile = [ theNewUrl stringByStandardizingPath ]
;
        // costruisco l'elenco degli argomenti
        NSMutableArray    * argList = [ NSMutableArray arrayWithCapacity: 1
];
        [ self setUrl2download: theNewUrl ];
```

```
        dstFile = [ dstFile lastPathComponent ] ;
        // il mio primo argomento e' il nome del file
         [ argList addObject: [ NSString stringWithFormat: @"-o %@", dstFile
] ];
        // ...
        // a quanto pare, le opzioni non abbreviate con argomento van messe
su due
        // argomenti successivi...
#if    0
        // la versione di default in Mac Os X 10.3 non gestisce
        // correttamente questa opzione...
        [ argList addObject: @"--limit-rate" ];
        [ argList addObject: @"50K" ];
#endif
        [ argList addObject: @"--progress-bar" ];
        // l'ultimo argomento e' l'URL
        [ argList addObject: url2download ];
        // mi premunisco in caso di problemi
        if (curlWrapper !=nil)
            [curlWrapper release];
        // costruisco il task di download
            curlWrapper= [ [CurlWrapper alloc] initWithCaller:self inDir:
saveFilePath ];
        // lancio il task di download
        [ curlWrapper startDownloadWithArgs: argList ];
    }
}
```

Comincio dalla parte `else` dell'istruzione `if`, che corrisponde all'operazione di lancio del trasferimento.

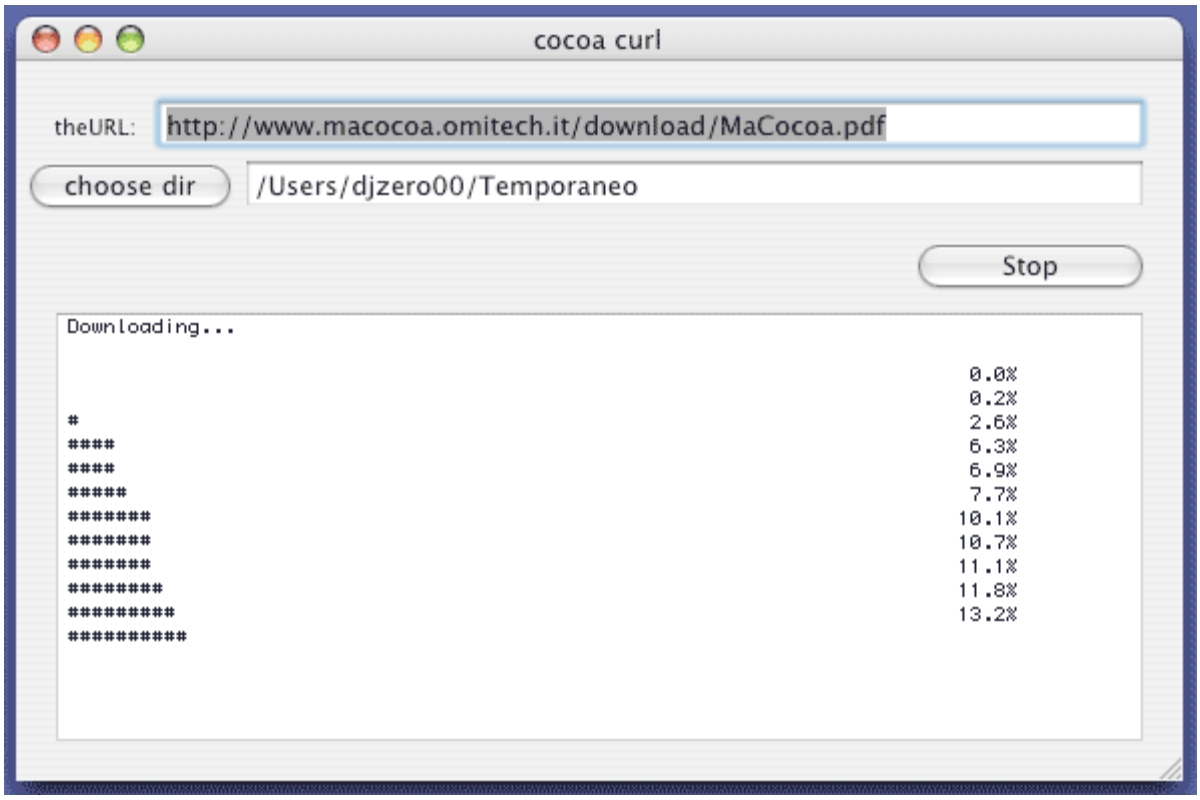
Prelevo lo URL completo del file da trasferire, e ne estraggo l'ultima parte, che uso anche come nome del file locale di destinazione. Preparo poi nel vettore `argList` l'insieme di tutti gli argomenti necessari al lancio del comando; col primo argomento, indico il nome del file destinazione, come secondo argomento faccio in modo che il comando scriva sullo standard output una sorta di barra di avanzamento piuttosto che brutali informazioni numeriche, infine aggiungo il percorso completo del file da trasferire. Costruisco poi un elemento della classe `CurlWrapper`, che è subito configurato passandogli la directory destinazione nel metodo di inizializzazione. Finalmente, si comanda l'esecuzione del trasferimento.

A questo punto, l'oggetto `CurlWrapper` appena creato, esegue il metodo `startDownloadWithArgs` sopra visto, costruendo il task e lanciando ill comando cURL passando gli argomenti indicati. Così facendo, si mette in moto il meccanismo di trasferimento; man mano che il procedimento va avanti, cURL scrive qualcosa nel pipe (la barra di avanzamento che ho specificato). Allora si scatena la notifica, che provoca l'esecuzione del metodo `getData:`, il quale a sua volta provoca l'esecuzione del metodo `appendOutput:`

```
- (void)
appendOutput:(NSString *)output
{
    float    perc ;
    NSMutableAttributedString    * appStr ;
    NSDictionary    * dd ;
    NSTextStorage    * txtSto = [messageField textStorage] ;
    appStr = [[NSMutableAttributedString alloc] initWithString: output] ;
```

```
        [ appStr setAttributes: curStyle range: NSMakeRange(0, [appStr
length])];
    [ txtSto appendAttributedString: appStr ];
      [messageField scrollRangeToVisible:NSMakeRange([[messageField string]
length], 0)];
}
```

Questo metodo piglia la stringa passatagli come argomento e l'aggiunge, con l'opportuna formattazione, al campo di testo.



Il procedimento di trasferimento può andare avanti fino al suo naturale compimento; in tal caso l'oggetto `CurlWrapper` provoca direttamente l'esecuzione del metodo `downloadFinished`, che termina il tutto. Alternativamente, può essere l'utente stesso che, stufo di aspettare o per qualche altro motivo, interrompe il trasferimento facendo clic sopra il pulsante che adesso si chiama *Stop*.

Viene così forzata l'esecuzione del metodo `stopDownload` della classe `CurlWrapper`, il quale, a sua volta, richiede l'esecuzione di `downloadFinished`, e tutto termina pulitamente anche questa volta (ma il file, pur presente nella directory di destinazione, è presente solo in modo parziale...).

L'applicazione che ho sviluppato è veramente grezza; è addirittura meno semplice da usare del comando cURL da linea di comando, ed è estremamente limitata nelle funzionalità proposte. Si comporta male in caso di errore, è brutta a vedersi, insomma, una vera schifezza. Però mi ha permesso di fare una cosa veramente importante: due task, uno scritto in Cocoa e l'altro preesistente, si sono scambiati informazioni. Tutto sommato, un buon punto di partenza.

Capitolo 066 - Che fortuna!

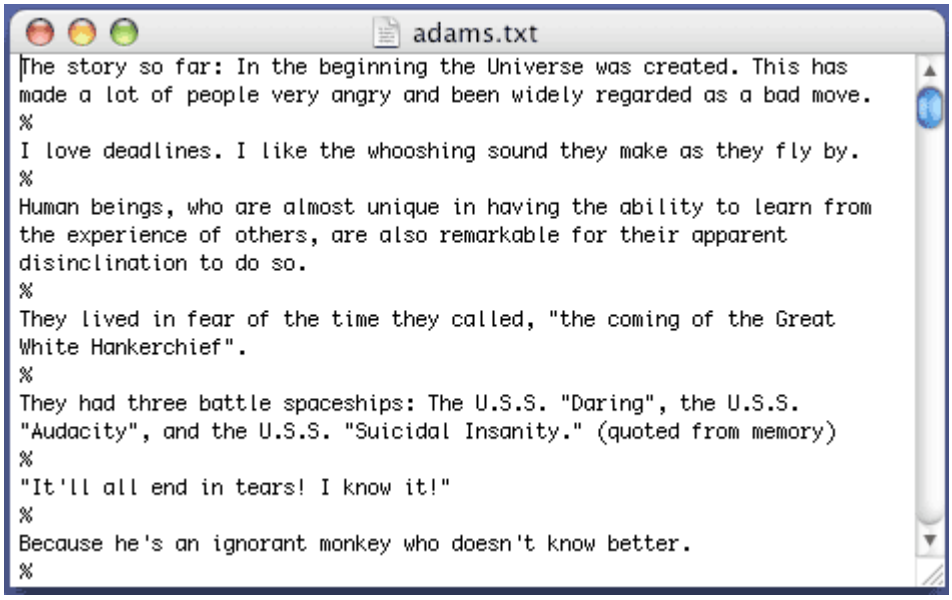
Piglio spunto da un tutorial su Tevac riguardante il programma Unix chiamato *Fortune* per produrre una interfaccia grafica per il comando stesso, che presenti in una bella finestra la citazione del giorno.

Sorgenti: quelli di fortune, esempi e documentazione Apple

Prima stesura: 15 dicembre 2004.

FORTUNE

Fortune è un programma Unix di vecchia data che funziona da linea di comando. Ad ogni lancio produce una citazione pescata a caso da un insieme specificato. L'uso classico di questo comando è di lanciarlo al login di un utente, in modo che costui possa essere gratificato da una citazione o epigramma (generalmente divertente o pensosa, comunque interessante). Le citazioni sono conservate in una serie di file, in formato testo, con una struttura molto semplice. Se ne trovano a bizzeffe in internet, e se ne possono produrre anche da soli con le proprie citazioni preferite. Io stesso, per completezza, ne ho prodotto uno. Ho recuperato una serie di citazioni dell'unica trilogia composta da cinque libri (si tratta della *Guida Galattica per Autostoppisti*, del compianto *Douglas Adams*) e le ho inserite in un file di testo; ciascuna citazione è separata dalla successiva dal carattere %, presente da solo nella riga.

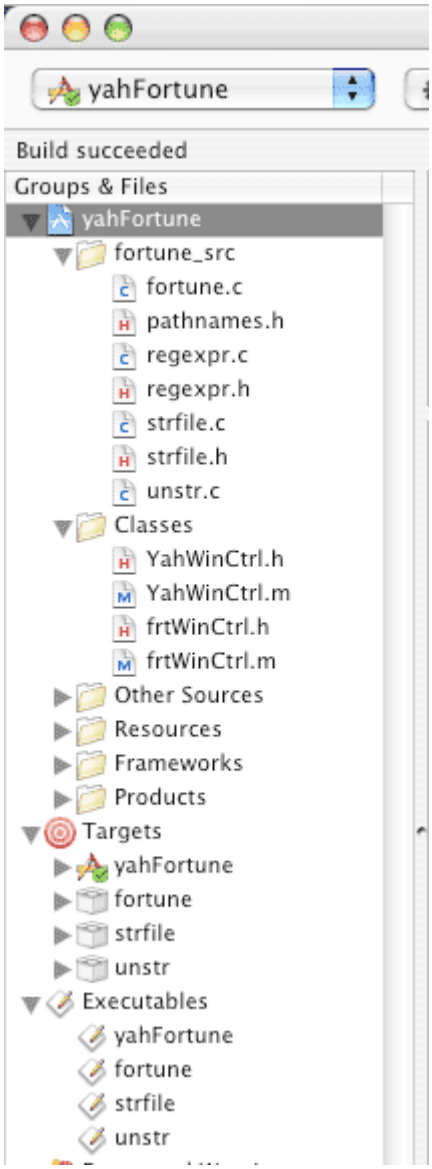


Ho poi recuperato i file sorgenti del comando fortune, ad esempio dal sito di Tevac, oppure da una distribuzione linux come Debian. Il primo compito che mi prefiggo è di compilare i file all'interno di XCode.

Faccio riferimento ai file sorgenti ottenuti tramite Tevac. L'intero pacchetto si compone di quattro file C con relativi header, più una serie di file ausiliari. La tecnica standard per la produzione del comando è di eseguire il comando `make` all'interno della directory dei sorgenti, ma io intendo operare in un altro modo.

PROGETTI E TARGET

Costruisco un progetto in XCode, una *Cocoa-Based Application*, chiamato *YahFortune*. Questo progetto conterrà sia l'applicazione grafica che il comando `fortune`.

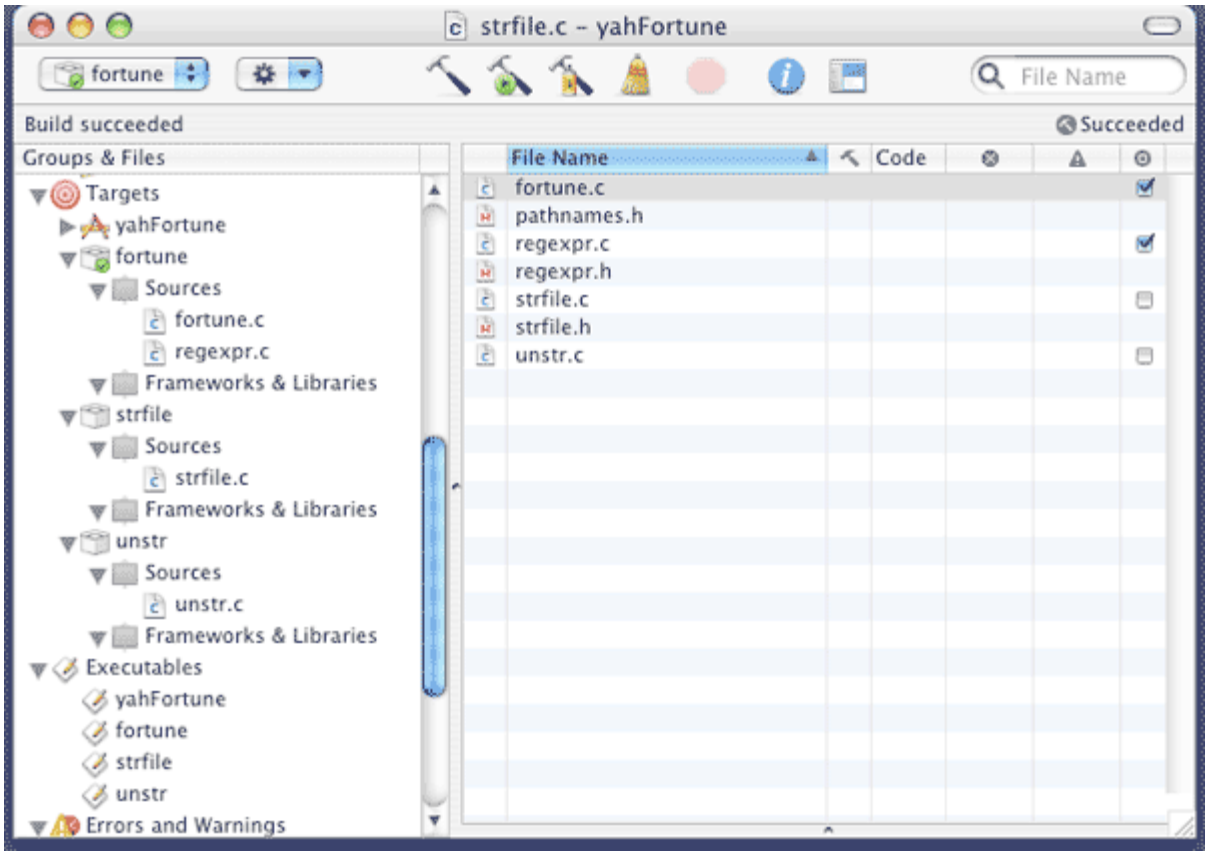


Converrà spendere due parole sul concetto di *Target*, finora mai esplicitamente dichiarato in quando la situazione di default ha sempre funzionato. XCode raccoglie in un progetto una serie di file (sorgenti di vari linguaggi, risorse, testi, figure, suoni, eccetera), eventualmente separandoli in diversi gruppi e sottogruppi. Un Target è un insieme di regole che specificano un gruppo di file, come manipolarli e come raggrupparli assieme per produrre un obiettivo. L'esempio più semplice è una applicazione Cocoa. Il target identifica i file Objective-C da compilare, i file nib da aggiungere e come organizzare la cartella .app del prodotto finito.

C'è poi da ricordare la presenza degli *Executables*, ovvero l'ambiente in cui il risultato del Target è fatto funzionare. Quando ho prodotto un salvaschermo, non avevo un Executable esplicito, in quanto un salvaschermo non vive autonomamente, ma solo all'interno di un'altra applicazione. A quel tempo, avevo specificato le *Preferenze di*

Sistema come eseguibile per poter verificare il funzionamento del salvaschermo.

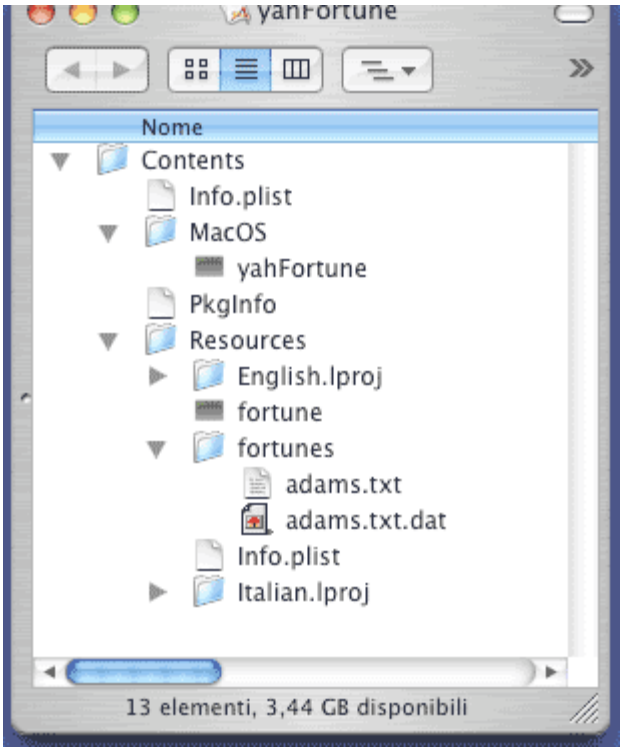
Il punto interessante è che all'interno di un unico progetto possono coesistere diversi Target (per capirci, una applicazione che consiste sostanzialmente negli stessi file, ma con due diverse modalità di compilazione, ad esempio per Mac Os 9 e Mac Os X). Sfrutto questa situazione per produrre quattro diversi target. Il primo target sarà l'interfaccia grafica di fortune, la mia applicazione principale. Un secondo target è il comando fortune, compilato a partire dai suoi sorgenti; il terzo ed il quarto target sono due altri comandi che sono associati a fortune, utili per la gestione dei file con le citazioni.



Per aggiungere un target, si seleziona l'apposita voce nel menu `Project`. Si devono poi selezionare i file che partecipano ad ogni target: ci sono diversi metodi per fare ciò.

Il primo metodo è di selezionare ciascun target (ad esempio con il menu pop-up nella toolbar) e spuntare nella finestra che presenta l'elenco dei file l'appartenenza o meno al target (è l'ultima colonna della tabella sulla destra in figura). Oppure si possono draggare i file nell'apposita sezione source file della vista laterale (l'alberatura sulla sinistra). Dei quattro file .c che costituisco la distribuzione di fortune, due sono dedicati al comando fortune vero e proprio, ed uno ciascuno ai due comandi ausiliari `strfile` e `unstr`.

Definiti i target, passo a compilare. A parte una lunga sequenza di warning, produrre i comandi strfile e unstr è banale. Un po' più complicato il comando fortune vero e proprio, in quanto l'eseguibile deve incorporare la directory di default dove trovare i file con le citazioni.



Normalmente, su Unix, la directory è qualcosa del tipo `/usr/share/games/fortune`; tuttavia, io vorrò incorporare l'eseguibile fortune e alcuni file di citazioni di default all'interno dell'applicazione *yahFortune.app* risultante, in modo che l'applicazione non necessiti, in prima battuta, di altri file al di fuori dell'applicazione stessa. L'idea è che, visualizzando il contenuto del pacchetto *yahFortune.app*, il comando fortune si trovi all'interno della cartella *Resources*, e che i file delle citazioni siano conservati all'interno della sottocartella fortunes (nome predefinito dal comando).

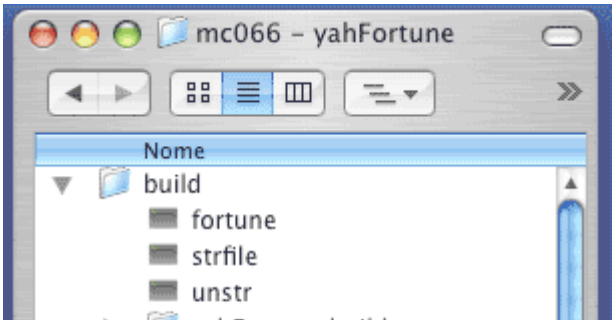
Tutta questa introduzione giustifica il contenuto del file `pathnames.h`

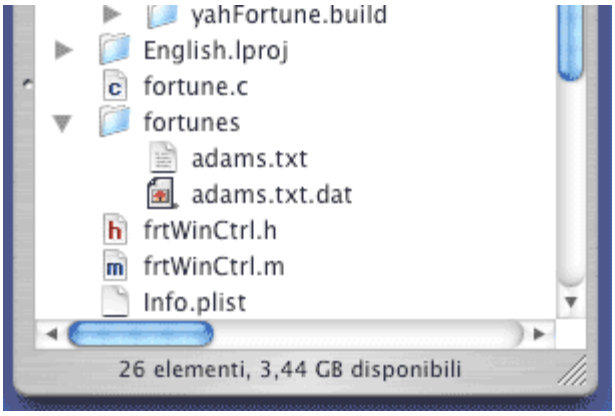
```
#define FORTDIR "."
```

che indica a fortune che la cartella fortune si trova nella stessa in cui si trova il comando stesso.

USO DI FORTUNE

A questo punto mi trovo con i tre comandi della distribuzione di fortune compilati e pronti all'uso.





Tuttavia, operando con il terminale, le cose non funzionano.

```
djzeropb:~/path/mc066 - yahFortune/build djzero00$ ./fortune
./fortunes: No such file or directory
djzeropb:~/path/mc066 - yahFortune/build djzero00$
```

In effetti, la configurazione non è quella definitiva: il comando fortune andrà a finire in altra posizione, assieme ai file delle citazioni. Per verificarne il funzionamento, bisogna ricreare delle condizioni opportune, ad esempio copiando il comando e la cartella fortune assieme in un'altra directory:

```
djzeropb:~/temp_fortune djzero00$ ls -l
total 24
-rwxr-xr-x 1 djzero00 djzero00 9300 15 Dec 16:23 fortune
drwxr-xr-x 5 djzero00 djzero00 170 15 Dec 16:34 fortunes
djzeropb:~/Temporaneo/temp_fortune djzero00$ ./fortune
fortune: ./fortunes: No fortune files in directory.
fortune:./fortunes not a fortune file or directory
djzeropb:~/Temporaneo/temp_fortune djzero00$
```

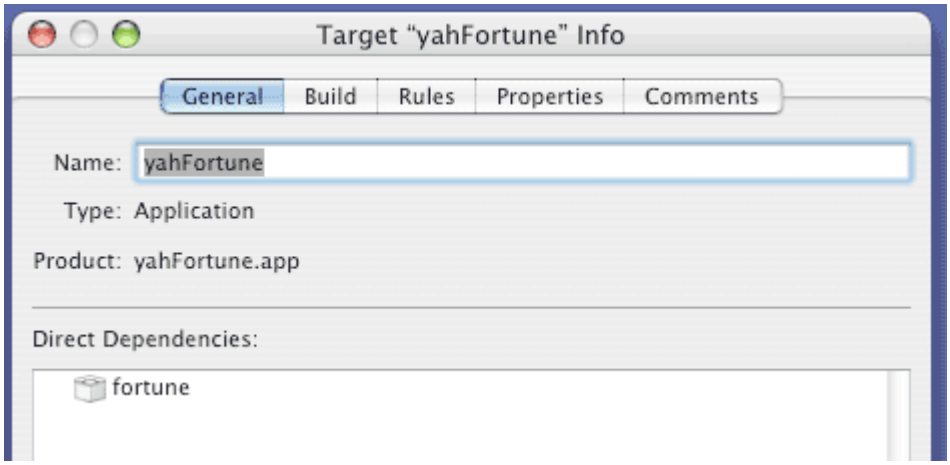
In realtà, nemmeno così va bene. Entrano in gioco il comando `strfile` e i file `dat`. Per poter funzionare correttamente, infatti, fortune richiede, oltre al file testo con le citazioni, una specie di indice di questo file; produrre questo indice è compito del comando `strfile`.

```
djzeropb:~/temp_fortune djzero00$ cd fortunes
djzeropb:~/temp_fortune/fortunes djzero00$ ../strfile adams.txt
"adams.txt.dat" created
There were 115 strings
Longest string: 569 bytes
Shortest string: 4 bytes
djzeropb:~/temp_fortune/fortunes djzero00$ cd ..
djzeropb:~/temp_fortune djzero00$ ./fortune
There is a theory which states that if anyone discovers just exactly
what the universe is for and why we are here, that it will instantly
disappear and be replaced by something even more bizarre and
inexplicable. Then there is a theory which states that this has already
happened.
djzeropb:~/temp_fortune djzero00$
```

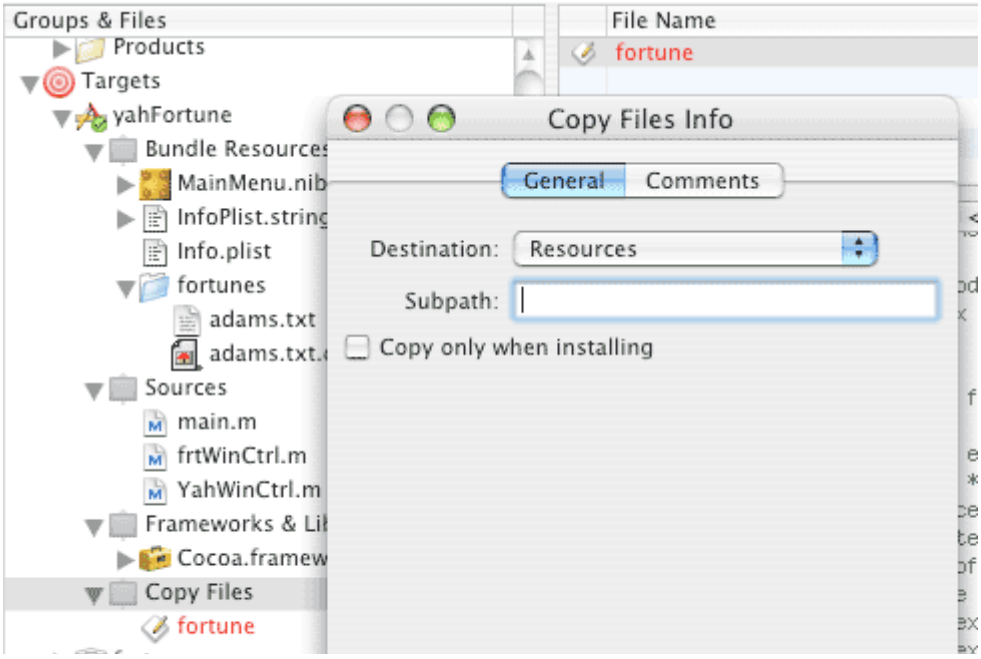
Ecco che adesso le cose funzionano. tengo da conto il file `dat` e passo finalmente a progettare l'interfaccia grafica.

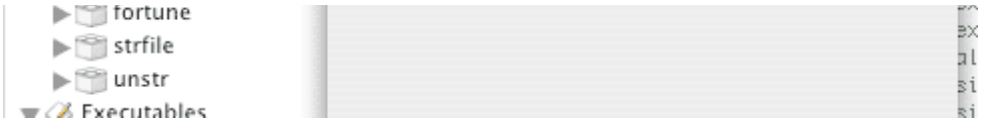
YAHFORTUNE

Ancora una volta, non sono particolarmente originale nell'idea di costruire una interfaccia grafica per il comando fortune. Ne trovate ad esempio una chiamata CocoaFortune, completa di molti file. Come al solito, il mio scopo non è produrre applicazioni senza le quali il mondo non può vivere, ma sperimentare con Cocoa (il nome stesso del progetto, *YahFortune*, per chi se lo fosse chiesto, sta per *Yet Another Horrible Fortune*, un altro orribile fortune). Tuttavia, cercherò di fare qualcosa di carico e multifunzionale, sviluppando nei prossimi capitoli alcune idee.

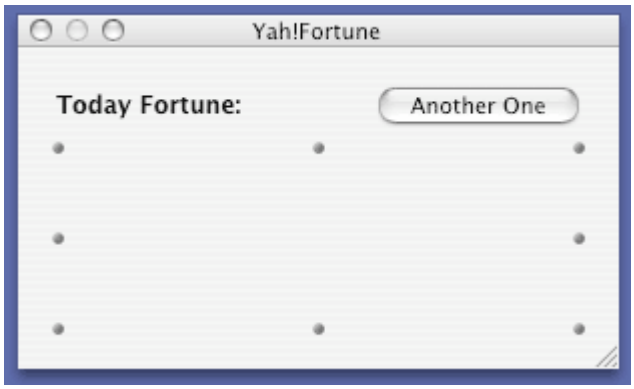


Il primo problema nasce quando voglio incorporare il comando fortune all'interno della applicazione. Due sono le operazioni necessarie: dire che il comando fortune è propedeutico alla compilazione dell'applicazione, e poi copiare il comando fortune all'interno dell'applicazione. La prima operazione si compie selezionando le informazioni del target, ed esplicitando la dipendenza del target YahFortune dal target fortune. In questo modo XCode, compilando l'applicazione, verifica in primo luogo l'esistenza del comando fortune. Se questo comando non fosse presente, esegue la compilazione del codice del comando prima di ogni altra operazione relativa all'applicazione.





Per inserire il comando fortune all'interno dell'applicazione, occorre invece definire una fase di build apposta. C'è una voce di menu nel menu Project che permette di aggiungere una fase di copiatura file. Questa voce aggiunge una cartella nelle proprietà del Target, le cui proprietà possono essere opportunamente configurate.



L'interfaccia è essenziale: un pannello con una scritta, un campo di testo per contenere la citazione, ed un pulsante per generare un'altra citazione.

E qui comincio subito a complicarmi al vita. Voglio che la finestra non abbia la barra del titolo, e che soprattutto possa essere chiusa (e l'intera applicazione con lei) con un clic del mouse o con la pressione di un tasto.

Per farlo, ho cercato a lungo nella documentazione la possibilità di nascondere la barra del titolo, ma non ho trovato nulla. O meglio, l'unica possibilità sembrava essere quella di costruire da programma la finestra, senza sfruttare le immense possibilità di *Interface Builder*. Dopo un po' di tribolazioni, la soluzione è molto semplice: si costruisce una sottoclasse di `NSWindow`, nel mio caso chiamata `YahWinCtrl`, e si riscrive il metodo di inizializzazione:

```
- (id)initWithContentRect:(NSRect)contentRect
    styleMask:(unsigned int)styleMask
    backing:(NSBackingStoreType)backingType
    defer:(BOOL)flag
{
    // riscrivo il metodo per togliere titoli bordi e quant'altro
```

```
    styleMask = NSBorderlessWindowMask ;
    // chiamo ovviamente il metodo della superclasse
    self = [ super initWithContentRect: contentRect styleMask: styleMask
                backing: backingType defer: flag ];
    return ( self );
}
```

Indicare infatti lo stile `NSBorderlessWindowMask` elimina tutto ciò che della finestra è accessorio (come appunto la barra del titolo).

A questo punto, già che ho costruito una sottoclasse, intercetto la pressione di un tasto ed il clic dei mouse:

```
- (void)
mouseDown:(NSEvent *)theEvent
{
    // passo la palla al delegato (frtWinCtrl)
    [[self delegate] handlesMouseDown: theEvent ] ;
}

- (void)
keyDown:(NSEvent *)theEvent
{
    // passo la palla al delegato (frtWinCtrl)
    [[self delegate] handlesKeyDown: theEvent ] ;
}
```

Il delegato della finestra sarà la classe che controlla l'intera applicazione, e che comincerà la procedura di terminazione dell'applicazione (perché non termino l'applicazione direttamente da qui sarà chiaro più avanti).

Se però la cosa funziona con il mouse, con la pressione di un tasto si ha il beep di sistema e la mancata trasmissione dell'incombenza al delegato. Dopo il solito lungo penare, vedo un esempio di apple (si chiama `FancyAbout`), dove si afferma che le finestre senza barra del titolo non possono diventare finestre principali e tanto meno finestre ricevere input da tastiera. Ma nello stesso tempo mostra anche come risolvere il problema: basta forzare la mano sovrascrivendo due metodi.

```
- (BOOL)
canBecomeMainWindow
{
return YES;
}

- (BOOL)
canBecomeKeyWindow
{
return YES;
}
```

Adesso, sia un clic del mouse sia la pressione di un tasto sono eventi passati al delegato.

FAR FORTUNA

La classe fondamentale dell'applicazione, `frtWinCtrl`, è responsabile della

visualizzazione della finestra e del riempimento del campo con il testo di una citazione.

Mi libero subito del problema di recuperare la citazione, visto che si tratta eseguire in un secondo task un comando Unix, e recuperare il prodotto di questo comando. Ho scritto un metodo che fa tutto, restituendo una stringa.

```
- (NSString *)
makeAFortune
{
    NSPipe            * pipe = [ NSPipe pipe ] ;
    NSFileHandle    * handle = [ pipe fileHandleForReading ] ;
    NSString        * fortunePath ;
    NSTask            * fortuneTask ;
    NSData            * data;
    NSString        * theNewFortune = [ NSString string ];
    long            xxx ;

    // recupero il percorso dell'eseguibile all'interno dell'applicazione
      fortunePath = [ [ NSBundle mainBundle ] pathForResource: @"fortune"
ofType: @"" ] ;
    // costruisco un task apposito
    fortuneTask = [[NSTask alloc] init];
    // predispongo i parametri di lancio del task
    [ fortuneTask setLaunchPath: fortunePath ];
    // la directory di lavoro e' quella di destinazione
             [   fortuneTask   setCurrentDirectoryPath:   [   fortunePath
stringByDeletingLastPathComponent] ];
    // dico che l'uscita del comando va su una pipe
    // ridirigo sia lo standard output che lo standard error
    [ fortuneTask setStandardOutput: pipe ] ;
    [ fortuneTask setStandardError: pipe ];
    // lancio il programma
    [ fortuneTask launch];
    // gli do un attimo per partire
    Delay ( 1, & xxx );
    // aspetto, in maniera sincrona, che il programma risponda con tutta la
stringa
    // prodotta secondo i suoi criteri
    while ((data = [handle availableData]) && [data length])
    {
        NSString    * tmpData ;
        // prelevo i caratteri arrivati
                    tmpData   =   [[NSString   alloc]   initWithData:data
encoding:NSUTF8StringEncoding] ;
        // e li aggiungo a quelli gia' arrivati
        theNewFortune = [ theNewFortune stringByAppendingString: tmpData ];
    }
    // ecco il nuovo fortune
    return ( theNewFortune );
}
```

Rispetto al capitolo precedente, c'è una unica fondamentale differenza, ovvero che la lettura del risultato avviene in maniera sincrona piuttosto che asincrona; in altre parole, si lancia il comando fortune, e si aspetta che produca il risultato.

Ora, potrei finire qui; dopotutto, si tratta di inserire questa stringa all'interno del campo apposito; invece, mi complico ulteriormente la vita.

Voglio fare una piccola coreografia, facendo apparire la finestra con un effetto di *fading*, manipolandone la trasparenza. In altre parole, parto con la finestra

completamente trasparente, e poi la opacizzo nel tempo fino a renderla del tutto solida.

Ugualmente, al termine delle operazione, eseguo l'operazione inversa, partendo da finestra solida e opaca, rendendola sempre più trasparente fino a farla scomparire.

Il metodo di inizializzazione si limita a predisporre una paio di variabili d'istanza:

```
- (id)
init
{
    self = [super init];
    if ( self )
    {
        // predispongo le variabili interne d'uso
        curStyle = defaultTextAttr( ) ;
        [ curStyle retain ] ;
        currentFade = 0 ;
        // predispongo un fortune
        [ self setLastFortune: [ self makeAFortune ] ];
    }
    return self;
}
```

La variabile `currentFade` è il grado di trasparenza iniziale: con Zero la finestra è completamente trasparente, con Uno del tutto opaca. La variabile `lastFortune` contiene la citazione corrente, ed è riempita con il metodo visto in precedenza.

La variabile `curStyle` contiene un `NSDictionary` con gli attributi di visualizzazione della stringa: in pratica, specifica il carattere:

```
NSMutableDictionary *
defaultTextAttr( )
{
    NSFont                      * locFont ;
    NSMutableDictionary          * txtAtt ;
    // inizializzo un dizionario epr contenere gli attributi
          txtAtt   =   [[[NSMutableDictionary   alloc]   initWithCapacity:1]
autorelease];
    // un font di default come primo attributo
    locFont = [ NSFont fontWithName: @"Georgia" size: 14 ];
    // aggiungo il tutto al dizionario
    [ txtAtt setObject: locFont forKey:NSFontAttributeName];
    return ( txtAtt );
}
```

Quando la finestra è caricata, benché non appaia ancora sulla scrivania, si esegue il metodo `awakeFromNib`:

```
- (void)
awakeFromNib
{
    // per fare in modo che la finestra sia comunque
    // davanti a tutte le altre, anche se l'applicazione
    // non ha menu e non e' presente in dock
    [ NSApp activateIgnoringOtherApps:YES];
    // la rendo invisibile
    [ theWindow setAlphaValue: currentFade ];
    // la porto davanti a tutti
```

```
    [ theWindow makeKeyAndOrderFront: self ];
    // imposto il font del campo
    [ theText setFont: [ curStyle objectForKey:NSFontAttributeName ] ];
    // ridimensiono e centro la finestra
    [ self resizeWindowToNewFortune ];
    [ theWindow center ];
    // faccio partire il processo di fading
    [ self makeWindowFadeInOut: YES ];
}
```

La prima istruzione serve a portare la finestra davanti a tutte le altre, anche se l'applicazione risultante sarà *background only*, in modo che non abbia la barra dei menu e non compaia nel dock (ho scoperto come fare ciò nel capitolo 59 con la proprietà `LSUIElement` posta a 1). Imposto la trasparenza iniziale, porto davanti la finestra, assegno il font al campo di testo, lascio perdere le due istruzioni successive, e poi chiamo il metodo `makeWindowFadeInOut:`, per modificare nel tempo il valore di trasparenza:

```
- (void)
makeWindowFadeInOut: (BOOL) uot
{
    // devo fare il fading della finestra
    // in entrambi i casi, faccio partire un timer, che lanci ad intervalli
    // predefiniti, il metodo updateAlpha:
    fadeTimer = [NSTimer scheduledTimerWithTimeInterval: 0.1F target:self
        selector: @selector(updateAlpha:)
        userInfo: [ NSNumber numberWithBool: uot ]
        repeats: YES ];
}
```

Il metodo fa semplicemente partire un timer (una tecnica già vista nel capitolo 24), ovvero faccio in modo che ogni 0.1 secondi sia chiamato il metodo `updateAlpha:` di se stessa, cioè `frtWinCtrl`. Il parametro `userInfo` mi serve per distinguere il quale direzione si muove la trasparenza

```
- (void)
updateAlpha: (NSTimer *)timer
{
    // recupero la direzioen del fading
    BOOL    dir = [ [ timer userInfo] boolValue ];
    // se YES, fading in crescendo
    if ( dir )
    {
        // fading da Zero a Uno, apparizione
        currentFade += 0.1 ;
        // se raggiungo o supero 1, ho finito
        if ( currentFade >= 1 )
        {
            // uccido il timer
            [ fadeTimer invalidate ];
            // imposto 1 per essere sicuro
            currentFade = 1 ;
                  // imposto un nuovo timer, dopo i secondi della morte
programmata
                fadeTimer = [NSTimer scheduledTimerWithTimeInterval: 60.0
target:self
                 selector: @selector(endOfDisplay:) userInfo: nil repeats:
NO ];
        }
    }
```

```
    else
    {
        // fading da Uno a Zero, sparizione
        currentFade -= 0.1 ;
        // se sotto o uguale a zero, ho finito
        if ( currentFade <= 0 )
        {
            // uccido il timer
            [ fadeTimer invalidate ];
            // ma tanto vale: uccido anche l'applicazione
            [ NSApp terminate: self ] ;
        }
    }
    // aggiorno la finestra con il nuovo valore alpha
    [ theWindow setAlphaValue: currentFade ];
}
```

Con `userInfo` pari a YES, passo da finestra trasparente a finestra opaca (e viceversa con NO). In ogni caso, aggiorno il valore di `currentFade`, incrementando o decrementando secondo il caso. Se arrivo al limite superiore di 1, termino l'incremento della trasparenza uccidendo il timer; ma lo reimposto subito (un singolo colpo dopo un minuto); in questo modo l'applicazione, dopo un minuto, anche in assenza di attività da parte dell'utente, si chiude e termina di lavorare (è una scelta che modificherò più avanti). Se arrivo al limite inferiore di Zero, la finestra è scomparsa del tutto, e termino l'applicazione esplicitamente.

Dopo un minuto dalla partenza, il metodo `endOfDisplay`: fa ripartire il processo di modifica della trasparenza in senso contrario.

```
- (void)
endOfDisplay: (NSTimer *)timer
{
    // faccio ripartire il fading in dissolvenza
    [ self makeWindowFadeInOut: NO ];
}
```

Anche i metodi chiamati dalla finestra attraverso il meccanismo di delega fanno partire lo stesso processo:

```
- (BOOL)
handlesKeyDown: (NSEvent *) keyDown
{
    [ self makeWindowFadeInOut: NO ];
}

- (BOOL)
handlesMouseDown: (NSEvent *) mouseDown
{
    [ self makeWindowFadeInOut: NO ];
}
```

Completo questo paragrafo con il metodo attivato da un clic sul pulsante che mostra un'altra citazione:

```
- (IBAction)
anotherOne:(id)sender
{
```

```
    // costruisco il nuovo fortune
    [ self setLastFortune: [ self makeAFortune ] ];
    // ridisegno la finestra
    [ self resizeWindowToNewFortune ];
    [ theWindow center ];
    // faccio ripartire il timer di morte programmata
    [ fadeTimer invalidate ];
    fadeTimer = [NSTimer scheduledTimerWithTimeInterval: 60.0 target:self
        selector: @selector(endOfDisplay:) userInfo: nil repeats: NO ];
}
```

Sostanzialmente, esegue le stesse operazioni di `awakeFromNib`, ma si ricorda di reimpostare il timer di morte programmata dell'applicazione di nuovo ad un minuto.

RIDIMENSIONAMENTO DELLA FINESTRA

Una cosa divertente che si può fare con la finestra è di dimensionarla opportunamente in modo che contenga comodamente la citazione (che può variare da una a tante linee).

È il compito del metodo `resizeWindowToNewFortune`:

```
- (void)
resizeWindowToNewFortune
{
    NSSize    limit ;
    NSRect    winFrame, prevFrame ;
    // pulisco il testo presentato
    [ theText setStringValue: @"" ];
    // dimensioni del testo rappresentato
    limit = [ lastFortune sizeWithAttributes: curStyle ] ;
    // piglio le dimensioni della finestra
    prevFrame = winFrame = [ theWindow frame ];
    // ricalcolo le nuove dimensioni a partire dalle dimensioni
    // del testo da rappresentare
    winFrame.size.width = limit.width + 50 ;
    if ( winFrame.size.width < 400 )
        winFrame.size.width = 400 ;
    winFrame.size.height = limit.height + 80 ;
    if ( winFrame.size.height < 160 )
        winFrame.size.height = 160 ;
    // il nuovo punto origine si sposta per via delle nuove dimensioni
    winFrame.origin.x -= 0.5 * (winFrame.size.width - prevFrame.size.width)
;
            winFrame.origin.y   -=   0.5   *   (winFrame.size.height   -
prevFrame.size.height) ;
    // imposto il nuovo frame eseguendo una animazione
    [ theWindow setFrame: winFrame display: YES animate: YES ];
    // inserisco il nuovo testo al suo posto
    [ theText setStringValue: lastFortune ];
}
```

Punto fondamentale è di conoscere le dimensioni occupate dalla stringa da visualizzare. Dopo lungo penare (come al solito), ho trovato il metodo `sizeWithAttributes:` che fa proprio al caso mio: calcola un `NSSize` con lo spazio occupato in orizzontale e verticale. Uso questi due valori per ridimensionare il frame della finestra (i 50 pixel in orizzontale e gli 80 in verticale servono a tenere conto dello spazio ai lati del campo di testo); evito comunque di avere una finestra troppo piccola limitando il valore inferiore delle

dimensioni calcolate.

Avendo cambiato le dimensioni della finestra, ed intendendo mantenere la finestra stessa al centro (più o meno) dello schermo, devo riposizionare anche il punto di origine della finestra stessa; è sufficiente spostare l'origine di metà della differenza tra le vecchie e le nuove dimensioni.

Per cambiare le dimensioni della finestra, uso il metodo `setFrame:display:animate:` che esegue un vezzoso effetto di ridimensionamento dinamico. Ho scoperto che la velocità di ridimensionamento è normalmente fissa. Però, avendo già fatto una sottoclasse della finestra `YahWinCtrl`, è un giochino modificarla. È sufficiente sovrascrivere il seguente metodo:

```
- (NSTimeInterval)
animationResizeTime:(NSRect)newFrame
{
    return ( 0.5 );
}
```

Adesso l'applicazione *YahFortune* mi sembra abbastanza graziosa; ma ci sono ancora molte cose che si possono migliorare.

Capitolo 067 - Un pannello delle preferenze

L'applicazione *yahFortune* ha poche possibilità di personalizzazione. Le aggiungo introducendo un elemento delle *Preferenze di Sistema*, oltre che a perfezionare il tutto con molte altre funzionalità.

Sorgenti: documentazione Apple, e poi fortune nelle varie accezioni

Prima stesura: 26 dicembre 2004.

AGGIORNAMENTO FORTUNE

La prima cosa che faccio è di aggiornare o allineare la versione di fortune con il nuovo rilascio italiano e con la versione BSD del tutto. In termini pratici, preleto da Sunsite o da uno dei suoi mirror i sorgenti più o meno ufficiali di fortune:

```
ftp://sunsite.unc.edu/pub/Linux/games/amusements/fortune/fortune-mod*
```

Io ho trovato il file `fortune-mod-9708` che risale appunto all'agosto 97 (ma che è comunque l'ultima disponibile). Rispetto alla versione utilizzata in precedenza, c'è un file in meno di cui tenere conto. Sostituisco i file precedenti con questi nuovi e, già che ci sono, faccio alcune modifiche cosmetiche in modo da eliminare alcuni warning di compilazione (sono un po' fanatico al riguardo). Sostanzialmente, ho introdotto qui e lì i prototipi delle funzioni. Più importante invece il fatto che i path dove rintracciare i file fortune sono adesso incorporati nel programma (e non in un file header separato) tramite due #define:

```
#define     FORTDIR     "./fortunes"
#define     OFFDIR      "./fortunes/off"
```

La directory `off` contiene i file di fortune dal contenuto così detto *offensivo*, che trattano di religione, sesso, minoranze, eccetera, non sempre in maniera garbata (è una delle peculiarità del politicamente corretto proveniente dagli stati uniti...). Per evitare che qualche utente si senta turbato, sono inseriti in file a parte, ed occorre attivamente spiegare al programma che si intendono ricevere fortune anche da questi file.

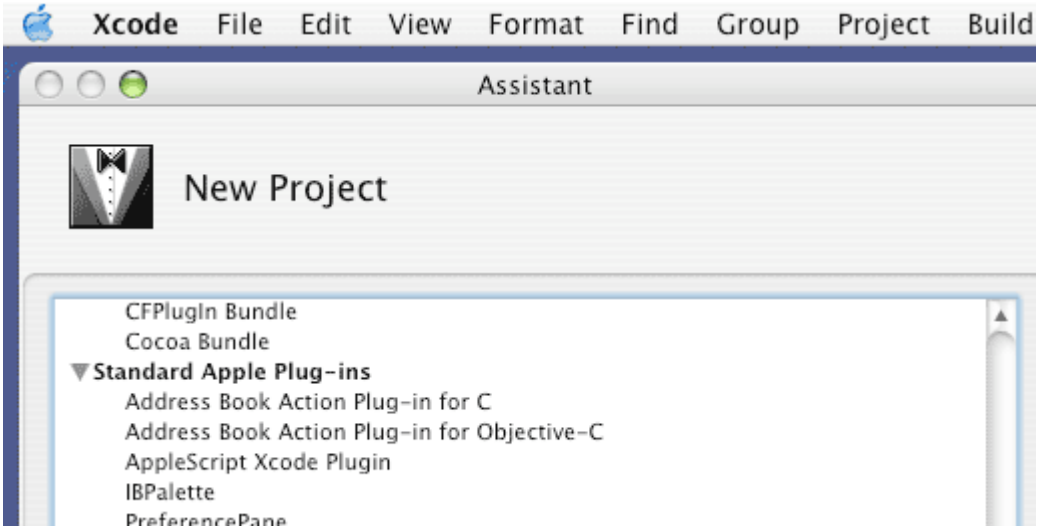
Più interessante invece è l'incorporazione all'interno dell'applicazione di tutti i fortune in lingua italiana, prelevati dal sito ufficiale fortune-it. In questo modo l'applicazione conta su una nutrita serie di fortune già incorporati nell'applicazione stessa, e non ha bisogno di file esterni.

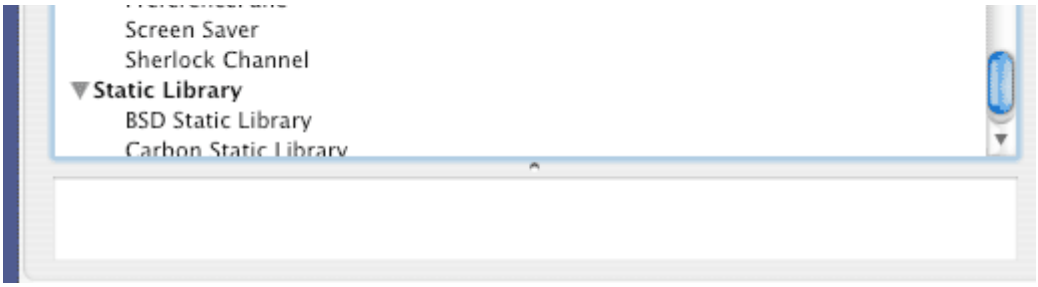
LE PREFERENZE

Dicevo che l'applicazione è piuttosto spartana e che non permette nessuna personalizzazione. Avevo davanti a me due strade: rendere l'applicazione uguale a tutte le altre (ricordo che adesso è una applicazione background only, che non ha menu e non

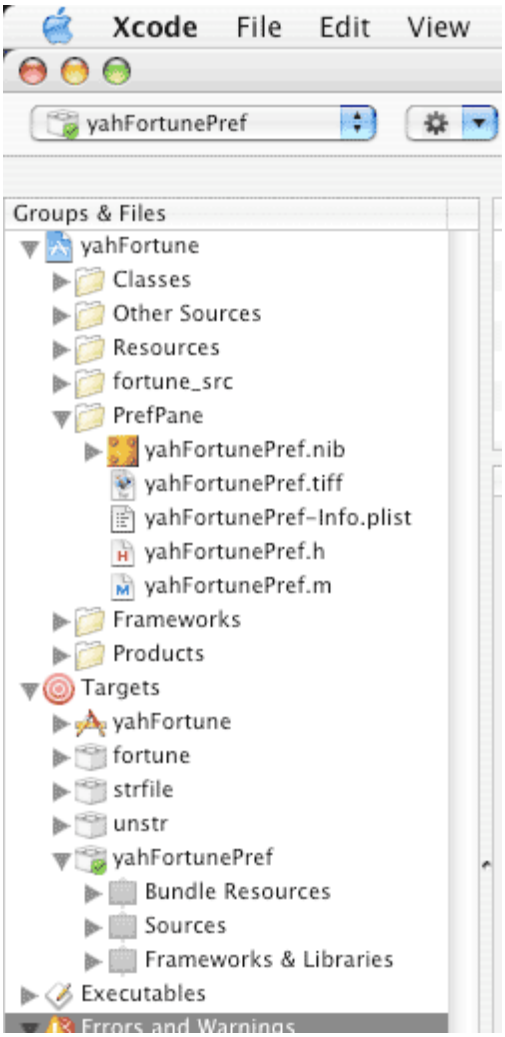
compare nel dock), dotandola di un menu di preferenze; oppure, imparare qualcosa di nuovo, estendendo l'applicazione *Preferenze di Sistema* con un nuovo panello per impostare alcuni valori di personalizzazione. Dato lo scopo di questo sito, è ovvia la scelta della seconda alternativa, anche se probabilmente non si tratta dell'opzione più economica, felice e facile.

Costruire un pannello delle preferenze da aggiungere all'applicazione Preferenze di Sistema è tutto sommato piuttosto facile. Occorre preparare un bundle che abbia l'estensione `prefPane`, e che la classe principale sia una sottoclasse di `NSPreferencePane`. Occorre includere esplicitamente il framework `PreferencePanes.framework`, dal momento che non è incorporato di default nelle applicazioni Cocoa.





In effetti io mi sono complicato la vita in quanto ho voluto mantenere l'applicazione YahFortune ed il pannello delle preferenze all'interno dello stesso progetto, in qualità di nuovo target. Se avessi optato per un diverso progetto, tra le varie opzioni fornite da XCode c'è proprio *Preference Pane*, che predispone un template adatto alla bisogna. A partire da questo template, le cose sono piuttosto semplici: c'è un nib dove disegnare l'interfaccia utente, e una classe corrispondente per controllare le varie opzioni. Nel mio caso, invece, ho dovuto costruire un nuovo target (ormai è il quinto all'interno di uno stesso progetto), al quale riportare i nuovi file di codice, nib, ed un nuovo e diverso file `Info.plist` (che non posso usare quello dell'applicazione, visto il diverso utilizzo). Infine, un nuovo eseguibile, l'applicazione Preferenze di Sistema, per poter lanciare e debuggare il pannello direttamente all'interno di XCode.



Tenendo presente il solito paradigma MVC, la sottoclasse di `NSPreferencePane` funziona da controller. Ovviamente i controlli presenti nel nib sono gli oggetti view; la

parte relativa al modello dei dati è il file delle preferenze, all'interno del quale sono appunti riportati i valori correnti di alcune grandezze. Passando attraverso la classe controller, le grandezze sono visualizzate in forma acconcia all'utente, in modo che possa eventualmente modificarle. Perché la cosa funzioni correttamente, il file delle preferenze deve essere condiviso tra il pannello stesso e l'applicazione. In questo modo l'applicazione accede (tipicamente in sola lettura) al file per leggerne il contenuto, mentre il pannello nelle Preferenze di Sistema permette di modificarle.

Dicevo che ho preferito inserire il pannello all'interno dello stesso progetto utilizzato per la costruzione dell'applicazione. Così facendo, riesco a tenere meglio sotto controllo le interazioni tra le due entità, ed in particolare l'accesso al file delle preferenze. Anzi, ho fatto in modo che il codice di accesso al file fosse sostanzialmente lo stesso, estraendo il tutto in un file condiviso tra il pannello e l'applicazione. Descrivo il file e lo header relativo, in modo da illustrare anche le varie grandezze (che fanno riferimento a nuove o diverse caratteristiche dell'applicazione).

Ho costruito una funzione per accedere al file delle preferenze ed inserire il tutto all'interno di un dizionario:

```
NSMutableDictionary *
getPrefFromFile ( )
{
        NSUserDefaults                          *  theDefs  =  [NSUserDefaults
standardUserDefaults];
       NSMutableDictionary             *  thePrefs  =  [  NSMutableDictionary
dictionaryWithCapacity: 1 ] ;
      NSMutableDictionary            *  currPrefs  =  [  NSMutableDictionary
dictionaryWithCapacity: 1 ] ;
    // recupero le preferenze e le inserisco in un dizionario
    [ thePrefs addEntriesFromDictionary: [ theDefs persistentDomainForName:
PFC_yahf_prefFileName ] ];
    // se uno dei valori non c'e', non c'e' il file
    if ( [ [ thePrefs valueForKey: PFC_yahf_version ]intValue] <= 0 )
    {
        // ed allora lo costruisco con i valori di default
        [ currPrefs addEntriesFromDictionary: defaultPrefValues( ) ];
        // e lo scrivo per la prima volta
                    [  theDefs  setPersistentDomain:  currPrefs  forName:
PFC_yahf_prefFileName ];
        [ theDefs synchronize ];
    }
    else
    {
        // preparo un dizionario con i valori estratti
        [ currPrefs addEntriesFromDictionary: thePrefs ];
    }
    // restituisco comunque il dizionario coi valori delle preferenze
    return ( currPrefs ) ;
}
```

Qui ci sono un paio di concetti notevoli. Il primo concetto è l'accesso al file delle preferenze avviene, per una applicazione in maniera piuttosto trasparente, utilizzando la classe `NSUserDefaults` ed i suoi metodi. Il nome del file da utilizzare è ottenuto dal file `Info.plist` (che però dovrà essere diverso nei due casi). Tutto funziona finché si parla di applicazioni, ma in un caso ho da trattare un pannello delle preferenze, che utilizza di default un altro file (il file delle preferenze di sistema, ovviamente). Indico quindi esplicitamente il nome del file (quasi...) atttraverso la define

`PFC_yahf_prefFileName`, che sta per la stringa `it.macocoa.djzero00.yahFortune`.

Utilizzo la stessa tecnica del capitolo 64 per verificare l'esistenza effettiva del file delle preferenze, controllando la presenza della grandezza `version`. Se assente, il file non esiste e lo devo costruire per la prima volta. Allo scopo, c'è una nuova funzione che produce un dizionario con i valori di default:

```
NSMutableDictionary *
defaultPrefValues ( )
{
          NSMutableDictionary            *  np  =  [  NSMutableDictionary
dictionaryWithCapacity: 9 ];
    NSFont                  * df = [ NSFont fontWithName: @"Georgia" size:
12 ] ;
    // versione
    [ np setObject: [ NSNumber numberWithInt: 1] forKey: PFC_yahf_version ]
;
    // font
    [ np setObject: [ df fontName] forKey: PFC_yahf_fontName ] ;
     [ np setObject: [ NSNumber numberWithFloat: [ df pointSize]] forKey:
PFC_yahf_fontSize ] ;
    // fading
         [  np  setValue:  [  NSNumber  numberWithFloat:  0.1F  ]  forKey:
PFC_yahf_fadeInSpeed ] ;
         [  np  setValue:  [  NSNumber  numberWithFloat:  0.1F  ]  forKey:
PFC_yahf_fadeOutSpeed ] ;
    // cycleTime
        [  np  setValue:  [  NSNumber  numberWithFloat:  60.0F  ]  forKey:
PFC_yahf_cycleTime ] ;
         [  np  setValue:  [  NSNumber  numberWithBool:  YES  ]  forKey:
PFC_yahf_timeBehaviour ] ;
    // dimensioni della finestra, limiti dei tempi
        [  np  setValue:  [  NSNumber  numberWithFloat:  400.0F  ]  forKey:
PFC_yahf_winMinWidth ] ;
        [  np  setValue:  [  NSNumber  numberWithFloat:  160.0F  ]  forKey:
PFC_yahf_winMinHeight ] ;
        [  np  setValue:  [  NSNumber  numberWithFloat:  2.0F  ]  forKey:
PFC_yahf_maxFadeIn ] ;
        [  np  setValue:  [  NSNumber  numberWithFloat:  2.0F  ]  forKey:
PFC_yahf_maxFadeOut ] ;
        [  np  setValue:  [  NSNumber  numberWithFloat:  20.0F  ]  forKey:
PFC_yahf_minCycleTime ] ;
        [  np  setValue:  [  NSNumber  numberWithFloat:  300.0F  ]  forKey:
PFC_yahf_maxCycleTime ] ;
    // colori
      [ np setObject: [ NSArchiver archivedDataWithRootObject: [ NSColor
blackColor ] ] forKey: PFC_yahf_fgColor ];
      [ np setObject: [ NSArchiver archivedDataWithRootObject: [ NSColor
whiteColor ] ] forKey: PFC_yahf_bgColor ];
    // gestione dei file di fortune
         [  np  setValue:  [  NSNumber  numberWithBool:  NO  ]  forKey:
PFC_yahf_equalProb ] ;
      [ np setValue: [ NSNumber numberWithInt: PFC_OU_NORMAL ] forKey:
PFC_yahf_useOffensive ] ;
         [  np  setValue:  [  NSNumber  numberWithInt:  120  ]  forKey:
PFC_yahf_lenghtLimit ] ;
    [ np setValue: [ NSNumber numberWithInt: PFC_LO_NOLENGHTLIMIT ] forKey:
PFC_yahf_lenghtOption ] ;
        [  np  setValue:  [  NSNumber  numberWithFloat:  20.0F  ]  forKey:
PFC_yahf_minFortuneLen ] ;
```

```
        [ np setValue: [ NSNumber numberWithFloat: 1000.0F ] forKey:
PFC_yahf_maxFortuneLen ] ;
    // all'inizio non ci sono file aggiuntivi
    return ( np );
}
```

Ho introdotto una gran quantità di preferenze, ma sono tutte piuttosto semplici da spiegare. Ci sono preferenze che determinano carattere e dimensione del testo con cui è visualizzato il fortune; preferenze per determinare il tempo di *fade* in apertura e chiusura; ho introdotto la possibilità che la finestra rimanga sempre aperta e cambi il fortune ad intervalli di tempo fissati, oppure che si chiuda dopo un certo tempo. Ho introdotto all'interno del file le dimensioni minime della finestre e valori estremali per i tempi; c'è la possibilità di determinare il colore del testo e dello sfondo della finestra, si può personalizzare il tipo di fortune impartendo alcune direttive specifiche al comando (uso o meno di fortune offensivi, messaggi di lunghezza specifica, maggiore o minore di un certo numero di caratteri). Ho previsto, ma non ancora realizzato, la possibilità di utilizzare file esterni all'applicazione come sorgenti di fortune (sarà per un prossimo capitolo).

LA NUOVA YAHFORTUNE

L'introduzione delle preferenze ha ovviamente e pesantemente modificato l'applicazione YahFortune. Ecco le modifiche principali.

Il metodo `init` si è molto semplificato, dato l'utilizzo delle funzioni viste sopra:

```
- (id)
init
{
    self = [super init];
    if ( self )
    {
        NSFont        * tmpFont ;
        // ricavo le preferenze dal file e le inserisco in un dizionario
        [ self setCurrPrefs: getPrefFromFile( ) ] ;
           tmpFont = [ NSFont     fontWithName: [ currPrefs valueForKey:
PFC_yahf_fontName ]
                                          size: [ [ currPrefs valueForKey:
PFC_yahf_fontSize ] intValue] ] ;
        // costruisco un dizionario per mettere dentro il font
        curStyle = [ [NSMutableDictionary alloc] initWithCapacity:1] ;
        [ curStyle setObject: tmpFont forKey: NSFontAttributeName];
        [ curStyle retain ] ;
        // inizializzo le altre variabili
        currentFade = 0 ;
        // predispongo un fortune
        [ self setLastFortune: [ self makeAFortune ] ];
    }
    return self;
}
```

Ho eliminato la maggior parte delle variabili d'istanza, rimpiazzate con l'accesso diretto all'unica variabile `currPrefs`, un dizionario che le contiene tutte. In questo modo cambiano, ma solo per l'aspetto gli altri metodi.

```
- (void)
awakeFromNib
{
    NSColor    * tmpcolor ;
    // per fare in modo che la finestra sia comunque
    // davanti a tutte le altre, anche se l'applicazione
    // non ha menu e non e' presente in dock
    [ NSApp activateIgnoringOtherApps:YES];
    // la rendo invisibile
    [ theWindow setAlphaValue: currentFade ];
    // la porto davanti a tutti
    [ theWindow makeKeyAndOrderFront: self ];
    // imposto il font del campo
    [ theText setFont: [ curStyle objectForKey:NSFontAttributeName ] ];
    // colore del font
    tmpcolor = [ NSUnarchiver unarchiveObjectWithData:
            [ currPrefs valueForKey: PFC_yahf_fgColor ] ] ;
    [ theText setTextColor: tmpcolor ];
    [ theStaticText setTextColor: tmpcolor ];
    // colore dello sfondo
    tmpcolor = [ NSUnarchiver unarchiveObjectWithData:
            [ currPrefs valueForKey: PFC_yahf_bgColor ] ] ;
    [ theWindow setBackgroundColor: tmpcolor ];
    // ridimensiono e centro la finestra
    [ self resizeWindowToNewFortune ];
    [ theWindow center ];
    // faccio partire il processo di fading
    [ self makeWindowFadeInOut: YES ];
}
```

Ho aggiunto qualche istruzione per la gestione dei colori del testo e dello sfondo. Sono cambiati anche i metodi per il fade della finestra, che devono tenere conto del tempo richiesto dalle preferenze, e soprattutto devono gestire la chiusura dell'applicazione o il cambio automatico del fortune.

```
- (void)
updateAlpha: (NSTimer *)timer
{
    // recupero la direzioen del fading
    BOOL    dir = [ [ timer userInfo] boolValue ];
    // se YES, fading in crescendo
    if ( dir )
    {
        // fading da Zero a Uno, apparizione
           currentFade += (BASIC_INTERVAL_TIME / [[ currPrefs valueForKey:
PFC_yahf_fadeInSpeed ] floatValue ] ) ;
        // se raggiungo o supero 1, ho finito
        if ( currentFade >= 1 )
        {
                        float       cycleTime  =  [[  currPrefs  valueForKey:
PFC_yahf_cycleTime ] floatValue ] ;
            // uccido il timer
            [ fadeTimer invalidate ];
            // imposto 1 per essere sicuro
            currentFade = 1 ;
            // imposto un nuovo timer, con il tempo di ciclo/chiusura
             fadeTimer = [NSTimer scheduledTimerWithTimeInterval: cycleTime
target:self
                  selector: @selector(endOfDisplay:) userInfo: nil repeats:
NO ];
        }
```

```
    }
    else
    {
        // fading da Uno a Zero, sparizione
           currentFade -= (BASIC_INTERVAL_TIME / [[ currPrefs valueForKey:
PFC_yahf_fadeOutSpeed ] floatValue ] ) ;
        // se sotto o uguale a zero, ho finito
        if ( currentFade <= 0 )
        {
            // uccido il timer
            [ fadeTimer invalidate ];
            // ma tanto vale: uccido anche l'applicazione
            [ NSApp terminate: self ] ;
        }
    }
    // aggiorno la finestra con il nuovo valore alpha
    [ theWindow setAlphaValue: currentFade ];
}
```

La chiusura/generazione di un nuovo fortune è gestita in maniera molto semplice dal metodo

```
- (void)
endOfDisplay: (NSTimer *)timer
{
    if ( [[ currPrefs valueForKey: PFC_yahf_timeBehaviour ] boolValue ] )
    {
        // chiusura della finestra,
        // faccio ripartire il fading in dissolvenza
        [ self makeWindowFadeInOut: NO ];
    }
    else
    {
        [ self anotherOne: self ];
    }
}
```

Anche il metodo `makeAFortune` è stato rivisto, per tenere conto delle possibilità insite nel comando di fortune, configurate tramite le preferenze e l'inserimento di parametri di lancio:

```
- (NSString *)
makeAFortune
{
    ...
    // la directory di lavoro e' quella di destinazione
              [   fortuneTask   setCurrentDirectoryPath:   [   fortunePath
stringByDeletingLastPathComponent] ];
    // aggiungo eventuali parametri di lancio
    switch ( [[ currPrefs valueForKey: PFC_yahf_useOffensive ] intValue ] )
{
    case PFC_OU_NORMAL :
        break ;
    case PFC_OU_ADDOFFENSIVE :
        [ argList addObject: @"-a" ];
        break ;
    case PFC_OU_ONLYOFFENSIVE :
        [ argList addObject: @"-o" ];
        break ;
    }
```

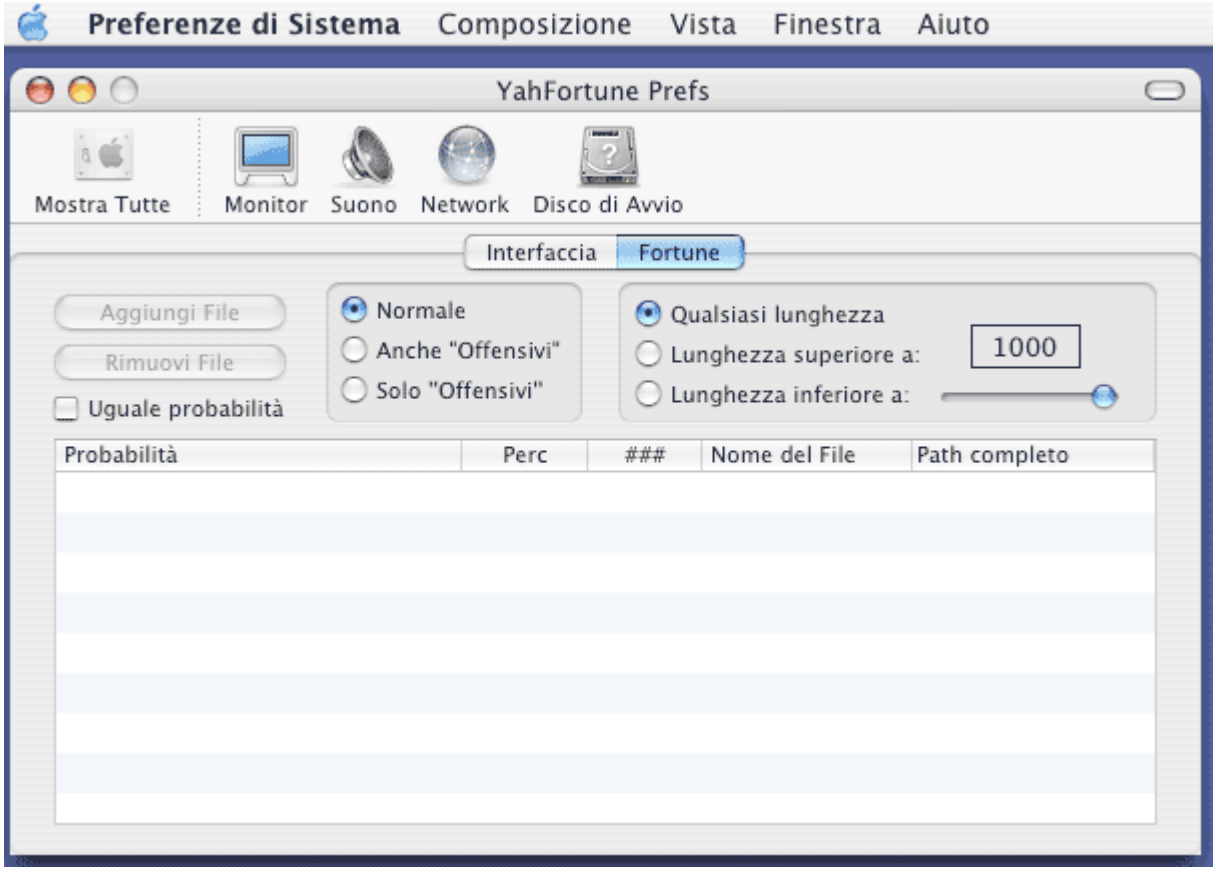
```
    switch ( [[ currPrefs valueForKey: PFC_yahf_lenghtOption ] intValue ] )
{
    case PFC_LO_NOLENGHTLIMIT :
        break ;
    case PFC_LO_LENGHTLONGER :
        [ argList addObject: @"-l" ];
        [ argList addObject: [ NSString stringWithFormat: @"-n %d",
            [[ currPrefs valueForKey: PFC_yahf_lenghtLimit ] intValue ] ]
];
        break ;
    case PFC_LO_LENGHTSHORTER :
        [ argList addObject: @"-s" ];
        [ argList addObject: [ NSString stringWithFormat: @"-n %d",
            [[ currPrefs valueForKey: PFC_yahf_lenghtLimit ] intValue ] ]
];
        break ;
    }
    if ( [ currPrefs valueForKey: PFC_yahf_equalProb ] )
        [ argList addObject: @"-e" ];
    // file aggiuntivi
//    [ argList addObject: @"10%" ];
//    [ argList addObject: @"/path/to/file" ];
    [ fortuneTask setArguments: argList ];
    // dico che l'uscita del comando va su una pipe
    ...
}
```

Con tutte queste modifiche , l'applicazione YahFortune è diventata più presentabile.

Ma adesso manca tutta la parte che permette di effettuare la configurazione senza modificare direttamente i file delle preferenze con un editor di testo.

IL PANELLO

Le preferenze da impostare sono talmente tante che le ho divise in due gruppi.



Il primo gruppo comprende le grandezze in grado di configurare l'interfaccia; sono quindi presenti i controlli per determinare i colori del testo e dello sfondo, slider per assegnare i tempi di apertura, chiusura e di aggiornamento del fortune. C'è il solito (vedi capitolo 64) pulsante per impostare il carattere da utilizzare.

Nel secondo gruppo sono invece presenti le opzioni proprie di fortune; per ora mi limito a determinare l'uso o meno dei fortune offensivi (assenti, assieme agli altri, solo offensivi) e la scelta dei fortune in base al numero di caratteri di cui sono composti (qualsiasi fortune, solo quelli lunghi, solo quelli corti, dove la soglia della lunghezza è determinata attraverso uno slider). C'è poi la possibilità di considerare i file dei fortune equiprobabili, ma qui non mi avventuro (per il momento).

Dal punto di vista del programmatore, devo scrivere una sottoclasse di `NSPreferencePane`, che funziona da controller. Inserisco quindi in questa classe tutti gli outlet verso i controlli dell'interfaccia e le azioni scatenate dai vari controlli. Con *Interface Builder* effettuo tutti i collegamenti, e sono pronto per codificare.

Ora, la logica del pannello dovrebbe essere la seguente: all'apertura, devo predisporre tutti i controlli in modo che rispecchino i valori contenuti nel file delle preferenze; alla chiusura (ed alla selezione da parte dell'utente di un altro pannello) devo prelevare i valori correnti impostati dall'utente per riscriverli nel file delle preferenze. Ma questa non è una applicazione, e non ho a disposizione il metodo `awakeFromNib` per la prima parte o di chiusura del per la seconda.

Però `NSPreferencePane` mette a disposizione una serie di metodi, equivalenti o dalla funzione simile, per eseguire queste operazioni.

Il primo metodo si chiama `mainViewDidLoad` ed è chiamato quando l'applicazione Preferenze di Sistema ha giusto caricato dal file nib tutti i vari controlli; è questo il punto dove aggiustarne l'aspetto. è un metodo lungo ma assolutamente lineare nel suo svolgimento:

```
- (void)
mainViewDidLoad
{
    BOOL                tmpBool ;
    int                    tmpInt ;
    NSMutableArray        * fl ;
    // ricavo le preferenze dal file e le inserisco in un dizionario
    [ self setCurrPrefs: getPrefFromFile( ) ] ;
    // predispongo l'interfaccia con i valori caricati
    // imposto il font corrente
    [ self setUserFont:
        [ NSFont fontWithName: [ currPrefs valueForKey: PFC_yahf_fontName ]
                        size: [ [ currPrefs valueForKey: PFC_yahf_fontSize ]
intValue] ] ];
    // i colori di sfondo e del testo (che sono archiviati come NSData)
    [ fgColor setColor: [ NSUnarchiver unarchiveObjectWithData:
            [ currPrefs valueForKey: PFC_yahf_fgColor ] ] ];
    [ bgColor setColor: [ NSUnarchiver unarchiveObjectWithData:
            [ currPrefs valueForKey: PFC_yahf_bgColor ] ] ];
    // il testo di esempio
    [ fontSample setStringValue: [ NSString stringWithFormat: @"%@ %4.1f",
                    [ userFont displayName ], [ userFont pointSize ] ] ];
    [ fontSample setFont: userFont ] ;
    [ fontSample setTextColor: [ fgColor color ] ];
    [ fontSample setBackgroundColor: [ bgColor color ] ];
    // gli slider per il fade
    [ fadeInSlider setMinValue: 0 ];
    [ fadeInSlider setMaxValue:
            [[ currPrefs valueForKey: PFC_yahf_maxFadeIn ] floatValue] ];
    [ fadeInSlider setFloatValue:
            [[ currPrefs valueForKey: PFC_yahf_fadeInSpeed ] floatValue] ];
    [ fadeInText setFloatValue: [ fadeInSlider floatValue ] ];
    [ fadeOutSlider setMinValue: 0 ];
    [ fadeOutSlider setMaxValue:
            [[ currPrefs valueForKey: PFC_yahf_maxFadeOut ] floatValue] ];
    [ fadeOutSlider setFloatValue:
              [[ currPrefs valueForKey: PFC_yahf_fadeOutSpeed ] floatValue]
];
```

```
    [ fadeOutText setFloatValue: [ fadeOutSlider floatValue ] ];
    // lo slider per il tempo di ciclo/chiusura
    [ cycleTimeSlider setMinValue:
             [[ currPrefs valueForKey: PFC_yahf_minCycleTime ] floatValue]
];
    [ cycleTimeSlider setMaxValue:
             [[ currPrefs valueForKey: PFC_yahf_maxCycleTime ] floatValue]
];
    [ cycleTimeSlider setFloatValue:
            [[ currPrefs valueForKey: PFC_yahf_cycleTime ] floatValue] ];
    [ cycleTimeText setFloatValue: [ cycleTimeSlider floatValue ] ];
     tmpBool = [[ currPrefs valueForKey: PFC_yahf_timeBehaviour ] boolValue
] ;
    // e la gestione dei radio button
    [ timeBehaviour selectCellWithTag: ( ( tmpBool ) ? 0 : 1 ) ] ;
    // probabilita' uguali o meno
    tmpBool = [[ currPrefs valueForKey: PFC_yahf_equalProb ] boolValue ] ;
    [ equalProbBtn setState: ( tmpBool ? NSOnState : NSOffState )];
    // uso dei fortune offensivi
    tmpInt = [[ currPrefs valueForKey: PFC_yahf_useOffensive ] intValue ] ;
    [ offensiveUse selectCellWithTag: tmpInt ];
    // selezione della lunghezza del fortune
    tmpInt = [[ currPrefs valueForKey: PFC_yahf_lenghtOption ] intValue ] ;
    [ qualeLung selectCellWithTag: tmpInt ];
    [ fortuneLenght setMinValue:
             [[ currPrefs valueForKey: PFC_yahf_minFortuneLen ] floatValue]
];
    [ fortuneLenght setMaxValue:
             [[ currPrefs valueForKey: PFC_yahf_maxFortuneLen ] floatValue]
];
    [ fortuneLenght setFloatValue:
            [[ currPrefs valueForKey: PFC_yahf_lenghtLimit ] floatValue] ];
    [ fortuneLenTxt setFloatValue: [ fortuneLenght floatValue ] ];
    // recupero, se esiste, la lista dei file aggiuntivi
    fl = [ currPrefs valueForKey: PFC_yahf_fortuneFiles ] ;
    if ( fl != nil )
    {
        [ self setFortuneFiles: fl ];
        fl = [ currPrefs valueForKey: PFC_yahf_fortunePercs ] ;
        [ self setFortunePercs: fl ];
    }
    else
    {
        [ self setFortuneFiles: [ NSMutableArray arrayWithCapacity: 0] ];
        [ self setFortunePercs: [ NSMutableArray arrayWithCapacity: 0] ];
    }
}
```

Come per l'applicazione, i valori del file delle preferenze sono contenuti in un dizionario, che è costruito come variabile d'istanza, usando la stessa funzione `getPrefFromFile()`.

Per ogni controllo presente, esiste almeno un valore nel file delle preferenze che permette di predisporne un aspetto acconcio. Per gli slider, ad esempio, oltre al valore corrente, è disponibile anche il valore massimo (ed anche in un caso il valore minimo) da impostare.

Per la chiusura delle operazioni, esiste, tra gli altri, il metodo `willUnselect`, che è chiamato da Preferenze di Sistema quando l'utente seleziona un altro pannello o chiude l'applicazione stessa, prima di passare ad un altro pannello; uso questo metodo perché

all'istante di chiamata sono ancora presenti e validi tutti i controlli, e quindi prelevo le informazioni direttamente. Anche qui, il metodo è lungo ma non difficile:

```
- (void)
willUnselect
{
       NSUserDefaults                        *  theDefs  =  [NSUserDefaults
standardUserDefaults];
    // per qualche motivo non posso usare direttamente currPrefs
      NSMutableDictionary             *  thePrefs  =  [  NSMutableDictionary
dictionaryWithCapacity: 1 ] ;
    // un nuovo dizionario con tutti i valori di default
    [ thePrefs addEntriesFromDictionary: currPrefs ];
    // sostituisco tutti quelli che possono essere stati cambiati
    [ thePrefs    setObject: [ userFont fontName]
                forKey: PFC_yahf_fontName ] ;
      [  thePrefs      setObject:  [  NSNumber  numberWithFloat:  [  userFont
pointSize] ]
                forKey: PFC_yahf_fontSize ] ;
     [  thePrefs     setValue:  [  NSNumber  numberWithFloat:  [  fadeInSlider
floatValue ] ]
                forKey: PFC_yahf_fadeInSpeed ] ;
     [  thePrefs     setValue:  [  NSNumber  numberWithFloat:  [  fadeOutSlider
floatValue ] ]
                forKey: PFC_yahf_fadeOutSpeed ] ;
     [  thePrefs     setValue:  [  NSNumber  numberWithFloat:  [  cycleTimeSlider
floatValue ] ]
                forKey: PFC_yahf_cycleTime ] ;
     [  thePrefs     setValue:  [  NSNumber  numberWithBool:  ([  timeBehaviour
selectedTag ] == 0 ? YES : NO) ]
                forKey: PFC_yahf_timeBehaviour ] ;
    // i colori vanno salvati come NSData
     [  thePrefs     setObject:  [  NSArchiver  archivedDataWithRootObject:  [
fgColor color ] ]
                forKey: PFC_yahf_fgColor ];
     [  thePrefs     setObject:  [  NSArchiver  archivedDataWithRootObject:  [
bgColor color ] ]
                forKey: PFC_yahf_bgColor ];
    // gestione dei file
      [  thePrefs     setValue:  [  NSNumber  numberWithBool:  ([  equalProbBtn
state ] == NSOnState) ]
                forKey: PFC_yahf_equalProb ] ;
      [  thePrefs      setValue:  [  NSNumber  numberWithInt:  [  offensiveUse
selectedTag ] ]
                forKey: PFC_yahf_useOffensive ] ;
       [  thePrefs       setValue:  [  NSNumber  numberWithInt:  [  qualeLung
selectedTag ] ]
                forKey: PFC_yahf_lenghtOption ] ;
     [  thePrefs     setValue:  [  NSNumber  numberWithFloat:  [  fortuneLenght
floatValue ] ]
                forKey: PFC_yahf_lenghtLimit ] ;
    // salvo il file delle preferenze al suo posto
     [ theDefs setPersistentDomain: thePrefs forName: PFC_yahf_prefFileName
];
    [ theDefs synchronize ];
}
```

A questo punto, ci sono solo un paio di semplici metodi da scrivere per facilitare l'interazione dell'utente. Ad esempio, uso metodi espliciti di gestione degli slider per poter mantenere accoppiati lo slider ed il testo corrispondente, e verificare che il testo inserito rispetti i limiti. Per lo slider che attribuisce la soglia della lunghezza del fortune

da utilizzare la cosa è semplice:

```
- (IBAction)
updateFortuneSlider: (id) sender
{
    float    tmpFlt, lim1, lim2 ;
    switch ( [ sender tag ] ) {
    case 101 :    // slider
        // solo numeri interi
        [ fortuneLenTxt setIntValue: [ fortuneLenght intValue ] ];
        break;
    case 102 :    // text
        tmpFlt = (float) [ fortuneLenTxt intValue ] ;
              lim1  =  [[  currPrefs  valueForKey:  PFC_yahf_minFortuneLen  ]
floatValue] ;
              lim2  =  [[  currPrefs  valueForKey:  PFC_yahf_maxFortuneLen  ]
floatValue] ;
        if ( tmpFlt < lim1 )    tmpFlt = lim1 ;
        if ( tmpFlt > lim2 )    tmpFlt = lim2 ;
        [ fortuneLenght setIntValue: (int) tmpFlt ];
        break;
    default:
        break ;
    }
}
```

Come sempre, ho attribuito opportuni codici di tag al testo ed allo slider, in modo che uso un unico metodo per entrambi. Nel caso del testo, verifico che il valore immesso sia un intero compreso tra il massimo ed il minimo ammissibile. Prelevo questi due estremi dal file delle preferenze, ma solo per mio sfizio di generalità (avrei potuto usare dei numeri fissati una volta per tutte all'interno del codice). Il vantaggio è che un utente smaliziato li può modificare a mano (non esiste infatti un controllo nel pannello) senza per questo modificare il codice; ritengo comunque una buona pratica inserire tante più informazioni possibili all'interno di file esterni all'applicazione, in modo da evitare modifiche sul codice per operazioni magari banali.

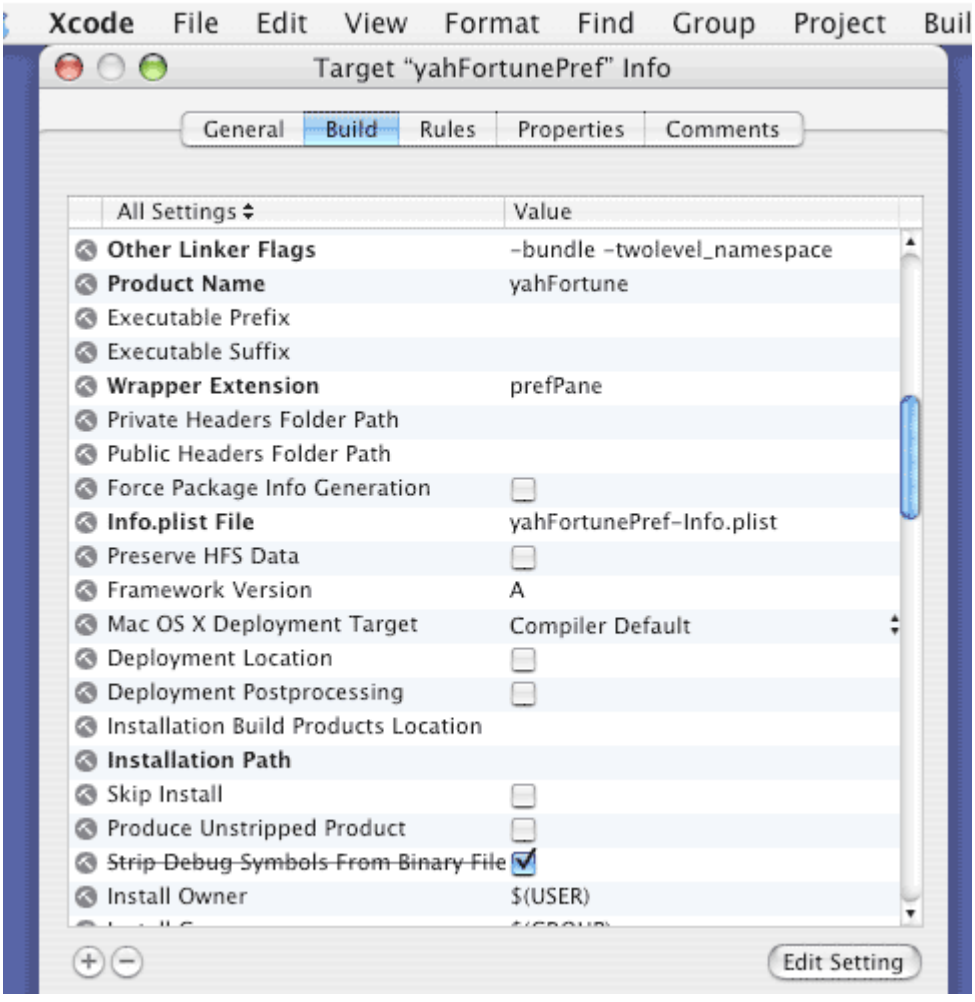
Come per il capitolo 64, anceh qui c'è un pulsante per visualizzare il pannello di scelta caratteri, ed un corrispondente metodo che modifica opportunamente un testo per verificare l'aspetto del risultato.

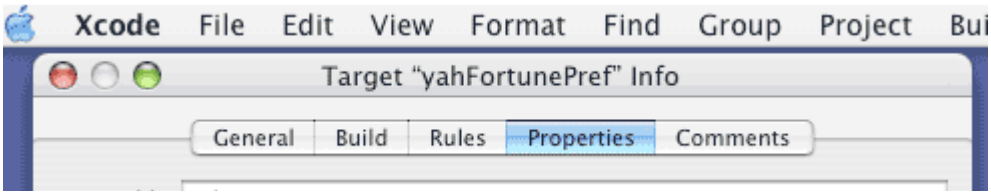
```
- (IBAction)
selectFont:(id) sender
{
    [ [ NSFontPanel sharedFontPanel] orderFront: nil ];
    [ [ NSFontPanel sharedFontPanel] setDelegate: self ];
}

- (void)
updatePreferences:(id)sender
{
    NSString    * thestr ;
    // in pratica, cambiamento di colore e font
    // aggiorno la stringa visualizzata
       thestr  =  [  NSString  stringWithFormat:  @"%@  %4.1f",  [  userFont
displayName ], [ userFont pointSize ] ] ;
      [  fontSample  setFont:  [  NSFont  fontWithName:  [  userFont  fontName  ]
size: 12 ] ] ;
    [ fontSample setStringValue: thestr ];
    [ fontSample setTextColor: [ fgColor color ] ];
```

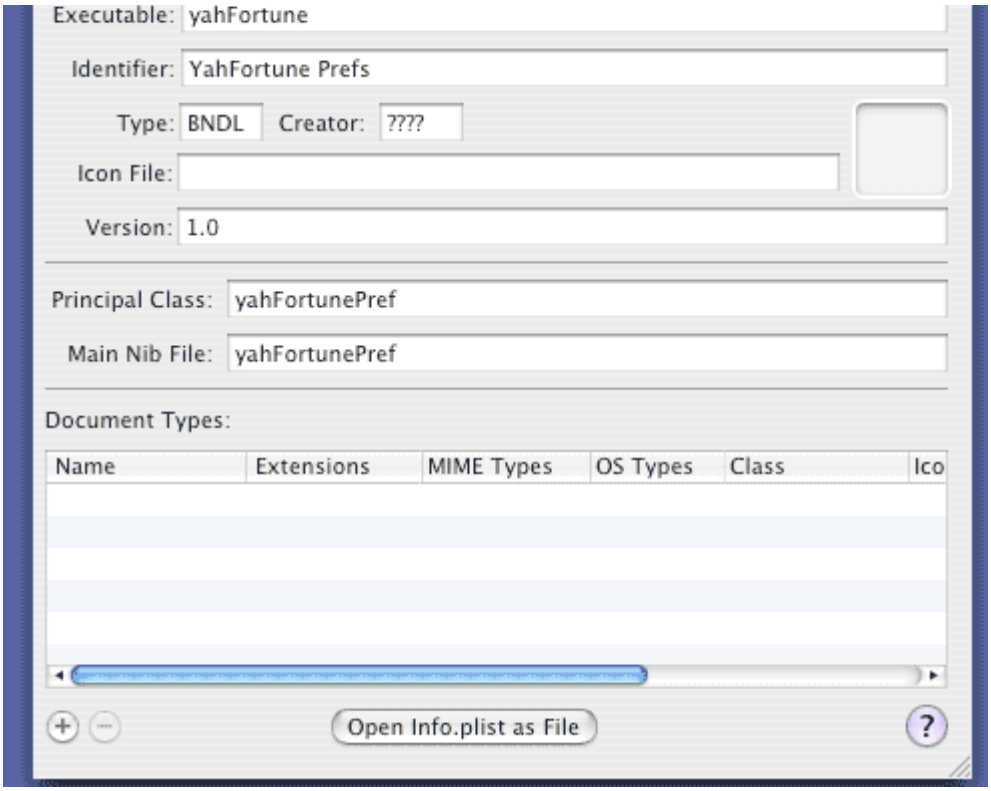
```
    [ fontSample setBackgroundColor: [ bgColor color ] ];
}

- (void)
changeFont:(id)sender
{
    // ottengo il nuovo font
    NSFont        * newFont = [sender convertFont:userFont] ;
    // lo imposto internamente
    [ self setUserFont: newFont ];
    [ self updatePreferences: sender ];
}
```

L'azione `selectFont` è associata al pulsante di selezione, il metodo `changeFont` è chiamato dal pannello di scelta font, mentre il metodo `updatePreferences` modifica l'aspetto di un campo di testo per dare conto dell'avvenuta modifica del carattere e del colore di sfondo.

ULTIME AVVERTENZE







La parte più difficile del tutto è stata configurare il target del pannello delle preferenze in modo che producesse appunto un pannello delle preferenze. In effetti non occorrono molte operazioni, ma queste sono fondamentali. Mi sono sicuramente dimenticato qualcosa: per certo il nuovo target deve essere un *bundle*, e questo bundle deve avere una wrapper extension pari a `prefPane`. Il nome del target, della classe controller e del file è sostanzialmente lo stesso, `yahFortunePref`, ed ho aggiunto un file `yahFortunePref-info.plist` come file delle info. Tutto ciò ha richiesto la modifica delle informazioni del Target, utilizzando la finestra che si apre con la voce di menu apposita. Sono da modificare voci nel pannello *Build* e nel pannello *Properties*.

All'interno dei file del progetto si trovano altri metodi, per la gestione della tabella dei file aggiuntivi, che non sono ancora attivi. Saranno commentati più avanti, quando avrò capito come fare. All'interno del file che contiene il progetto sono presenti anche i sorgenti del comando fortune (privati dei fortune in inglese per ragioni di spazio) e i file accessori del pacchetto fortune-IT; i file di fortune veri e propri sono già incorporati all'interno del progetto (che proprio per questo è piuttosto corposo).

Capitolo 068 - Fortuna stravolta

Ho parecchio stravolto l'applicazione *YahFortune* ed il pannello delle preferenze relativo; ho introdotto nuove funzioni, e razionalizzato quelle già presenti. Piuttosto che esaminare tutto il codice nel dettaglio (che spesso è rimasticatura di concetti già esposti), mi concentro sui punti più interessanti.

Sorgenti: documentazione Apple

Prima stesura: 30 dicembre 2004

SCRITTURA DI TAGLINE

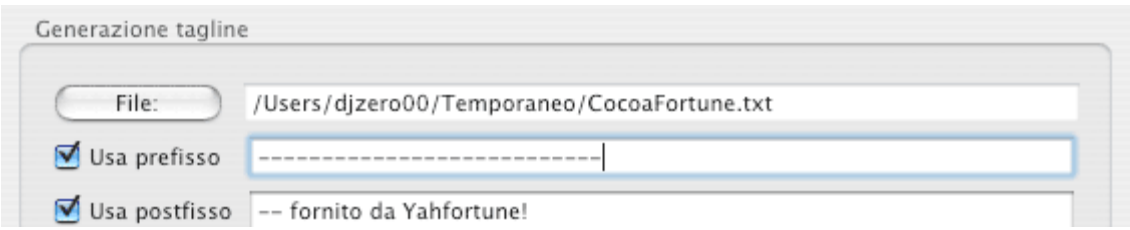
Una *tagline* è una serie di linee di testo aggiunte in coda ad un messaggio di posta, che può contenere o meno informazioni utili al ricevente (che so, indirizzo o telefono del mittente). Spesso però la tagline è un pretesto per aggiungere un motto, una frase spiritosa, qualcosa insomma che caratterizzi il mettente. Talvolta è carino che la tagline non sia sempre la stessa, ma che possa cambiare nel tempo, più o meno casualmente. La maggior parte dei programmi di posta (*Mail*, *Thunderbird*, eccetera) ha la possibilità di inserire automaticamente una tagline, meno programmi invece hanno la possibilità di generarla automaticamente e casualmente. Però Thunderbird utilizza come tagline il contenuto di un file esterno all'applicazione. Se esiste una qualche applicazione in grado di modificare il contenuto di tale file, ecco che Thunderbird è in gado di inserire una tagline casuale all'interno dei messaggi.

La possibilità di inserire un messaggio di fortune all'interno di uno specifico file è proprio una della caratteristiche desiderate fin dall'inizio per la mia applicazione YahFortune. Si tratta di una cosa piuttosto facile da realizzare.

Per prima cosa, inserisco nuove grandezze nel file delle preferenze. Il nome del file è ovviamente la prima cosa, poi aggiungo due valori booleani per inserire o meno una riga di prefisso e di postfisso (si dirà così) al fortune generato. Va da sé che poi devo aggiungere nelle preferenze le due stringhe desiderate. In questo modo, inserendo sia il prefisso che il postfisso, il contenuto del file risulta qualcosa del tipo:

```
---------------
"Non permettete a un vogon, per nessuna ragione al mondo, di leggervi
le sue poesie."
        -- Douglas Adams, "Guida Galattica per gli Autostoppisti"
-- fornito da Yahfortune!
```

Oltre ad inserire le grandezze nelle preferenze, devo ovviamente provvedere a gestirle; aggiungo quindi gli opportuni controlli e la loro gestione nel panello delle preferenze.



Non c'è del codice particolarmente notevole per gestire il tutto: associato al pulsante di scelta file c'è un metodo per selezionare un file (il cui nome sarà inserito nel campo a fianco, mentre ai due check button è associato un metodo per abilitare o meno il campo di testo associato.

All'interno dell'applicazione YahFortune, invece, ho inserito un nuovo metodo, chiamato dal metodo `makeAFortune` in occasione della generazione di un nuovo fortune, che si occupa della scrittura del testo su di un file.

```
- (void )
writeFortuneOnfile: (NSString *) theString
{
    NSString     * ft ;
    NSString     * fn ;

    // se arrivo qui, devo scrivere su file
           fn   =   [[   currPrefs   objectForKey:   PFC_yahf_tagTextFile   ]
stringByExpandingTildeInPath ];
    if ( [[ currPrefs valueForKey: PFC_yahf_usePrefix ] boolValue ] )
    {
        NSString     * s1 ;
        s1 = [ currPrefs objectForKey: PFC_yahf_tagTextPrefix ] ;
        ft = [ NSString stringWithFormat: @"%@\n%@", s1, theString ];
    }
    else    ft = [ NSString stringWithString: theString ] ;
    if ( [[ currPrefs valueForKey: PFC_yahf_usePostfix ] boolValue ] )
    {
        NSString     * s2 ;
        s2 = [ currPrefs objectForKey: PFC_yahf_tagTextPostfix ] ;
        ft = [ NSString stringWithFormat: @"%@ %@\n", ft, s2 ];
    }
    [ ft writeToFile: fn atomically: YES ];
}
```

Come si può vedere, è piuttosto semplice. L'argomento del metodo è il nuovo fortune appena generato. A seconda del valore delle preferenze, si aggiungono il testo di prefisso ed il testo postfisso, per poi inserire il risultato ottenuto all'interno del file specificato.

LANCIO ED ARRESTO DELL'APPLICAZIONE

Mi piacerebbe gestire l'applicazione direttamente dall'interno del pannello delle preferenze. In altre parole, vorrei lanciare l'applicazione YahFortune dall'interno del pannello, verificarne lo stato, ed eventualmente arrestarne l'esecuzione.

Allo scopo ho aggiunto un pulsante, che svolge una duplice funzione. Il testo del pulsante indica l'azione che si può eseguire, e così facendo informa anche sullo stato dell'applicazione. Se il testo è *Avvia YahFortune*, significa che l'applicazione non è in esecuzione; facendo clic sul pulsante, l'applicazione verrà allora lanciata. Il pulsante diventa quindi *Arresta YahFortune*, indicando che l'applicazione sta lavorando onestamente. Facendo nuovamente clic, l'applicazione termina la propria esistenza.

Mi sono trovato subito in difficoltà. Infatti, il metodo `launchedApplications` di `NSWorkspace`, che pensavo di usare, ha un difetto (se così posso esprimermi):

nell'elenco delle applicazioni restituite non sono presenti le applicazioni *background only*, come appunto è YahFortune.

Piuttosto che impegolarmi su altre strade d'interazione con il sistema operativo (ad esempio, avrei potuto eseguire il comando unix `ps -ax`, cercare nella lista risultante YahFortune, cose del genere), utilizzo un altro costrutto, la cui esistenza ho appreso dalla documentazione Apple, proprio nel tutorial dedicato ai Pannelli delle Preferenze.

Si tratta della classe `NSConnection`, che mette a disposizione un meccanismo perché due oggetti in diversi thread o task possano comunicare tra loro. Copio abbastanza pedissequamente dall'esempio, quindi non aspettatevi spiegazioni approfondite. L'idea è che un oggetto dichiari la propri disponibilità ad eseguire servizi. Nel metodo `awakeFromNib` dell'applicazione YahFortune aggiungo allora questo pezzo di codice:

```
    theConnection = [NSConnection defaultConnection];
    [ theConnection setRootObject: self];
    [ theConnection registerName: CONNECTION_PREF2APPL ];
```

Qui dico che la classe `self` (cioé `ftrWinCtrl`) si rende disponibile come *root object* ad eseguire dei servizi (ovvero, dei metodi...); per accedere a questi servizi si deve effettuare una connessione utilizzando il nome `CONNECTION_PREF2APPL` (che ho scelto astrusamente complicato per evitare che altre applicazioni in giro per il mondo abbiamo lo stesso nome).

Dall'altra parte, all'interno del pannello delle preferenze, per verificare se l'applicazione YahFortune è in esecuzione, basta che controlli se la connessione è disponibile: ho un metodo che si occupa della cosa:

```
- ( BOOL )
checkIfYFIsRunning
{
    NSConnection            * tc;
     tc = [ NSConnection connectionWithRegisteredName: CONNECTION_PREF2APPL
host:nil];
    if ( tc == nil )
        return ( NO ) ;
    // e' vero' che c'e' l'applicazione, ma potrebbe essere in chiusura...
    return ( YES );
}
```

C'è un piccolo problema (l'applicazione potrebbe sì essere in esecuzione, ma essere nella fase di chiusura, con il fading della finestra), ma che non ha conseguenze, come si vedrà più avanti.

Uso poi una variabile d'istanza per tenere traccia di questo stato. Per tanto, all'apertuan del pannello, all'interno del metodo `mainViewDidLoad` ho aggiunto queste due istruzioni:

```
    appIsActive = [ self checkIfYFIsRunning ] ;
    [ self adjustLaunchButton ] ;
```

La seconda istruzione chiama il metodo seguente per aggiustare l'aspetto del pulsante:

```
- (void)
adjustLaunchButton
{
    if ( appIsActive )
        [ launchYF setTitle: @"Arresta YahFortune" ];
    else
        [ launchYF setTitle: @"Avvia YahFortune" ];
}
```

A questo punto, il metodo scatenato dal clic sul pulsante è:

```
- (IBAction)
handleLaunchStopYF: (id) sender
{
    if ( appIsActive )
    {
        // arresto l'applicazione: le mando un messaggio apposito
        ...
    }
    else
    {
        // lancio l'applicazione
        [[ NSWorkspace sharedWorkspace ] launchApplication: @"YahFortune" ]
;
    }
}
```

Non è presente il codice per terminare l'applicazione; ci arrivo in un attimo. Infatti, è il sottoprodotto della soluzione alla questione di riconoscere quando l'applicazione YahFortune è lanciata in esecuzione o terminata in maniera indipendente dal pannello delle preferenze.

Sfrutto ancora il concetto di `NSConnection`, però dalla parte opposta. Nel pannello delle preferenze dichiaro la disponibilità ai servizi all'interno del metodo di inizializzazione (dove ho spostato anche la lettura delle preferenze)

```
- (id)
initWithBundle:(NSBundle *)bundle
{
    if ( ( self = [super initWithBundle:bundle] ) != nil )
    {
        NSConnection    * theConnection ;
        // ricavo le preferenze dal file e le inserisco in un dizionario
        [ self setCurrPrefs: getPrefFromFile( ) ] ;
        // rendo disponibile un connessione in ingresso
        theConnection = [NSConnection defaultConnection];
        [ theConnection setRootObject: self];
        [ theConnection registerName: CONNECTION_APPL2PREF ];
    }
    return self;
}
```

Poi, all'interno dell'applicazione YahFortune, al lancio della stessa (nel metodo `awakeFromNib`, per esempio, esegue le poche linee di codice:

```
    // verifico se c'e' il prefpane
      if  (  (theConnection  =  [  NSConnection  connectionWithRegisteredName:
```

```
CONNECTION_APPL2PREF host:nil] ))
    {
          yahFortunePref     * theYahFortPref = [[theConnection rootProxy]
retain];
        [ theYahFortPref applicationNotification: YES ];
    }
```

Così facendo, eseguo il metodo `applicationNotification`, con argomento YES. Parallelamente, ogni volta che l'applicazione YahFortune decide di chiudersi, esegue il seguente segmento di codice:

```
      if  ( (theConnection = [ NSConnection connectionWithRegisteredName:
CONNECTION_APPL2PREF host:nil] ))
    {
          yahFortunePref     * theYahFortPref = [[theConnection rootProxy]
retain];
        [ theYahFortPref applicationNotification: NO ];
    }
    [ NSApp terminate: self ] ;
```

A questo punto, all'interno della classe `yahFortunePref`, nel pannello delle preferenze, basta realizzare il metodo `applicationNotification` in maniera piuttosto semplice:

```
- ( void )
applicationNotification: (BOOL) uot
{
    appIsActive = uot ;
    [ self adjustLaunchButton ] ;
}
```

Con queste poche istruzioni, ho risolto il problema di conoscere lo stato di esecuzione dell'applicazione YahFortune. Con la stessa tecnica, invio anche il messaggio di chiusura dell'applicazione; basta eseguire un metodo opportuno:

```
- (IBAction)
handleLaunchStopYF: (id) sender
{
    if ( appIsActive )
    {
        // arresto l'applicazione: le mando un messaggio apposito
        NSConnection    * tc = [ NSConnection connectionWithRegisteredName:
                                    CONNECTION_PREF2APPL host:nil];
        frtWinCtrl        * theYahFortApp = [[tc rootProxy] retain];
        [ theYahFortApp killMeKillMeKillMe ] ;
    }
    ...
```

All'interno della classe `frtWinCtrl` c'è quindi il metodo:

```
- (void )
killMeKillMeKillMe
{
    [ fadeTimer invalidate ];
              fadeTimer    =    [NSTimer    scheduledTimerWithTimeInterval:
BASIC_INTERVAL_TIME target:self
        selector: @selector(updateAlpha:)
```

```
        userInfo: [ NSNumber numberWithBool: NO ]
        repeats: YES ];
    ...
}
```

In questo modo la prossima operazione eseguita (dopo un minimo intervallo di tempo) è il fading della finestra fino a scomparire (e quindi poi la chiusura del tutto).

Per finire, un piccolo tocco interessante; il pannello modifica il contenuto delle preferenze. Se l'applicazione è aperta, questa non se ne accorge se non al prossimo riavvio, dal momento che il file delle preferenze è letto solo alla partenza e poi basta. Basta allora, sempre con il meccanismo di `NSConnection`, informare l'applicazione dell'avvenuta modifica. Aggiungo, all'interno del metodo `updatePrefFile` le solite istruzioni:

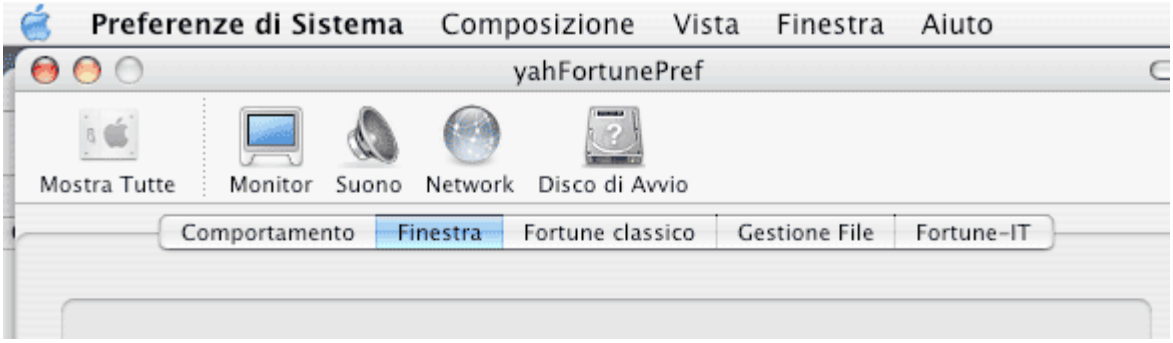
```
    // se l'applicazione e' aperta, le dico di aggiornare le preferenze
      if ( (theConnection = [ NSConnection connectionWithRegisteredName:
CONNECTION_PREF2APPL host:nil] ))
    {
        frtWinCtrl    * theYahFortApp = [[theConnection rootProxy] retain];
        [ theYahFortApp updateWindowDisplay: thePrefs ];
    }
```

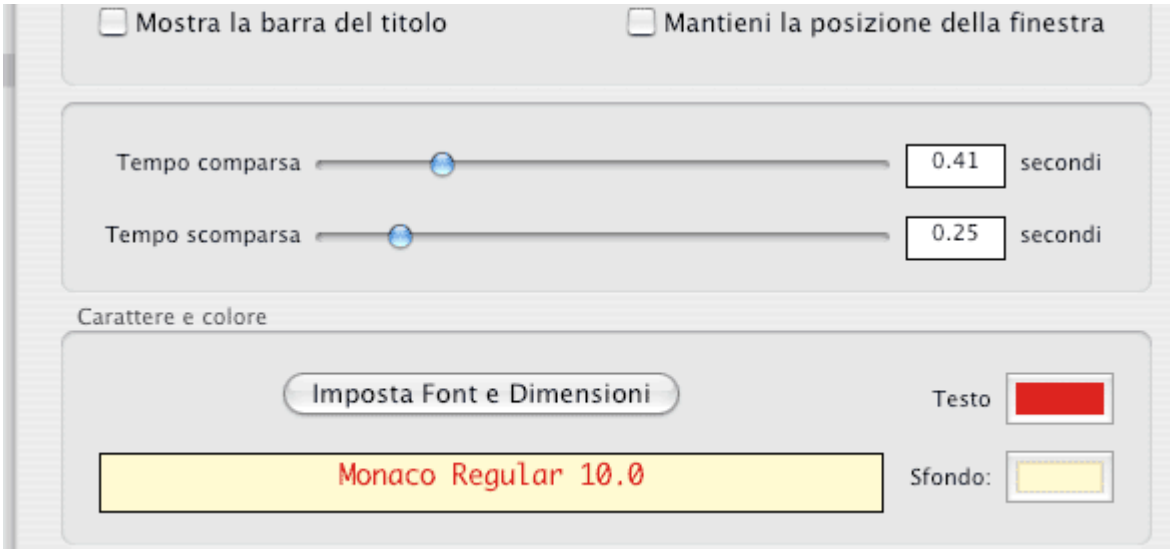
Dall'altra parte, il metodo corrispondente cambia tutte le preferenze, utilizzando i nuovi valori:

```
- ( void )
updateWindowDisplay: (NSMutableDictionary *) theNewPrefs
{
    ...
    [ self setCurrPrefs: theNewPrefs ] ;
     [ self setUserFont: [ NSFont fontWithName: [ currPrefs valueForKey:
PFC_yahf_fontName ]
             size: [ [ currPrefs valueForKey: PFC_yahf_fontSize ] intValue]
] ];
    ...
}
```

METTIAMOCI UN TITOLO

All'inizio del progetto avevo fatto in modo che la finestra di YahFortune non avesse la barra del titolo e che si mantenesse sempre al centro della scena. Andando avanti, mi sono accorto che in effetti potrebbe essere comodo mantenere la barra del titolo, in modo da posizionare la finestra dove si preferisce.





Ho quindi aggiunto due preferenze nel pannello; la prima indica se disegnare o meno il titolo della finestra, la seconda se la finestra sarà sempre centrata, oppure se mantiene la posizione. Se la finestra è sempre centrata, l'effetto di animazione quando cambia il fortune visualizzato coinvolge tutti e quattro i lati della finestra. Se invece la finestra deve mantenere la posizione, allora l'animazione coinvolge solo il lato destro e il lato inferiore. Va da sé che, in baso di mantenimento della posizione, al lancio successivo dell'applicazione la finestra mantiene l'ultima posizione utilizzata in precedenza.

Ora però, non c'è un metodo in `NSWindow` che permetta di aggiungere o togliere il titolo a piacimento. A quanto pare, si può fare solo al momento della creazione, utilizzando il parametro `styleMask`, come avevo fatto nella sottoclasse Yahwindow, forzandolo a `NSBorderlessWindowMask`.

Ma poco fa ho aggiunto la possibilità di modificare le preferenze al volo, con piena accettazione dei nuovi valori alla visualizzazione del fortune successivo. Ho quindi stravolto ancora il meccanismo di costruzione della finestra, che ho raccolto nel metodo seguente, all'interno della classe `frtWinCtrl`:

```
- ( void )
makeANewWindowMoving: (BOOL) titleChange
{
    YahWindow        * tempWindow ;
    NSView            * winContent ;
    unsigned int    winStyle ;
    NSRect            theRect ;
    // faccio scomparire la finestra presente con un fading a zero
    // titleChange indica che ho aggiunto/tolto la barra del titolo,
    // quindi occorre far scomparire la vecchia finestra e poi far
    // comparire la nuova finestra al suo posto
    if ( titleChange )
    {
        while ( currentFade > 0 )
        {
            long    xxx ;
            [ theWindow setAlphaValue: currentFade ];
            Delay ( 1, & xxx );
            currentFade -= 0.2 ;
        }
        currentFade = 0 ;
```

```
        [ theWindow setAlphaValue: currentFade ];
    }
    theRect = [[theWindow contentView] frame] ;
```

Fino a questo punto, ho fatto scomparire graziosamente la finestra corrente: ho utilizzato un ciclo `while` piuttosto che il solito meccanismo col timer, ma il concetto è lo stesso.

Ricordo che `theWindow` è una finestra che è stata prelevata da un file nib, ed ha quindi già tutto predisposto da *Interface Builder*. Adesso invece ne costruisco una del tutto nuova, chiamato direttamente un metodo di inizializzazione con opportuni parametri:

```
    // adatto lo stile a seconda delle preferenze
    if ( [[ currPrefs valueForKey: PFC_yahf_showWinTitle ] boolValue ] )
        winStyle = NSTitledWindowMask | NSClosableWindowMask ;
    else
        winStyle = NSBorderlessWindowMask ;
    // costruisco una nuova finestra fuori schermo
    tempWindow = [ [ YahWindow alloc ]
        initWithContentRect: NSMakeRect( -350, -200, 300, 160 )
        styleMask: winStyle
        backing: [theWindow backingType]
        defer: YES ];
    // aggiunsto un po' di parametri
    [ tempWindow setAlphaValue: currentFade ];
    [ tempWindow setDelegate: self ];
    [ tempWindow setHasShadow: YES ];
    [ tempWindow setBackgroundColor: [ theWindow backgroundColor ] ];
    [ tempWindow setFrameAutosaveName: @"YahFortWin" ];
    // piglio il contenuto della vecchia finestra
    // e lo inserisco in quella nuova
    winContent = [ [ theWindow contentView ] retain ];
    [ winContent removeFromSuperview ];
    [ tempWindow setContentView: winContent ];
    [ winContent release ];
    // sostituisco la finestra vecchia con quella nuova
    [ theWindow release ] ;
    theWindow = [ tempWindow retain ] ;
```

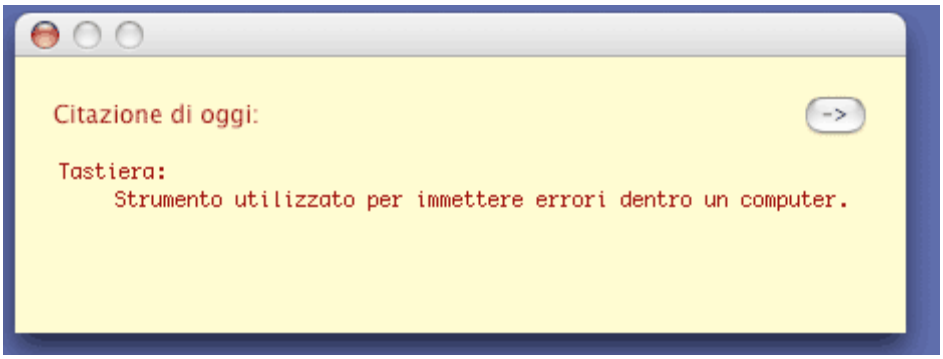
Dopo aver costruito la finestra nuova, ho impostato qualche suo parametro (che sono costretto a fare così, visto che non c'è Interface Builder ad aiutare). Poi, piuttosto che costruire a mano i vari elementi presenti al suo interno, li prelevo direttamente dall'altra finestra (!). Sposto quindi la vista contenuta nella vecchia finestra, con tutte le sue sottoviste (ed anche i collegamenti con le altri classi, outlet e target/Action compresi, impostati tramite Interface Builder), all'interno della nuova finestra. Infine, rilascio la vecchia finestra e tengo buona quella nuova.

```
    // vedo se e' il caso di centrarla
    if ( [[ currPrefs valueForKey: PFC_yahf_remeberWinPos ] boolValue ] )
    {
        // recupero la posizione dal file delle preferenze
        [ theWindow setFrameUsingName: @"YahFortWin" ];
    }
    else
    {
        // centro la finestra
        [ theWindow setFrame: theRect display: YES animate: NO ];
```

```
        [ theWindow center ];
    }
    // la porto davanti a tutti
    [ theWindow makeKeyAndOrderFront: self ];
    // la faccio comparire davanti a tutti con un fade
    if ( titleChange )
    {
        while (currentFade < 1 )
        {
            long    xxx ;
            [ theWindow setAlphaValue: currentFade ];
            Delay ( 1, & xxx );
            currentFade += 0.2 ;
        }
        currentFade = 1 ;
        [ theWindow setAlphaValue: currentFade ];
    }
}
```

Il resto del metodo centra se il caso la finestra, e la fa comparire col solito effetto sfumato.

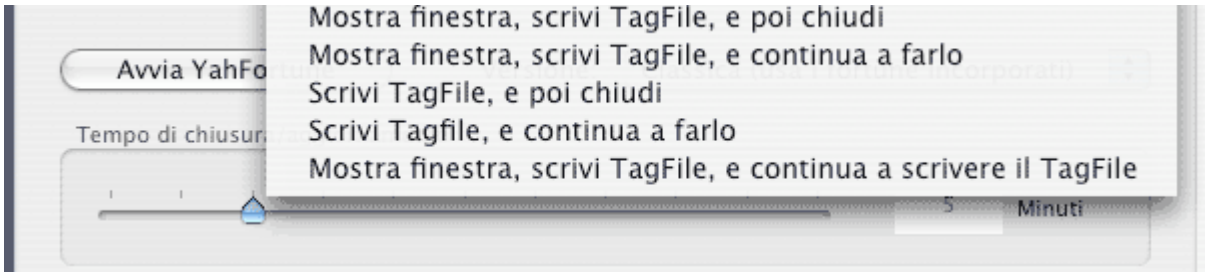
Ciò che scatena tutto questo procedimento è la chiamata del metodo all'interno di `awakeFromNib` (col parametro a NO, dal momento si tratta della prima visualizzazione e non c'è bisogno del meccanismo di fading) e poi ogni volta che si modifica la preferenza relativa alla barra del titolo (questa volta col parametro a YES)



Il funzionamento non è ancora corretto in tutti i casi (sto pasticciando con la questione della posizione, che non so ancora bene cosa voglio fare), ma mi pare un buon punto d'inizio.

DIVERSO COMPORTAMENTO

La modifica più appariscente dell'applicazione è il fatto che ho individuato sette diverse modalità di comportamento, che vado nell'ordine ad elencare:



- Generazione di un fortune, presentazione dello stesso in una finestra, e tutto finisce lì. è la modalità classica: si mette l'applicazione in avvio automatico al login, ed ogni volta che accende il Mac, eccovi una frase memorabile.

- Generazione di un fortune, presentazione in finestra, daccapo. è un metodo comodo per perdere tempo, leggendo un po' di frasi più o meno celebri.

- Come la prima, con in più la scrittura del fortune su un file. Utile se si intende utilizzare il programma come generatore di tagline, e non si intende aggiornare la tagline troppo spesso.

- Come la seconda, con la scrittura del fortune su un file. Alla ricerca della perfetta tagline per la propria posta.

- Generazione di un fortune, scrittura dello stesso su di un file; niente finestra. Per avere una tagline nuova ad ogni login, senza lasciare finestre aperte

- Generazione continua di fortune e scrittura degli stessi su file. Ottimo per avere una tagline casuale; si imposta il tempo e si usa un apposito programma di posta.

- L'applicazione parte, mostra un fortune in una finestra, poi chiude la finestra, ma l'applicazione non muore: continua a scrivere periodicamente nuovi fortune su di un file; combina il precedente comportamento con il primo, unendo l'utile al dilettevole.

Per di più, con la storia di aggiornare le preferenze e farle recepire subito dall'applicazione, si può anche saltare dall'una all'altra modalità.

In tutta questa confusione, ho praticamente riscritto la logica di funzionamento dell'applicazione YahFortune (e giusto per farvi vedere quanto sono bravo, dirò che si tratta di una *macchina a stati finiti*).

La storia dell'applicazione comincia col metodo `initWithBundle`, che ho già illustrato sopra. Ho trasferito qui per buona misura la lettura del file delle preferenze. Poi si carica il file nib e viene eseguito il metodo `awakeFromNib`. Come ho descritto sopra, qui la finestra caricata dal nib è buttata via, e al suo posto ne viene costruita un'altra con le caratteristiche desiderate. Se il pannello delle preferenze è attivo (c'è una connessione), gli si comunica che l'applicazione è viva ed in salute.

Sempre all'interno del metodo `awakeFromNib` c'è la prima discriminazione sul comportamento da tenere:

```
    // adesso distinguo il comportamento a seconda del codice
    switch ( [[ currPrefs valueForKey: PFC_yahf_timeBehaviour ] intValue ]
) {
```

```
    case PFC_BT_WINDOW_ONESHOT :
    case PFC_BT_WINTAG_ONESHOT :
    case PFC_BT_WINTAG_FIRST :
    case PFC_BT_WINTAG_CONTINUE :
    case PFC_BT_WINDOW_CONTINUE :
        // c'e' da visualizzare la finestra
        // genero il fortune e preparo la finestra
        [ self resizeWindowToNewFortune ];
        // faccio partire il processo di fading
                  fadeTimer  =  [NSTimer  scheduledTimerWithTimeInterval:
BASIC_INTERVAL_TIME target:self
            selector: @selector(updateAlpha:)
            userInfo: [ NSNumber numberWithBool: YES ]
            repeats: YES ];
        break ;
    case PFC_BT_TAG_ONESHOT :
    case PFC_BT_TAG_CONTINUE :
        // c'e' da gestire solo il file con la tagline
        // genero il fortune
        [ self makeAFortune ] ;
        // aspetto il tempo imposto
         fadeTimer = [NSTimer scheduledTimerWithTimeInterval: [ self getSec
] target:self
            selector: @selector(endOfDisplay:) userInfo: nil repeats: NO ];
        break ;
    }
```

Se c'è una finestra da visualizzare (i primi cinque casi dello `switch`), ebbene, si genera il fortune, si aggiusta la finestra e si fa partire il timer che chiama il metodo `updateAlpha:`. Questo metodo modifica gradualmente il canale alpha fino ad 1. Quando lo raggiunge, aspetta il tempo indicato nelle preferenze (recuperato dal metodo `getSec`) e poi dice di eseguire il metodo `endOfDisplay:`.

Se non ci sono finestre, allora si genera il fortune, si attende il tempo necessario e poi si passa direttamente al metodo `endOfDisplay`.

Questo metodo è un po' il cuore di tutto il meccanismo:

```
- (void)
endOfDisplay: (NSTimer *)timer
{
     switch ( [[ currPrefs valueForKey: PFC_yahf_timeBehaviour ] intValue ]
) {
    case PFC_BT_WINDOW_ONESHOT :
    case PFC_BT_WINTAG_ONESHOT :
        // chiusura della finestra,
        // faccio ripartire il fading in dissolvenza
                  fadeTimer  =  [NSTimer  scheduledTimerWithTimeInterval:
BASIC_INTERVAL_TIME target:self
            selector: @selector(updateAlpha:)
            userInfo: [ NSNumber numberWithBool: NO ]
            repeats: YES ];
        break;
```

Questo è il caso di di generazione di un solo fortune (con o meno scrittura del file); faccio ripartire il timer che chiama `updateAlpha:` per far svanire la finestra. Alla scomparsa della finestra, l'applicazione termina.

```
    case PFC_BT_WINTAG_FIRST :
        // se ho gia' fatto un passaggio
        if ( shooted )
        {
            // devo solo generare il file
            [ self makeAFortune ] ;
            // aspetto il tempo imposto
               fadeTimer = [NSTimer scheduledTimerWithTimeInterval: [ self
getSec ] target:self
                  selector: @selector(endOfDisplay:) userInfo: nil repeats:
NO ];
        }
        else
        {
            // e' la prima volta che passo di qui, dico che l'ho fatto
            shooted = YES ;
            // faccio partire il fading in dissolvenza
                    fadeTimer  =  [NSTimer  scheduledTimerWithTimeInterval:
BASIC_INTERVAL_TIME target:self
                selector: @selector(updateAlpha:)
                userInfo: [ NSNumber numberWithBool: NO ]
                repeats: YES ];
        }
        break;
```

Questo è il caso più complicato, in cui la finestra è visualizzata una prima volta, ma poi l'applicazione continua a scrivere periodicamente i fortune su un file. Uso una variabile d'istanza aggiuntiva, inizialmente posta a NO. La prima volta che si passa di qui, `shooted` vale NO, quindi si esegue la parte `else` del codice: si mette a YES la variabile, poi comincia la fase di scomparsa della finestra. Questa volta (ci arrivo tra poco), alla scomparsa della finestra, l'applicazione non viene chiusa, me è eseguito nuovamente il metodo `endOfDisplay`. Si ripassa quindi di qui, ma questa volta `shooted` è YES, quindi si genera un fortune, lo si scrive su file, si aspetta il tempo indicato, e si ritorna qui. E via così.

```
    case PFC_BT_WINTAG_CONTINUE :
    case PFC_BT_WINDOW_CONTINUE :
        // ridisegno la finestra
        [ self resizeWindowToNewFortune ];
        // faccio ripartire il timer di ciclo/chiusura
         fadeTimer = [NSTimer scheduledTimerWithTimeInterval: [ self getSec
] target:self
            selector: @selector(endOfDisplay:) userInfo: nil repeats: NO ];
        break ;
    case PFC_BT_TAG_CONTINUE :
        [ self makeAFortune ] ;
        // aspetto il tempo imposto
         fadeTimer = [NSTimer scheduledTimerWithTimeInterval: [ self getSec
] target:self
            selector: @selector(endOfDisplay:) userInfo: nil repeats: NO ];
        break ;
```

Qui le operazioni di generazione di nuovi fortune continuano fino a che l'utente non si stufa. Però c'è una finestra da visualizzare nei primi due casi (metodo `resizeWindowToNewFortune`), mentre è assente nel terzo (semplice `makeAFortune`). Si finisce con l'aspettare il solito tempo e a ripassare di qui.

```
    case PFC_BT_TAG_ONESHOT :
```

```
        // mi limito a chiudere l'applicazione direttamente
        [ self endOftheStory ] ;
        break ;
    }
}
```

L'ultimo caso è quello in cui si scrive il fortune sul file, e poi basta. Eseguo allora il metodo `endOfTheStory`, che comunica al pannello delle preferenze (se disponibile all'ascolto) la chiusura dell'applicazione.

```
- (void )
endOftheStory
{
    NSConnection * theConnection ;
      if  (  (theConnection  =  [  NSConnection  connectionWithRegisteredName:
CONNECTION_APPL2PREF host:nil] ))
    {
          yahFortunePref     * theYahFortPref = [[theConnection rootProxy]
retain];
        [ theYahFortPref applicationNotification: NO ];
    }
    [ NSApp terminate: self ] ;
}
```

C'è infine il metodo `updateAlpha:`, che non ha subito modifiche d'interesse nella parte iniziale (fading verso 1), ma deve tenere conto dei nuovi comportamenti quando fa scomparire la finestra:

```
- (void)
updateAlpha: (NSTimer *)timer
{
    // recupero la direzione del fading
    BOOL    dir = [ [ timer userInfo] boolValue ];
    // se YES, fading in crescendo
    if ( dir )
    {
        ...
    }
    else
    {
        // fading da Uno a Zero, sparizione
          currentFade  -=  (BASIC_INTERVAL_TIME  /  [[  currPrefs  valueForKey:
PFC_yahf_fadeOutSpeed ] floatValue ] ) ;
        // se sotto o uguale a zero, ho finito
        if ( currentFade <= 0 )
        {
            // uccido il timer
            [ fadeTimer invalidate ];
            // se e' il caso speciale di finestra one shot ma tag continuo
                 if  (  [[  currPrefs  valueForKey:  PFC_yahf_timeBehaviour  ]
intValue ] == PFC_BT_WINTAG_FIRST )
            {
                // eseguo subito dopo un nuovo aggiornamento
                       fadeTimer  =  [NSTimer  scheduledTimerWithTimeInterval:
BASIC_INTERVAL_TIME target:self
                           selector: @selector(endOfDisplay:) userInfo: nil
repeats: NO ];
            }
            else
            {
```

```
                // negli altri casi uccido anche l'applicazione
                [ self endOftheStory ] ;
            }
        }
    }
    // aggiorno la finestra con il nuovo valore alpha
    [ theWindow setAlphaValue: currentFade ];
}
```

Quando il valore di `currentFade` è nullo, si chiede se è il caso speciale della finestra che si chiude, ma l'applicazione resta viva (ed allora lancia subito il metodo `endOfdisplay`), oppure è proprio il caso di chiudere tutto e finire qui con `endOftheStory`.

Per ora mi fermo qui, con l'applicazione ed il pannello che non sono in forma smagliante (ci sono ancora alcuni problemi noti, e molti altri ignoti), non prima di notare come adesso sia possibile, con la finestra di fortune in primo piano, copiare il fortune corrente con la combinazione di tasti *Mela C*. Mi è bastato aggiungere in `frtWinCtrl` il metodo seguente

```
- (IBAction)
copy:(id)sender
{
    // accedo alla clipboard
    NSPasteboard    * pb = [NSPasteboard generalPasteboard];
    // dico che nella clipboard ci sara' una stringa
     [ pb declareTypes: [NSArray arrayWithObjects:NSStringPboardType, nil]
owner:nil];
    // inserisco la stringa
    [ pb setString: [ theText stringValue ] forType: NSStringPboardType ];
}
```

La cosa funziona perché la classe è indicata come delegata della finestra.

Capitolo 069 - Continua Fortuna

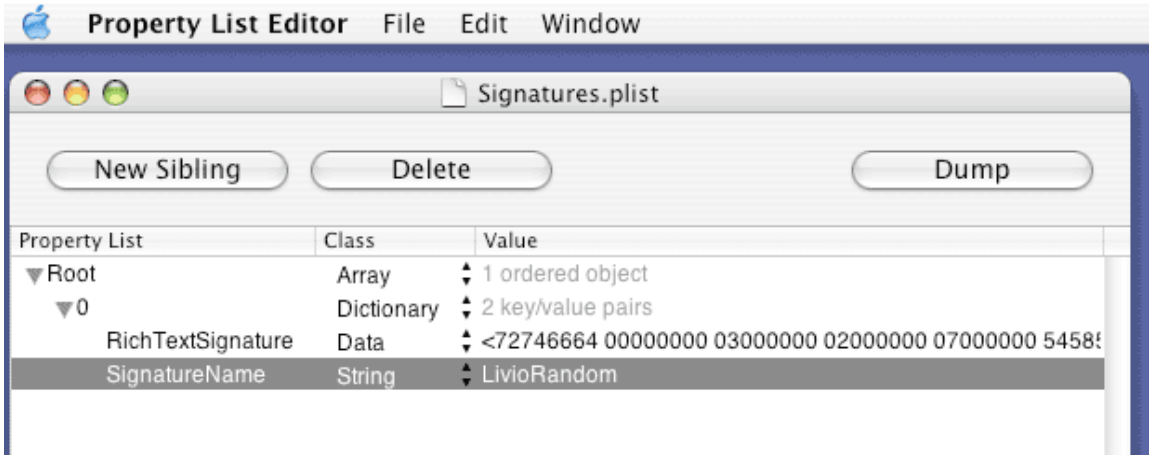
Continuo lo sviluppo di *YahFortune*, aggiungendo nuove funzionalità, modificando quelle presenti, facendo insomma una grande confusione.

Sorgenti: documentazione Apple

Prima stesura: 10 gennaio 2005

Integrazione con Apple Mail

Il motivo per cui tutto ciò è cominciato era l'utilizzo di fortune come generatore più o meno automatico e casuale di tagline da inserire all'interno dei messaggi di posta elettronica. Nel precedente capitolo, la cosa era risolta per tutti quelle applicazioni di posta che basano la propria signature su un file di testo esterno al programma. Ciò non accade purtroppo per il programma di posta più diffuso di Mac OS X, ovvero *Apple Mail*. Le signature di questa applicazione si trovano infatti memorizzate all'interno di un file sì di testo, ma strutturato XML. Il file in questione si chiama `Signatures.plist` e si trova all'interno della cartella `~/Library/Mail`, dove la tilde indica la vostra cartella d'inizio.



Ho cercato a lungo un modo per interpretare tale file (che è la rappresentazione XML di una array di signature, dove ogni signature è un dizionario con due termini, il nome della signature ed il testo vero e proprio). Dopo molto penare, e senza avere una soluzione in tasca buona in ogni occasione, ho deciso di utilizzare un Applescript che piloti direttamente Apple Mail. Ma nemmeno in questa versione funziona come vorrei, per cui presumo che modificherò ancora...

Riporto direttamente il metodo della classe `frtWinCtrl` che esegue il tutto:

```
- ( void)
execScript4Mail: (NSString*) sigName withContent: (NSString *) signature
{
    // costruisco lo script giuntando un po' di stringhe
    NSMutableString * s1 = [ NSMutableString string ];
    // verifico che Mail sia aperto (altrimenti lo script lo apre)
    [ s1 appendString: @"tell application \"Finder\" \n" ];
```

```
    [ s1 appendString: @"set procList to the name of every process \n" ];
    [ s1 appendString: @"if (procList contains \"Mail\") then \n" ];
    // Mail e' aperto
    [ s1 appendString: @"tell application \"Mail\" \n" ];
    // assegno la signature ad una variabile
    [ s1 appendFormat: @"set theSignature to \"%@\" \n", signature ];
    // assegno alla signature col nome indicato il nuovo valore
    [ s1 appendFormat: @"set the content of signature \"%@\" to theSignature
\n", sigName ];
    [ s1 appendString: @"end tell \n" ];
    // chiusura dello script
    [ s1 appendString: @"end if \n" ];
    [ s1 appendString: @"end tell \n" ];
    // costruisco lo script e lo eseguo
    NSAppleScript    * as = [[ NSAppleScript alloc ] initWithSource: s1 ];
    [ [ as executeAndReturnError: nil ] stringValue ] ;
}
```

Parto dal nome della signature e dalla signature stessa, ed uso l'istruzione Applescript

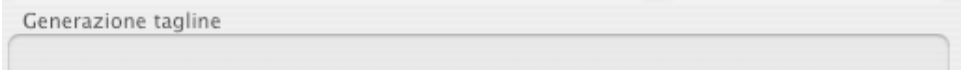
```
    set the content of signature "nomeSig" to "testoSig""
```

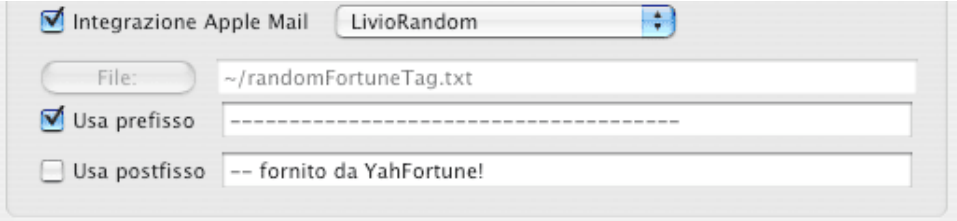
Attorno a questa istruzione, c'è il controllo che l'applicazione sia effettivamente in uso, e la preparazione dell'istruzione stessa a partire dalla stringhe passate come parametri al metodo. Il tutto è concluso dal lancio in esecuzione dello script.

Va da sé che mentre il testo della signature è quello generato dall'applicazione *YahFortune*, occorre conoscere il nome della signature utilizzato. Ovviamente, tale nome si trova all'interno del file delle preferenze. L'integrazione con Mail o la scrittura su file dipende ancora dalle preferenze:

```
- (void )
writeFortuneOnfile: (NSString *) theString
{
    ...
    // distinguo se devo scrivere su di un file o integrarmi con Apple Mail
    if ( [[ currPrefs valueForKey: PFC_yahf_mailIntegration ] boolValue ] )
    {
        // integrazione con Apple Mail; uso la signature col nome indicato
        // dalle preferenze
                  NSString        *  subSig  =  [  currPrefs  valueForKey:
PFC_yahf_mailSignName ] ;
        [ self execScript4Mail: subSig withContent: ft ];
    }
    else
    {
        NSString     * fn ;
        // banale (!) scrittura su di un file
                fn  =  [[  currPrefs  objectForKey:  PFC_yahf_tagTextFile  ]
stringByExpandingTildeInPath ];
        [ ft writeToFile: fn atomically: YES ];
    }
}
```

A questo punto, all'interno del pannello delle preferenze, ho dovuto aggiungere qualche elemento per poter impostare queste preferenze





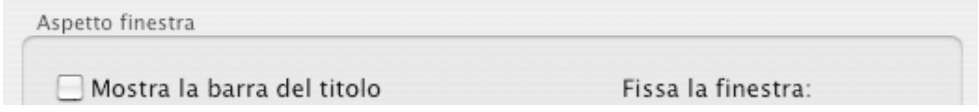
A parte il pulsante di spunta per l'integrazione o meno con Mail, c'è un menu pop-up che permette di scegliere quale signature di mail utilizzare (presuppongo che tale signature esista già; ne segue che YahFortune la modifica). Questo menu deve essere riempito con l'elenco delle signature presenti. L'operazione avviene all'interno del metodo `mainViewDidLoad`. La cosa è più semplice del previsto; sfrutto infatti il fatto che il file `Signature.plist` è una property list che rappresenta un `NSArray` di `NSDictionary` (come lo so? guardando il file, mi sembrava... ho provato... è andata). Quindi il metodo `arrayWithContentsOfFile` è sufficiente per costruire l'oggetto `NSArray` a partire dal file. Poi ho spazzato il vettore e ne ho estratto i nomi dal dizionario usando la chiave apposita. Questi nomi li ho inseriti man mano come voci del menu.

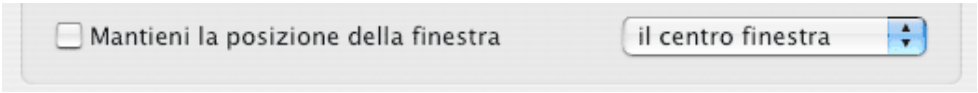
```
- (void)
mainViewDidLoad
{
    ...
    // integrazione con Apple Mail
    tmpBool = [[ currPrefs valueForKey: PFC_yahf_mailIntegration ] boolValue
] ;
    [ mailIntegBtn setState: ( tmpBool ? NSOnState : NSOffState )];
    [ mailSignature removeAllItems ];
    // per l'integrazione con mail, recupero i nomi delle signature
        NSString        *  aPath   =  [  @"~/Library/Mail/Signatures.plist"
stringByExpandingTildeInPath ];
    NSArray        * ams = [ NSArray arrayWithContentsOfFile: aPath ] ;
    for ( tmpInt = 0 ; tmpInt < [ ams count ] ; tmpInt ++ )
    {
        NSDictionary * di = [ ams objectAtIndex: tmpInt ];
                [  mailSignature  addItemWithTitle:  [  di  objectForKey:
@"SignatureName" ]];
    }
        [  mailSignature  selectItemWithTitle:  [  currPrefs  objectForKey:
PFC_yahf_mailSignName ] ];
    // - - - - - - - - - - - - - - - - - -
    ...
```

Quando poi l'utente seleziona una della voci di menu, faccio nulla, o meglio, inserisco all'interno del file delle preferenze il nome selezionato, lo stesso nome che poi è utilizzato dall'applicazione per costruire lo script.

CENTRAGGIO DELLA FINESTRA

Nel capitolo precedente mi lamentavo di non aver ancora deciso di cosa fare della finestra e di come centrarla. Come mio solito, ho stravolto la cosa.





Ho individuato cinque modi per gestire la posizione della finestra, tenendo fisso uno tra i cinque punti notevoli (i quattro angoli, oppure il centro). Ovviamente, la scelta della modalità è come al solito gestita dal pannello delle preferenze, dove è possibile anche indicare se ricordare la posizione della finestra tra un lancio ed il successivo. Non c'è nulla di particolarmente interessante nella gestione del menu pop up, per cui passo direttamente a come l'applicazione gestisce la situazione.

Il metodo `makeANewWindowMoving` è responsabile della costruzione della finestra; è qui dunque che si deve tenere conto della posizione:

```
    ...
            if   (   titleChange   ==   NO   &&   [[   currPrefs   valueForKey:
PFC_yahf_remeberWinPos ] boolValue ] )
    {
        // recupero la posizione dal file delle preferenze
         float     xpos = [[ currPrefs valueForKey: PFC_yahf_winStartXPos ]
floatValue ] ;
         float     ypos = [[ currPrefs valueForKey: PFC_yahf_winStartYPos ]
floatValue ] ;
        // se le posizioni sono negative, centro e non ci penso piu'
        if ( xpos <= 0 || ypos <= 0 )
        {
            [ theWindow center ];
        }
        else
        {
            // cambio le coordinate della finestra
            theRect.origin.x = xpos ;
            theRect.origin.y = ypos - theRect.size.height ;
            [ theWindow setFrame: theRect display: YES animate: NO ];
        }
    }
    else
    {
        // centro la finestra
        [ theWindow setFrame: theRect display: YES animate: NO ];
        [ theWindow center ];
    }
    ...
```

Se occorre ricordare la posizione della finestra, prelevo le coordinate dalle preferenze e, se hanno un valore sensato, cambio le coordinate della finestra in accordo ai valori letti. Altrimenti, piazzo la finestra al centro e non ci penso più.

Inoltre, occorre modificare il metodo `endOfThestory` per tenere traccia dell'ultima posizione nota della finestra:

```
- (void )
endOftheStory
{
    NSConnection * theConnection ;
        NSUserDefaults                          *  theDefs  =  [NSUserDefaults
standardUserDefaults];
       NSMutableDictionary              *  thePrefs  =  [  NSMutableDictionary
```

```
dictionaryWithCapacity: 1 ] ;
    NSRect                    theRect = [ theWindow frame] ;

    // un nuovo dizionario con tutti i valori di default
    [ thePrefs addEntriesFromDictionary: currPrefs ];
    // salvo la posizione corrente della finestra
    [ thePrefs    setValue: [ NSNumber numberWithFloat: theRect.origin.x ]
                forKey: PFC_yahf_winStartXPos ] ;
     [ thePrefs    setValue: [ NSNumber numberWithFloat: (theRect.origin.y +
theRect.size.height )]
                forKey: PFC_yahf_winStartYPos ] ;
    // salvo il file delle preferenze al suo posto
     [ theDefs setPersistentDomain: thePrefs forName: PFC_yahf_prefFileName
];
    [ theDefs synchronize ];
    // dico all'eventuale pannello che l'applicazione muore
      if ( (theConnection = [ NSConnection connectionWithRegisteredName:
CONNECTION_APPL2PREF host:nil] ))
    {
          yahFortunePref     * theYahFortPref = [[theConnection rootProxy]
retain];
        [ theYahFortPref applicationNotification: NO ];
    }
    [ NSApp terminate: self ] ;
}
```

Già che ci sono, faccio notare una modifica al metodo `killMeKillMeKillMe`, che, nel caso, fa scomparire graziosamente la finestra prima di terminare l'applicazione (in precedenza, era piuttosto brutale).

```
- (void )
killMeKillMeKillMe
{
    // c'e' da chiudere la finestra
    [ fadeTimer invalidate ];
    // faccio un fading tutto mio
    while ( currentFade > 0 )
    {
        long    xxx ;
        [ theWindow setAlphaValue: currentFade ];
        Delay ( 1, & xxx );
        currentFade -= 0.2 ;
    }
    currentFade = 0 ;
    [ theWindow setAlphaValue: currentFade ];
    // muoio infelice
    [ self endOftheStory ] ;
}
```

Infine, c'è la questione di come gestire il passaggio da un fortune ad un altro fortune, ovvero di come muovere la finestra. All'interno del metodo `resizeWindowToNewFortune` c'è il meccanismo che tiene conto delle cinque possibilità:

```
    ...
    // il nuovo punto origine si sposta secondo il valore delle preferenze
    switch ( [ [ currPrefs valueForKey: PFC_yahf_growingMode ] intValue ] )
{
    case PCF_GM_CENTERWIN :
                    winFrame.origin.x  -=  0.5  *  (winFrame.size.width  -
```

```
prevFrame.size.width) ;
                    winFrame.origin.y  -=  0.5  *  (winFrame.size.height  -
prevFrame.size.height) ;
        break ;
    case PCF_GM_FIXTOPLEFT :
         winFrame.origin.y -= (winFrame.size.height - prevFrame.size.height)
;
        break ;
    case PCF_GM_FIXTOPRIGHT :
        winFrame.origin.x -= (winFrame.size.width - prevFrame.size.width) ;
         winFrame.origin.y -= (winFrame.size.height - prevFrame.size.height)
;
        break ;
    case PCF_GM_FIXBOTTOMLEFT :
        break ;
    case PCF_GM_FIXBOTTOMRIGHT :
        winFrame.origin.x -= (winFrame.size.width - prevFrame.size.width) ;
        break ;
    }
    ...
```

Le coordinate della finestra sono modificate in accordo al punto fisso individuato, ed il codice mostra il tutto in maniera abbastanza esplicativa; c'è solo da tener presente che le coordinate della finestra sono date a partire dall'angolo in basso a sinistra dello schermo.

Ho anche unificato la gestione del tempo di visualizzazione/chiusura dell'applicazione, prima effettuato separatamente dall'applicazione e dal pannello delle preferenze. Adesso ci sono un po' di funzioni in `commonUtils.h` che gestiscono i tempi:

```
// etichette corrispondenti delle tacche dello slider
static int    dspValues[ 12 ] = { 0, 1, 2, 5, 10, 20, 30, 1, 2, 6, 12, 24 };
// effettivo valore (in minuti per chiarezza) del tempo da utilizzare
static    long    time2wait[12] = { 0, 1, 2, 5, 10, 20, 30, 60, 120, 300,
600, 1200 };

int
labelFromCode( int idx )
{
    return( dspValues[ idx] ) ;
}

long
secsFromCode( int idx )
{
    return( time2wait[ idx] * 60 ) ;
}

NSString *
unitFromCode( int idx )
{
    switch ( idx ) {
    case 1 :
        // unita', minuti
        return( [ NSString stringWithString: @"Minuto"] );
    case 2 :        case 3 :        case 4 :
    case 5 :        case 6 :
        // unita', minuti
        return( [ NSString stringWithString: @"Minuti"] );
    case 7 :    // unita', ore
        return( [ NSString stringWithString: @"Ora"] );
    case 8 :        case 9 :        case 10 :
```

```
    case 11 :
        // unita', ore
        return( [ NSString stringWithString: @"Ore"] );
    case 0 :    // giusto per precauzione
    default :
        return( [ NSString stringWithString: @"nessuno"] );
    }
}
```

Il metodo che gestisce lo slider all'interno del pannello delle preferenze è quindi:

```
- ( void )
adjustTimeSlider
{
    int         theTime = [ cycleTimeSlider intValue ] ;
    [ cycleTimeText setIntValue: labelFromCode( theTime) ];
    [ cycleTimeUnit setStringValue: unitFromCode( theTime) ];
}
```

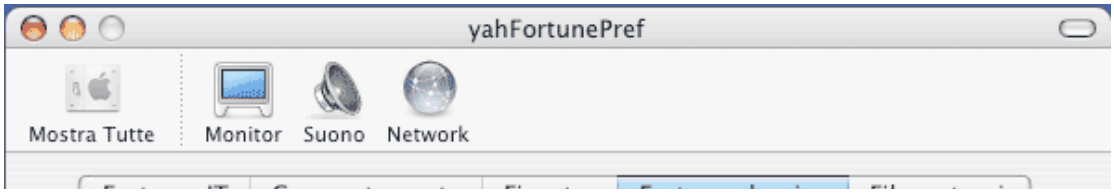
Il metodo che recupera il tempo di aggiornamento/chiusura dell'applicazione diventa:

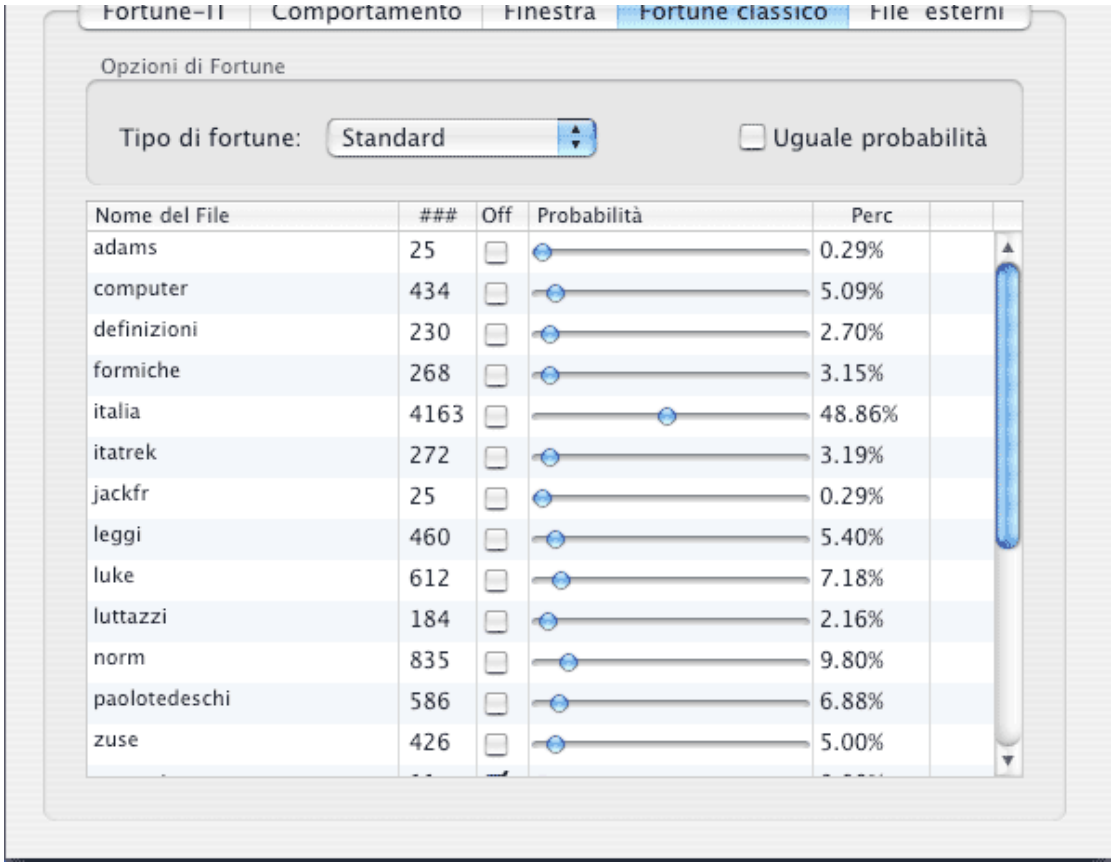
```
- ( long )
getSec
{
    int cycleTime = [[ currPrefs valueForKey: PFC_yahf_cycleTime ] intValue
] ;
    return ( secsFromCode( cycleTime) );
}
```

GESTIONE DEI FILE

La parte più consistente delle modifiche all'applicazione riguarda la gestione dei file. Avevo già notato come il comando fortune si basi su una serie di file di testo (e file di indici accessori) presenti all'interno di una serie di directory predefinite. Attualmente il comando è compilato in modo che cerchi questi file all'interno delle cartelle `fortunes` e `fortunes/off` presenti nella directory in cui il comando stesso è lanciato (ricordo che questa cartella è interna al bundle dell'applicazione YahFortune). Ogni file presente, per essere considerato come sorgente di frasi, deve avere come compagno un altro file, avente lo stesso nome ma suffisso `dat`. Per produrre questi file compagni (che sono una specie di tabella per facilitare l'accesso alle singole frasi) si deve utilizzare un altro comando, `strfile`.

Quando si lancia il comando, fortune esamina i file contenuti all'interno delle cartelle indicate, alla ricerca dei file che hanno suffisso dat. Attraverso questi file riesce a calcolare il numero complessivo di frasi presenti; tra tutte, ne sceglie una caso. Il risultato è che ogni frase ha la medesima probabilità di essere selezionata; di conseguenza, la probabilità che sia selezionata una frase all'interno di un dato file è proporzionale al numero di frasi contenute all'interno del file medesimo.





Tutto questo discorso mi serve per arrivare a spiegare la vista dell'elenco dei file considerati dal comando nella generazione dei fortune, secondo quella che io chiamo la modalità classica (quella appunto che utilizza le directory predefinite). In questa vista presento l'elenco dei file utilizzati, comprensivi di una serie di informazioni: numero delle frasi presenti, probabilità che il fortune scelto sia in loro contenuto, se il file è considerato pieno di frasi offensive o meno.

Non è possibile modificare la lista (si tratta appunto di file predefiniti, che si trovano, ribadisco ancora una volta, all'interno del bundle dell'applicazione). Però, utilizzando due opzioni di fortune, è possibile cambiare il modo in cui il fortune è generato. Attraverso l'opzione di equiprobabilità, ogni file è considerato equiprobabile, indipendentemente dal numero di frasi in esso contenuto. Con l'opzione di scelta se utilizzare o meno i fortune considerati offessivi, invece, si selezionano l'uno o l'altro o entrambi i gruppi di file.

Ma prima di poter fare qualsiasi cosa, occorre costruire un modello dei dati.

Il modello dei fortune

Per sapere quanti e quali file di fortune sono interessati dall'applicazione, ho definito due nuove classi. La classe `FortuneFile` modella un singolo file di frasi adatte ad essere utilizzate con fortune. La classe `FortFileCollection` invece gestisce una collezione di file di fortune.

La classe `FortuneFile` mi serve a contenere le informazioni specifiche di ogni file di fortune:

```
@interface FortuneFile : NSObject
{
    // percorso del file
    NSString    * fortuneFilePath ;
    // probabilita' associata
    float        fileProb ;
    // numero di messaggi presenti
    int            numOfFortunes ;
    // lunghezza massima e minima delle stringhe
    int            minLenght ;
    int            maxLenght ;
    // se e' considerato offensivo o meno
    BOOL        isOffensive ;
    // se va utilizzato nella generazione o meno
    BOOL        isUsed ;
}
```

Il problema fondamentale di questa classe è come recuperare i dati. In effetti, tutto quello che posso sperare è di poter chiamare il seguente metodo, fornendogli appunto il path completo del file che si suppone contenere frasi di fortune:

```
- (id)        initWithPath: (NSString *) thePath ;
```

Con approccio pragmatico, decido che il file è considerato di fortune se, sottoposto alle operazioni del comando `strfile`, produce un risultato sensato, ovvero un file dat accettabile. In altre parole, si tratta di eseguire il comando

```
strfile thePath
```

e vedere cosa salta fuori; in effetti, come era già stato visto qualche capitolo fa, il comando produce sul terminale una serie di righe come le seguenti:

```
djzeropb:~/temp_fortune/fortunes djzero00$ ../strfile adams.txt
"adams.txt.dat" created
There were 115 strings
Longest string: 569 bytes
Shortest string: 4 bytes
```

Metto tutto assieme e produco il metodo:

```
- (id)
initWithPath: (NSString *) thePath
{
    self = [ super init ] ;
    if ( self )
    {
        NSPipe            * pipe = [ NSPipe pipe ] ;
        NSFileHandle    * handle = [ pipe fileHandleForReading ] ;
        NSString        * fortunePath ;
        NSTask            * fortuneTask ;
         NSMutableArray    * argList = [ NSMutableArray arrayWithCapacity: 1
];
        NSData            * data;
        NSString        * theNewFortune = [ NSString string ];
        long            xxx ;
             NSBundle            *  mb  =  [  NSBundle  bundleWithIdentifier:
@"yahFortunePref" ] ;
```

```
                    //  recupero  il  percorso  dell'eseguibile  all'interno
dell'applicazione
        fortunePath = [ mb pathForResource: @"strfile" ofType: @"" ] ;
        // costruisco un task apposito
        fortuneTask = [[NSTask alloc] init];
        // predispongo i parametri di lancio del task
        [ fortuneTask setLaunchPath: fortunePath ];
        // la directory di lavoro e' quella di destinazione
                        [   fortuneTask   setCurrentDirectoryPath:   [   thePath
stringByDeletingLastPathComponent] ];
        [ argList addObject: thePath ];
        [ fortuneTask setArguments: argList ];
        // dico che l'uscita del comando va su una pipe
        // ridirigo sia lo standard output che lo standard error
        [ fortuneTask setStandardOutput: pipe ] ;
        [ fortuneTask setStandardError: pipe ];
        // lancio il programma
        [ fortuneTask launch];
        // gli do un attimo per partire
        Delay ( 1, & xxx );
        // aspetto, in maniera sincrona, che il programma risponda con tutta
la stringa
        // prodotta secondo i suoi criteri
        while ((data = [handle availableData]) && [data length])
        {
            NSString    * tmpData ;
            // prelevo i caratteri arrivati
                            tmpData  =  [[NSString  alloc]  initWithData:data
encoding:NSUTF8StringEncoding] ;
            // e li aggiungo a quelli gia' arrivati
            theNewFortune = [ theNewFortune stringByAppendingString: tmpData
];
        }
```

Fino a qui, son tutti concetti già noti: costruisco task, pipe, eseguo il comando, leggo il risultato. Adesso mi trovo con la stringa `theNewfortune` che contiene tutto ciò che l'esecuzione del comando avrebbe scritto sul terminale. Si tratta di interpretarlo estraendone in po' di informazioni; risolvo il problema passando la palla ad un metodo specifico, che chiamo `scanStringForValues`:

```
        // da qui devo estrarre le varie informazioni
        if ( [ self scanStringForValues: theNewFortune ] )
        {
            // aggiusto le altre variabili
            NSString    * fn = [ thePath lastPathComponent ];
            [ self setFortuneFilePath: fn ] ;
            // discrimino se e' offensivo dalla terminazione
            if ( [ fn hasSuffix: @"-o" ] )
                isOffensive = YES ;
            else
                isOffensive = NO ;
            // e' inizialmente sicuramente utilizzato
            isUsed = YES ;
        }
        else
        {
            // non ho trovato informazioni sensate
            return ( nil );
        }
    }
```

```
    return ( self );
}
```

Poi completo il tutto riempiendo le variabili d'istanza che è facile desumere: il nome del file, se è un file offensivo o meno in base alla presenza del suffisso `-o` al nome del file, e poi dico che per il momento il file sarà utilizzato da fortune per la ricerca di una nuova frase.

SCANNERS!!!

Ho solo spostato il problema: adesso devo scrivere un metodo `scanStringForValues` che riesca ad estrarre dalla stringa prodotta le informazioni che sono lì raccolte. Pensavo di faticare parecchio (dopotutto, si tratta di guardare i vari caratteri della stringa, e pescare i numeri che ci si aspetta dove si pensa di trovarli), ma ho trovato una affascinante classe di Cocoa che si preoccupa di fare il tutto al posto mio: `NSScanner`.

Questa classe rappresenta una specie di topo di biblioteca a cui si da in pasto una stringa; poi, tramite opportuni comandi, gli si dice di andare avanti fino a trovare una determinata combinazione di caratteri, e magari di metterla da qualche parte, o vedere se per caso c'è un numero, cose del genere. Molto bello. Il codice allora del metodo diventa piuttosto semplice e noioso:

```
- (BOOL)
scanStringForValues: (NSString *) theString
{
    // questa e' la stringa che strfile produce:
    /*    "path/to/file.dat" created
        There were <numofstrings> strings
        Longest string: <maxlenght> bytes
        Shortest string: <minlenght> bytes
    */
    // utilizzo quest'utile classe NSScanner
    NSScanner    * theScanner;
    int            numofstrings, maxlenght, minlenght ;
    BOOL        tmpVal ;
    NSString    * scanned ;
        // inizializzo lo scanner
    theScanner = [NSScanner scannerWithString: theString ];
    // leggo fino a trovare created
    tmpVal = [theScanner scanUpToString: @"created" intoString: & scanned] ;
    if ( tmpVal == NO ) return ( NO );
    tmpVal = [theScanner scanString: @"created" intoString: & scanned] ;
    if ( tmpVal == NO ) return ( NO );
    tmpVal = [theScanner scanString: @"There were" intoString: & scanned] ;
    if ( tmpVal == NO ) return ( NO );
    // adesso dovrebbe esserci il numero delel stringhe presenti
    tmpVal = [theScanner scanInt:& numofstrings ] ;
    if ( tmpVal == NO ) return ( NO );
    // se non riconosco almeno una stringa, lascio perdere
    if ( numofstrings <= 0 ) return ( NO );
    tmpVal = [theScanner scanString: @"strings" intoString: & scanned] ;
    if ( tmpVal == NO ) return ( NO );
       tmpVal = [theScanner scanString: @"Longest string:" intoString: &
scanned] ;
    if ( tmpVal == NO ) return ( NO );
    // la lunghezza della stringa piu' lunga
    tmpVal = [theScanner scanInt:& maxlenght ] ;
```

```
    if ( tmpVal == NO ) return ( NO );
    tmpVal = [theScanner scanString: @"bytes" intoString: & scanned] ;
    if ( tmpVal == NO ) return ( NO );
      tmpVal = [theScanner scanString: @"Shortest string:" intoString: &
scanned ] ;
    if ( tmpVal == NO ) return ( NO );
    // la lunghezza della stringa piu' corta
    tmpVal = [theScanner scanInt:& minlenght ] ;
    if ( tmpVal == NO ) return ( NO );
    numOfFortunes = numofstrings ;
    minLenght = maxlenght ;
    maxLenght = minlenght ;
    return ( YES ) ;
}
```

La comprensione è facile, una volta noto che `scanUpToString:` mangia caratteri fino a trovare la stringa che gli si passa come argomento, mentre `scanString:` cerca la combinazione di caratteri specificata dal suo argomento. Così, le prime tre istruzioni riguardanti `theScanner` cercano prima la parola `created` (e ripuliscono la stringa fino a quella parola) e poi le stringhe `created` e `there were`, eliminandole. A questo punto potrebbe esserci un numero intero; e quindi uso `scanInt`. Ogni operazione su `NSScanner` produce un risultato booleano: se questo è NO, significa che la scansione non ha avuto successo, e quindi il prodotto non è quello che ci si aspetta che sia; piuttosto che chiedermi cosa potrebbe essere andato storto, dico che è inutile continuare e dico che il file non è un buon file di fortune.

Il metodo continua mangiando man mano pezzi della stringa, e conservando i valori che sono utili; alla fine, li assegna tutti alle variabili d'istanza, e considera il file esaminato come un buon file di fortune.

È però una bella noia dover ogni volta eseguire il comando strfile per sapere quanti fortune si trovano all'interno di un file. Per questo ho studiato un meccanismo (che sarà esplicitato all'interno della classe `FortFileCollection`) per mantenere le informazioni su di un file XML. Mi occorrono quindi due nuovi metodi per costruire un `FortuneFile` e salvarlo a partire da un dizionario. Nulla di più facile:

```
- (id)
initWithDict: (NSDictionary *) theDict
{
    if ( self = [ super init ] )
    {
        // assegno i vari dati a partire dagli elementi del dizionario
        [ self setFortuneFilePath: [ theDict objectForKey: @"FilePath" ]];
        fileProb = [ [ theDict objectForKey: @"fileProb" ] floatValue ];
            numOfFortunes = [ [ theDict objectForKey: @"numOfFortunes" ]
intValue ];
        minLenght = [ [ theDict objectForKey: @"minLenght" ] intValue ];
        minLenght = [ [ theDict objectForKey: @"minLenght" ] intValue ];
           isOffensive = [ @"YES" isEqualToString: [ theDict objectForKey:
@"isOffensive" ] ];
        isUsed = [ @"YES" isEqualToString: [ theDict objectForKey: @"isUsed"
] ];
    }
    return ( self );
}

- (NSMutableDictionary *)
```

```
getXMLDescription
{
    // un dizionario per contenere le varie proprieta' XML
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];
    // nome della classe
    [dict setObject: NSStringFromClass([self class])
            forKey: @"className" ];
    // tutte le altre variabili d'istanza fondamentali
    [dict setObject: fortuneFilePath forKey: @"FilePath" ];
    [dict setObject: [NSString stringWithFormat:@"%f", fileProb]
            forKey: @"fileProb" ];
    [dict setObject:[NSString stringWithFormat:@"%d", numOfFortunes]
            forKey: @"numOfFortunes" ];
    [dict setObject:[NSString stringWithFormat:@"%d", minLenght]
            forKey: @"minLenght" ];
    [dict setObject:[NSString stringWithFormat:@"%d", maxLenght]
            forKey: @"maxLenght" ];
    [dict setObject:(isOffensive ? @"YES" : @"NO")
            forKey: @"isOffensive" ];
    [dict setObject:(isUsed ? @"YES" : @"NO")
            forKey: @"isUsed" ];
    return dict;
}
```

UNA COLLEZIONE DI FILE

Una collezione di file fortune è, agli scopi della mia applicazione, l'elenco di tutti i file fortune validi all'interno di una serie di directory. Ragione per cui la classe `FortFilesCollection` è presto fatta:

```
@interface FortFilesCollection : NSObject
{
    NSMutableArray    * theFortList ;
}
```

C'è il problema di inizializzarla; verosimilmente, avrò un metodo cui passo l'elenco delle directory da esaminare. Dal metodo ho tolto alcune istruzioni che spiegherò più avanti.

```
- (id )
initWithDirList: (NSArray *) thePaths
{
    if ( self = [ super init ] )
    {
        NSFileManager     *fileManager = [NSFileManager defaultManager];
        NSEnumerator * filelist = [ thePaths objectEnumerator ];
        NSString     * aPath ;
        // la collezione e' inizialmente vuota
        [ self setTheFortList: [ NSMutableArray arrayWithCapacity: 0 ] ];
        // spazzolo tutti i percorsi
        while ( aPath = [filelist nextObject])
        {
            BOOL            isAdir, fileOK, retVal ;
            NSMutableArray    * theArr ;
            ...
            if ( ... )
            {
                ...
            }
```

```
            else
            {
                 // non c'e', oppure non va bene, parto dalla directory e la
esploro
                 // controllo se esiste veramente la cartella, che non si sa
mai
                  fileOK = [ fileManager fileExistsAtPath: aPath isDirectory:
& isAdir ];
                if ( fileOK && isAdir)
                {
                    // esiste ed e' proprio una cartella
                    // produco i dat file e le informazioni necessarie
                    theArr = [ self produceFileAndLoadData: aPath ];
                    // aggiungo la lista dei file alla collezione
                    [ theFortList addObjectsFromArray: theArr ] ;
                    ...
                }
            }
        }
    }
    return ( self );
}
```

Essenzialmente, c'è un ciclo che esamina tutte le directory passate come argomento (dopo averne verificato l'esistenza, che non si sa mai), e per ciascuna di esse esegue il metodo `produceFileandLoadData:`; il risultato di questo metodo è un array con i file contenuti nella directory che sono stati considerati validi. Questo array è aggiunto alla lista corrente.

Per ogni directory quindi si esegue:

```
- ( NSMutableArray * )
produceFileAndLoadData: (NSString*) fullPath
{
    int                 i ;
    // elenco dei file presenti
    NSFileManager     *fileManager = [NSFileManager defaultManager];
    NSArray            * dirContent = [ fileManager directoryContentsAtPath:
fullPath ];
    // costruisco un nuovo vettore per i FortuneFile
    int                 numFile = [ dirContent count ];
    NSMutableArray    * theFileList = [ NSMutableArray arrayWithCapacity: 1
] ;
    // adesso, per ogni elemento, costruisco l'elemento e poi lo aggiungo
    for ( i = 0; i < numFile; i++)
    {
        BOOL        isAdir ;
        FortuneFile    * ff ;
        // nome del file
        NSString     * myfile = [ dirContent objectAtIndex: i ] ;
        // percorso completo
                      NSString           *   fileFullPath   =   [   fullPath
stringByAppendingPathComponent: myfile] ;
        // se il file non va bene, passo oltre
         if ( ! [ fileManager fileExistsAtPath: fileFullPath isDirectory: &
isAdir ] )
            continue ;
        // se la sua estensione e' .dat, passo oltre
        if ( [ [ fileFullPath pathExtension ] isEqualToString: @"dat" ] )
            continue ;
        // se il nome comincia con un punto, passo oltre
```

```
        if ( [ myfile hasPrefix: @"."] )
            continue ;
        // se e' una directory, passo oltre
        if ( isAdir )
            continue ;
        // se arrivo qui, ho un vero e proprio file
        // costruisco un fortuneFile con i dati che ne ricavo
        ff = [[ FortuneFile alloc ] initWithPath: fileFullPath ];
        // se ci sono riuscito, lo aggiungo alla collezione
        if ( ff )
            [ theFileList addObject: ff ];
    }
    // restituisco l'elenco dei FortuneFile
    return ( theFileList );
}
```

Qui si ricava l'elenco dei file presenti nella directory, e si esaminano uno ad uno alla ricerca di candidati validi. Si saltano senz'altro una serie di file che si è sicuri non andare bene (quelli che hanno estensione dat, quelli che cominciano con un punto, le directory...). Per i file rimanenti, si prova ad eseguire il metodo `initWithPath`. Se il risultato ha senso, abbiamo un file e lo aggiungiamo alla lista.

A questo punto, aggiungendo alla classe `yahFortunePref` la variabile d'istanza

```
    FortFilesCollection            * stdFortFiles ;
```

alla quale è associato un valore nelle preferenze

```
                            [       np       setObject:       [       NSArray
arrayWithObjects:    @"/Contents/Resources/fortunes",
                                              @"/Contents/Resources/fortunes/
                                              nil ]
                forKey: PFC_yahf_stdFortFiles ] ;
```

composto dalle due directory con le quali è stato configurato il comando di fortune, all'apertura del pannello delle preferenze posso recuperare l'intera lista dei file di fortune in questo modo:

```
                tmpColl  =  [[  FortFilesCollection  alloc]  initWithPath:
yahFortuneAppPath
                                    andDirList: [ currPrefs objectForKey:
PFC_yahf_stdFortFiles ] ] ;
        [ self setStdFortFiles: tmpColl ];
```

dove per mia comodità ho costruito un altro metodo di inizializzazione per `FortFileCollection` ed utilizzato la variabile `yahFortuneAppPath` che contiene il percorso dell'applicazione YahFortune. Il nuovo metodo altro non fa che chiamare il metodo esposto in precedenza dopo aver adattato opportunamente gli argomenti.

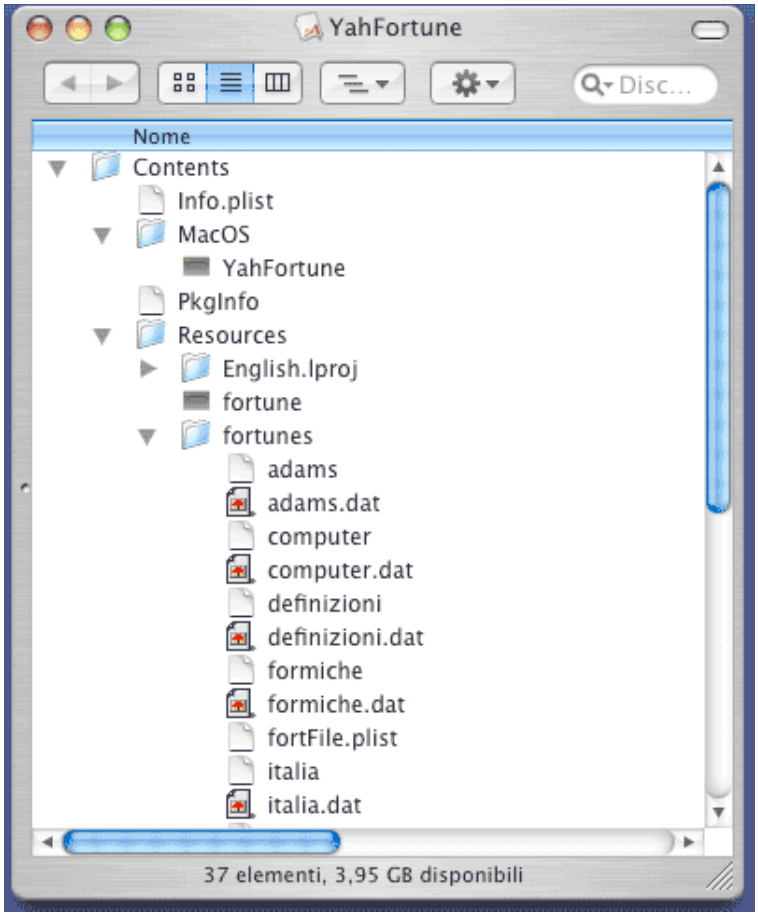
```
- (id )
initWithPath: (NSString*) appPath andDirList: (NSArray *) thePaths
{
    NSMutableArray    * newList = [ NSMutableArray array ] ;
    NSEnumerator    * nl = [ thePaths objectEnumerator ];
    NSString        * aPath ;
    // costruisco un vettore con i path completi
```

```
    while ( aPath = [nl nextObject])
    {
        NSString * ns = [ NSString stringWithFormat: @"%@%@", appPath, aPath
];
        [ newList addObject: ns ];
    }
    // e faccio partire l'inizializzatore designato
    return ( [ self initWithDirList: newList ] );
}
```

Ora, è piuttosto seccante ogni volta riscostruire da zero l'elenco dei file fortune. Ho pensato quindi di salvare la lista e di conservarla da qualche parte

SALVO LA LISTA

L'idea è che in ogni directory che contiene file di fortune sia presente anche un file, dal nome `fortFile.plist`, che mantenga l'elenco dei file di fortune presenti, completi delle informazioni relative.



Quindi, il metodo `initWithDirList:`, prima di lanciarsi nella costruzione della lista, verifica se non sia già presente il file (ecco le istruzioni mancanti prima sostituite con puntini):

```
- (id )
initWithDirList: (NSArray *) thePaths
{
        ...
```

```
        while ( aPath = [filelist nextObject])
        {
            BOOL            isAdir, fileOK, retVal ;
            NSMutableArray    * theArr ;
                 // all'interno di ogni cartella dovrebbe esserci il file
fortFile.plist
            // che conserva le info relative ai file
             NSString        * indexFile = [ aPath stringByAppendingString:
@"/fortFile.plist" ];
            // vedo se esiste il file indicato
              if ( [ fileManager fileExistsAtPath: indexFile isDirectory: &
isAdir ] )
            {
                // il file esiste, carico le info a partire dal file
                theArr = [ self loadXMLDataFromFile: indexFile ];
                // controllo che non ci siano state variazioni
                 retVal = [ self checkForModifiedFile: theArr inDir: aPath ]
;
                // aggiungo la lista dei file alla collezione
                [ theFortList addObjectsFromArray: theArr ] ;
                          // se ci sono aggiornamenti sui file, riscrivo
fortFile.plist
                if ( retVal )
                    [ self writeFortList: theArr toFile: indexFile ];
            }
            else
            {
            ...
```

Se il file è pre-esistente, preleva le informazioni proprio da questo file con il metodo apposito:

```
- ( NSMutableArray * )
loadXMLDataFromFile: (NSString*) fullPath
{
    int                i ;
    // estraggo i dati dal file
     NSArray             * arrFromFile = [ NSArray arrayWithContentsOfFile:
fullPath ] ;
    // costruisco un nuovo vettore per i FortuneFile
    int                numFile = [ arrFromFile count ];
      NSMutableArray     * fortArray = [ NSMutableArray arrayWithCapacity:
numFile ];
    // esamino i record uno alla volta
    for ( i = 0; i < numFile; i++)
    {
        FortuneFile    * ff ;
        // costruisco un Fortunefile con i dati presenti nel dizionario
              ff  =  [[  FortuneFile  alloc  ]  initWithDict:  [  arrFromFile
objectAtIndex: i ] ];
        // se ci sono riuscito, lo aggiungo alla collezione
        if ( ff )
            [ fortArray addObject: ff ];
    }
    // restituisco l'elenco dei FortuneFile
    return ( fortArray );
}
```

Trattandosi di un file XML, le cose sono piuttosto semplici (o meglio, sono rese semplici da Cocoa): nel file è contenuta una rappresentazione XML di un `NSArray`. Ogni elemento di questo array è un `NSDictionary`, che posso utilizzare per costruire

un oggetto della classe `FortuneFile`. Alla fine, ho costruito una lista di file di fortune senza dover passare per il comando `strfile`.

Però qualche utente birichino portrebbe aver modificano qualcosa, per cui verifico che non ci siano incongruenze palesi utilizzando il metodo `checkForModifiedFile`:

```
- ( BOOL )
checkForModifiedFile: (NSMutableArray*) theArr inDir: (NSString *) aPath
{
    NSEnumerator    * filelist = [ theArr objectEnumerator ];
    FortuneFile        * theFile ;
    NSFileManager     * fm = [NSFileManager defaultManager];
    BOOL            isToBeWritten = NO ;
    // spazzolo tutti i file presenti
    while ( theFile = [ filelist nextObject ] )
    {
        BOOL            isAdir, fileOK ;
        NSString        * fullPath ;
        NSDictionary    * fattrs     ;
        NSDate            * fileModDate ;
        NSString        * datPath ;
        NSDate            * fileDatDate ;
        // path completo del file fortune
             fullPath  =  [  aPath  stringByAppendingPathComponent:  [  theFile
fortuneFilePath]];
        // ne pesco la sua ultima data di modifica
        fattrs     = [fm fileAttributesAtPath: fullPath traverseLink:NO];
        fileModDate = [fattrs fileModificationDate] ;
        // path completo del file .dat corrispondente
        datPath = [ fullPath stringByAppendingPathExtension: @"dat"];
        // verifico che questo file esista
        fileOK = [ fm fileExistsAtPath: datPath isDirectory: & isAdir ];
        if ( fileOK )
        {
            // il file esiste, ma potrebe essere troppo vecchio
            NSComparisonResult    xx ;
            // pesco la sua ultima data di modifica
                      fattrs         =  [  fm  fileAttributesAtPath:  datPath
traverseLink:NO];
            fileDatDate = [fattrs fileModificationDate] ;
            // la confronto con quella del suo file
            xx = [ fileModDate compare: fileDatDate] ;
            // si suppone che il file .dat sia piu' giovane
            // se non lo e', non va bene...
            if ( xx == NSOrderedDescending )
                fileOK = NO ;
        }
        if ( ! fileOK )
        {
            // qui dentro entro se il file .dat non esiste
            // oppure e' troppo vecchio
            FortuneFile    * ff ;
            // costruisco un nuovo oggetto FortuneFile
            // (cosa che produce un .dat)
            ff = [[ FortuneFile alloc ] initWithPath: fullPath ];
            // ed assegno tutti i valori ottenuti al vecchio oggetto
            [ theFile setFileProb: [ ff fileProb ]];
            [ theFile setNumOfFortunes: [ ff numOfFortunes ]];
            [ theFile setIsOffensive: [ ff isOffensive ]];
            [ theFile setIsUsed: [ ff isUsed ]];
            [ theFile setMinLenght: [ ff minLenght ]];
            [ theFile setMaxLenght: [ ff maxLenght ]];
```

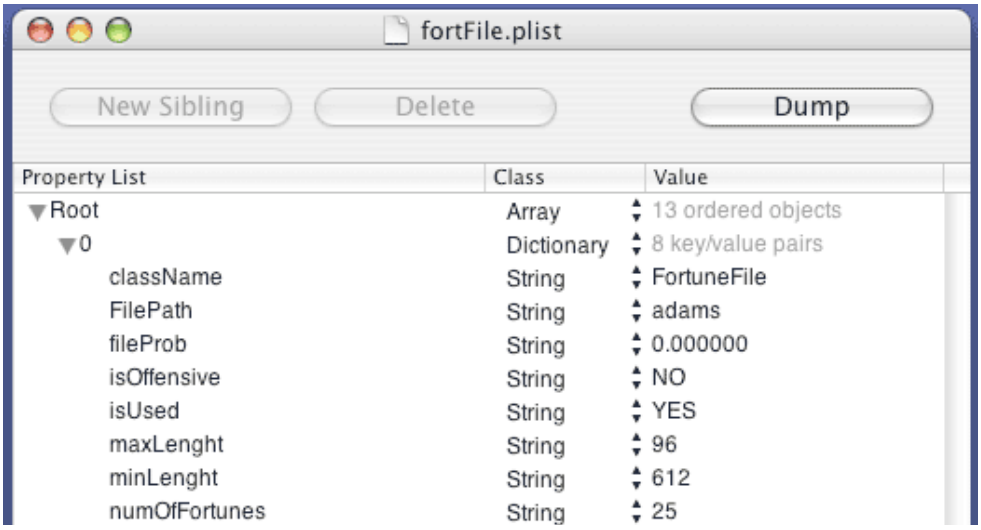
```
            // dico che occorre riscrivere il file delle info
            isToBeWritten = YES ;
        }
    }
    return ( isToBeWritten ) ;
}
```

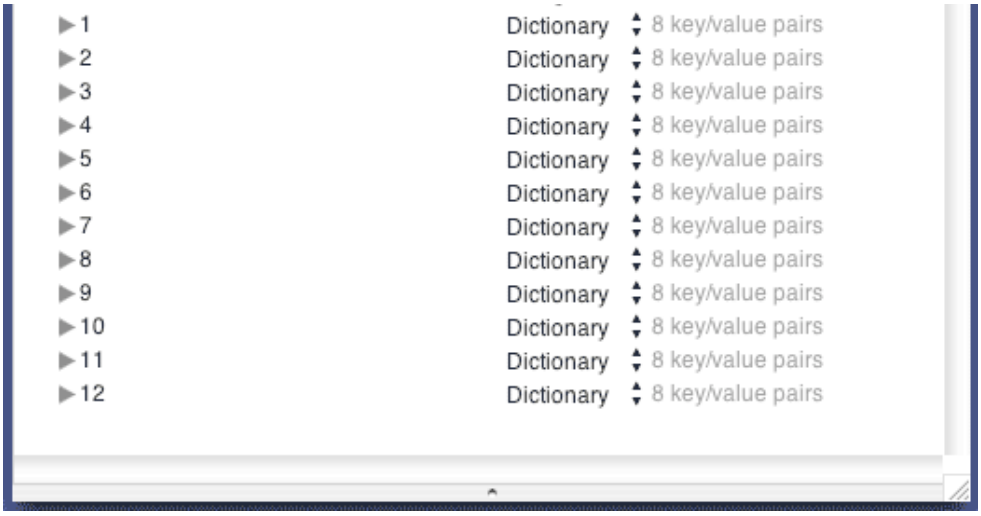
Per ogni file della collezione, verifico che esista il corrisponde file indice con estensione dat, e che la data di questo file sia posteriore a quella del file stesso (se fosse anteriore, potrebbe esserci stata una modifica sul file...). Nel caso infelice in cui queste condizioni non siano soddisfatte, per stare sul sicuro ricostruisco le informazioni (ed il file dat) chiamando il metodo `initWithPath` sull'oggetto `FortuneFile`.

Ma non basta; per evitare che al lancio successivo debba rifare l'operazione, è bene che aggiorni il file `fortFile.plist`, cosa invero piuttosto semplice. Ho scritto il metodo `writeFortList: toFile:`

```
- ( void )
writeFortList: (NSMutableArray *) theArr toFile: (NSString *) aPath
{
    int    i ;
    int    numElem = [ theArr count ] ;
     NSMutableArray * dictList = [NSMutableArray arrayWithCapacity: numElem
];
for (i = 0; i < numElem; i++)
    {
        // ogni elemento dello array e' un dictionary che raccoglie
        // la descrizione dell'elemento in XML
        NSMutableDictionary * elemDict ;
        // descrizione dell'elemento in XML
        elemDict = [[theArr objectAtIndex:i] getXMLDescription] ;
        // aggiungo il dizionario allo array
[dictList addObject: elemDict ];
}
    // scrittura del vettore sul file
    [ dictList writeToFile: aPath atomically: YES ];
}
```

Per ogni elemento della lista dei file, costruisco un dizionario con la descrizione delle variabili d'istanza, li infilo all'interno di un array, e scrivo il tutto sul file apposito, sfruttando ancora una volta la potenza di Cocoa nella gestione di file XML.





Ora che tutte le informazioni relative ai file di fortune si trovano all'interno degli oggetti della classi `FortFilesCollection`, questa stessa classe costituisce il modello dei dati della tabella che li rappresenta: ne segue che la tabella che presenta la lista degli stessi conviene che utilizzi appunto `FortFileCollection` come classe sorgente dei dati. Questo fatto va dichiarato esplicitamente all'interno del metodo `mainViewDidLoad` del pannello delle preferenze con l'istruzione:

```
    [ stdFileList setDataSource: stdFortFiles ];
```

Il collegamento tra la `NSTableView` e la sua sorgente dei dati non può essere stabilito all'interno di *Interface Builder*, dal momento che lì non è disponibile la variabile `stdFortFiles`...

I due metodi necessari alla gestione della tabella sono molto semplici:

```
- (int)
numberOfRowsInTableView:    (NSTableView *)tableView
{
    return [ theFortList count ] ;
}

- (id)
tableView:             (NSTableView *)tableView
    objectValueForTableColumn:    (NSTableColumn *)tableColumn
    row:             (int)row
{
    NSString        * colId = [ tableColumn identifier ] ;
    FortuneFile        * ff = [ theFortList objectAtIndex: row ] ;
    if ( [ colId isEqual: @"filename" ] )
    {
        return ( [ [ ff fortuneFilePath ] lastPathComponent ] );
    }
    else if ( [ colId isEqual: @"percvalue" ] )
    {
        return ( [ NSNumber numberWithFloat: [ ff fileProb ] * 100 ] );
    }
    else if ( [ colId isEqual: @"numoffort" ] )
    {
        return ( [ NSNumber numberWithInt: [ ff numOfFortunes ]] );
    }
    else if ( [ colId isEqual: @"offensive" ] )
```

```
    {
        return ( [ NSNumber numberWithInt:( [ ff isOffensive ] ? NSOnState:
NSOffState) ] );
    }
    else if ( [ colId isEqual: @"used" ] )
    {
        return ( [ NSNumber numberWithInt:( [ ff isUsed ] ? NSOnState:
NSOffState) ] );
    }

    return ( [ ff fortuneFilePath ] );
}
```

CALCOLO DELLE PROBABILITÀ

Il motivo per cui mi serve la lista dei file ed il numero delle frasi contenute è per calcolare le probabilità di utilizzo di ogni singolo file: è molto semplice. All'apertura della vista (`mainViewDidLoad`) eseguo la seguente istruzione:

```
            [ stdFortFiles adjustProbValues: [ offensiveUse selectedTag ]
updProbZero: YES ];
```

che calcola le probabilità di ogni singolo file. Bisogna tenere conto di quale tipo di file sono utilizzati: `offensiveUse` è un outlet al menu pop-up che mi consente tre possibilità: usare solo i file normali (PFC_OU_NORMAL); usare solo i file offensivi (PFC_OU_ONLYOFFENSIVE); usare tutti i file (`PFC_OU_ADDOFFENSIVE`). Per semplificarmi la vita, utilizzo la seguente funzione:

```
BOOL
useMe ( BOOL std, BOOL off, BOOL isOff )
{
    // un file partecipa se si accettano standard e offensivi
    // oppure se si accettano i soli standard ed il file non e' offensivo
    // oppure se si accettano i soli offensivi ed il file e' offensivo
    return ( ( std && off) || ( std && ! off && ! isOff ) || ( ! std && off
&& isOff ) ) ;
}
```

Il metodo presenta anche il parametro `updProb`; se questo è vero, forza un ricalcolo completo delle probabilità, piuttosto che fare affidamento sui valori contenuti all'interno dei singoli oggetti `FortuneFile` (ma questa è un'opzione non ancora attivata...).

```
- ( BOOL )
adjustProbValues: (int) useOff updProbZero: (BOOL) updProb
{
    NSEnumerator * fl = [ theFortList objectEnumerator ];
    FortuneFile     * aFF ;
    long        totNumOfFort = 0 ;
    float        sumFFileProb = 0 ;
    float        eps ;
    // vedo se devo includere i file standard e quelli offensivi
    BOOL        incStd = (useOff != PFC_OU_ONLYOFFENSIVE ) ;
    BOOL        incOff = (useOff != PFC_OU_NORMAL ) ;
    // conto il numero di fortune e sommo le varie probabilita'
    while ( aFF = [ fl nextObject ] )
    {
```

```
        // un file partecipa alla storia solo se e' considerato utilizzato
        // e se risponde alla caratteristiche desiderate
        if ( [ aFF isUsed ] && useMe( incStd, incOff, [ aFF isOffensive ]) )
        {
            totNumOfFort += [ aFF numOfFortunes ] ;
            sumFFileProb += [ aFF fileProb ] ;
        }
    }
    // se non ci sono fortune, non c'e' proprio nulla da aggiornare
    if ( totNumOfFort <= 0 )
        return ( NO ) ;
    // se arrivo qui, totNumOfFort e' sicuramente positivo
    // aggiorno le probabilita' da zero, nei seguenti casi:
    // - me lo dicono con il parametro
    // - le probabilita' assegnate sono veramente basse
    if ( updProb || sumFFileProb <= 1E-6 )
    {
        // ricalcolo tutto in base al numero di fortune nei file
        float    baseProb = 1.0F / totNumOfFort ;
        fl = [ theFortList objectEnumerator ];
        // rispazzolo tutti i file
        while ( aFF = [ fl nextObject ] )
            // e se il file fa parte della festa, aggiorno la probabilita'
            if ( [ aFF isUsed ] && useMe( incStd, incOff, [ aFF isOffensive
]) )
                [ aFF setFileProb: baseProb * [ aFF numOfFortunes ] ] ;
            else    // altrimenti, ha probabilita' nulla
                [ aFF setFileProb: 0.0F ] ;
        // ho finito e tutto va bene
        return ( YES ) ;
    }
    // se arrivo qui, devo rimodulare le probabilita'
    // sono anche sicuro che sumFFileProb e' significativo
    // non lo faccio se la loro somma e' abbastanza vicina ad 1
    eps = sumFFileProb - 1 ;
    if ( eps < 0 ) eps = - eps ;
    if ( eps > 1E-6 )
    {
        // rimudulo le probabilita' in maniera proporzionale
        float    baseProb = 1.0F / sumFFileProb ;
        fl = [ theFortList objectEnumerator ];
        // rispazzolo tutti i file
        while ( aFF = [ fl nextObject ] )
            // e se il file fa parte della festa, aggiorno la probabilita'
            if ( [ aFF isUsed ] && useMe( incStd, incOff, [ aFF isOffensive
]) )
                [ aFF setFileProb: baseProb * [ aFF fileProb ] ] ;
            else    // altrimenti, ha probabilita' nulla
                [ aFF setFileProb: 0.0F ] ;
        // ho finito e tutto va bene
        return ( YES ) ;
    }
    return ( YES ) ;
}
```

Le probabilità sono proporzionali al numero di frasi (`numOfFortunes`) contenute nel file. Calcolando il numero complessivo di fortune, e riscalando tutto utilizzando questo valore, si ottiene per ogni file un numero compreso tra Zero ed Uno che appunto rispecchia la probabilità che il fortune scelta appartenga al file.

Qualora invece l'utente decidessi di attribuire ad ogni file identica probabilità, occorre ricalcolare il tutto in altra maniera: c'è un metodo apposta.

```
- ( BOOL )
setStdEqualProb: (int) useOff
{
    NSEnumerator * fl = [ theFortList objectEnumerator ];
    FortuneFile     * aFF ;
    long        totNumOfFort = 0 ;
    float        baseProb = 0 ;
    BOOL        incStd = (useOff != PFC_OU_ONLYOFFENSIVE ) ;
    BOOL        incOff = (useOff != PFC_OU_NORMAL ) ;
    // conto quanti file partecipano alla festa
    while ( aFF = [ fl nextObject ] )
        if ( [ aFF isUsed ] && useMe( incStd, incOff, [ aFF isOffensive ]) )
            totNumOfFort += 1 ;
    // se non ci sono file, non c'e' proprio nulla da aggiornare
    if ( totNumOfFort <= 0 )
        return ( NO ) ;
    // se arrivo qui, totNumOfFort e' sicuramente positivo
    // la probabilita' e' la stessa per tutti
    baseProb = 1.0F / totNumOfFort ;
    fl = [ theFortList objectEnumerator ];
    // rispazzolo i file
    while ( aFF = [ fl nextObject ] )
        // se partecipano alla festa, hanno una probabilita'
        if ( [ aFF isUsed ] && useMe( incStd, incOff, [ aFF isOffensive ]) )
            [ aFF setFileProb: baseProb ] ;
        else    // altrimenti, probabilita' nulla
            [ aFF setFileProb: 0.0F ] ;
    return ( YES ) ;
}
```

Questa volta la probabilità dipende solo dal numero di file che partecipano alla scelta del fortune. La probabilità attribuita ad ogni file è proprio l'inverso del numero dei file.

Per finire, manca il metodo di `yahFortunePref` che scatena il tutto ad ogni modifica: è l'action associata ad una scelta del menu pop-up e al pulsante di spunta sulle probabilità:

```
- (IBAction)
updateStdProb: (id) sender
{
    if ( [ equalProbBtn state ] == NSOnState )
        [ stdFortFiles setStdEqualProb: [ offensiveUse selectedTag ] ];
    else
            [ stdFortFiles adjustProbValues: [ offensiveUse selectedTag ]
updProbZero: YES ];
    [ stdFileList reloadData ];
    [ self updatePrefFile: sender ] ;
}
```

ALLA RICERCA DELL'APPLICAZIONE PERDUTA

Mi sono accorto che i semplici meccanismi di lancio dell'applicazione non sempre funzionavano, per cui ho battezzato una variabile d'istanza di `yahFortunePref` che contenga il percorso completo dell'applicazione, ed un metodo per inserirci un buon valore. Il metodo è utilizzato ogni volta che è necessario conoscere il percorso dell'applicazione *YahFortune*. La prima volta che viene lanciato, il percorso non è noto, e quindi il metodo si affanna a cercare l'applicazione. Se invece è noto, il metodo verifica

comunque che l'applicazione sia nel percorso indicato, e si allarma se così non è.

```
- (NSString * )
getYahFortuneAppPath
{
    // prima provo col path
    if ( yahFortuneAppPath )
    {
        NSString    * appName ;
        NSString    * type ;
        BOOL        appExists ;
        // vedo se corrisponde effettivamente
            appExists = [[ NSWorkspace sharedWorkspace ] getInfoForFile:
yahFortuneAppPath
                application: & appName type: & type ] ;
        // se e' a posto, lo restituisco
        if ( appExists )
            return ( yahFortuneAppPath ) ;
    }
```

Verosimilmente, qui la variabile è già stata inizializzata da una invocazione precedente del metodo; tuttavia, si controlla se l'applicazione è proprio lì dove indicato (non sia mai che l'utente la sposti). Se va tutto bene, si restituisce il percorso; altrimenti, si continua, perché è come se il percorso non fosse noto.

```
     // se arrivo qui, o l'applicazione indicata non esiste, oppure il path
e' nullo
      // provo a vedere se e' stata spostata, ma registrata dal sistema
operativo
    NSString * fp = [[ NSWorkspace sharedWorkspace ] fullPathForApplication:
@"YahFortune" ];
    // c'e', ho finito
    if ( fp )
        return ( fp ) ;
```

A quanto pare, se l'applicazione è stata lanciata almeno una volta, oppure si trova nella directory predefinite (come la cartella *Applicazioni*), il sistema operativo ne tiene traccia, e il metodo `fullPathForApplication` della classe `NSWorkspace` è sufficiente a determinare il percorso. Se invece l'applicazione non è mai stata lanciata, non ho altra ppossibilità che chiedere all'utente di cercarla per me.

```
    // se arrivo qui, non c'e' verso di trovarla, chiedo di cercarla per me
    NSOpenPanel    * sPanel ;
    int            risposta ;
    // apro un pannello di avvertimento
    NSAlert *alert = [[NSAlert alloc] init];
          [alert  setMessageText:@"Non  riesco  a  trovare  l'applicazione
YahFortune"];
    [alert setInformativeText:@"Localizza per me l'applicazione"];
    [alert addButtonWithTitle:@"OK"];
    [alert setAlertStyle:NSWarningAlertStyle];
    [alert runModal] ;
    // e all'ok apro un pannello di selezione file
    sPanel = [ NSOpenPanel openPanel ] ;
    [ sPanel setTitle:@"Cerca applicazione YahFortune" ];
    [ sPanel setPrompt:@"Eccola" ];
    [ sPanel setCanChooseDirectories: NO ];
    [ sPanel setCanChooseFiles: YES ];
    risposta = [ sPanel runModalForDirectory: nil file: nil
```

```
                    types: [ NSArray arrayWithObject: @"app"] ] ;
    [alert release];
```

Per farlo, apro il solito pannello di apertura file, e permetto la selezione dei soli file che hanno estensione app (in pratica, le applicazioni). L'utente però potrebbe essere dispettoso e selezionare una applicazione che c'entra nulla con fortune. Devo allora verificare che il percorso corrisponda proprio all'applicazione YahFortune.

```
    if ( risposta == NSOKButton )
    {
        // se e' andato bene, prelevo il nome del file
        fp = [ sPanel filename ] ;
        // verifico che e' effettivamente lei
        // confronto il bundle identifier
              NSString       *  bi  =  [  [  NSBundle  bundleWithPath:  fp  ]
bundleIdentifier ] ;
        if ( [ bi isEqualToString: PFC_yahf_prefFileName ] )
            return ( fp );
    }
    // se arrivo qui, proprio non l'ho trovata...
    return ( nil );
}
```

Piuttosto che controllare il nome (un utente ancora più dispettoso potrebbe prendere un file qualsiasi e rinominarlo...) verifico che il bundleIdentifier corrisponda a quello registrato, ovvero alla stringa `it.macocoa.djzero00.yahFortune`. Se così è, sono riuscito nell'impresa di localizzare l'applicazione; altrimenti, ci rinuncio in maniera definitiva.

Capitolo 070 - Un servizio di fortuna

Aggiungo un piccolo tassello a *YahFortune*, attraverso la realizzazione di un servizio (sì, quello che si trova nel menu *Servizi* di ogni applicazione).

Sorgenti: l'esempio Apple `Simple Service`.

Prima stesura: 16 agosto 2005

Servizi

Una delle caratteristiche di Mac OS X è la presenza, in ogni applicazione, del menu *Servizi*, proprio all'interno del menu principale di ogni applicazione.

Qui le normali applicazioni possono inserire delle voci tali da attivare alcune funzioni proprie dell'applicazione. Ad esempio, l'applicazione *Mail* inserisce delle funzioni: una per inviare una mail all'indirizzo che si suppone selezionato all'interno di una qualche altra applicazione, oppure per inserire il testo selezionato (in un contesto generico, anche nel Finder) in una nuova mail; cose del genere. Il mio intento è di aggiungere una voce a questo menu che produca, utilizzando tutto quanto già sviluppato finora in *YahFortune*, una stringa con un nuovo fortune.

Si tratta di far comunicare tra loro due applicazioni; la comunicazione avviene in maniera molto slegata, attraverso l'uso degli appunti. In altre parole, una applicazione fornisce servizi ricevendo informazioni attraverso gli appunti (un oggetto di tipo `NSPasteboard`) e restituisce risultati usando lo stesso meccanismo. Un servizio funziona da utilizzatore (*processor*) se riceve informazioni e le utilizza per eseguire qualche operazione (come appunto l'esempio di Mail citato sopra); oppure da fornitore (*provider*) se produce qualche informazione utilizzata dall'applicazione in primo piano

(che è appunto quello che intendo fare con il servizio di Fortune). Ovviamente, il servizio può funzionare sia da utilizzatore che da fornitore, se le operazioni svolte sono in qualche modo di *filtro* (ad esempio, il servizio di *Istantanea*).

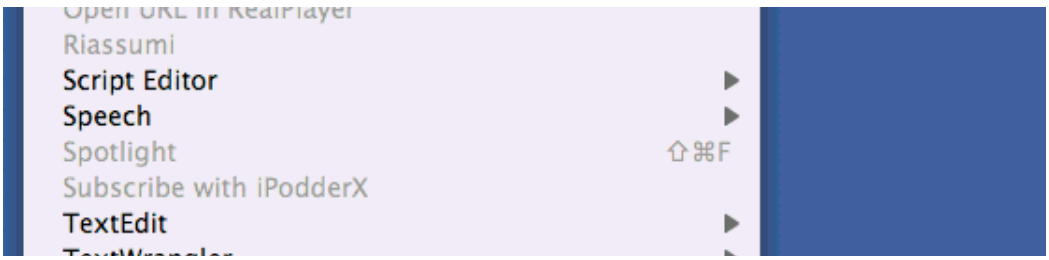
Le applicazioni fornitrici di servizi sono potenzialmente tutte le applicazioni disponibili. Mac OS X si preoccupa in piena autonomia di rintracciare tutte le applicazioni fornitrici di servizi e di inserire le voci di menu al loro posto. È inoltre possibile costruire della applicazioni specializzate nell'esclusiva fornitura di servizi. Queste applicazioni devono trovarsi all'interno di una cartella *Services* all'interno di una delle Librerie standard (di sistema, globali o per ogni singolo utente), ed avere come suffisso la dicitura `.service` al posto della classica `.app`. Per informare il sistema operativo su quali siano i servizi messi a disposizione da una applicazione, occorre riempire in maniera opportuna il file Info.plist dell'applicazione, secondo uno schema predefinito.

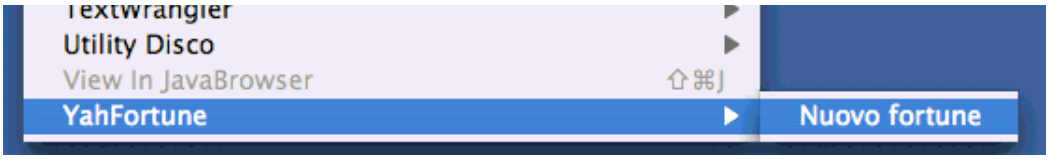
DICHIARAZIONE DI SERVIZIO

All'interno del file `Info.plist` dell'applicazione, oltre agli elementi standard, occorre infatti aggiungere qualcosa del genere.

```
    <key>NSServices</key>
    <array>
        <dict>
            <key>NSMenuItem</key>
            <dict>
                <key>default</key>
                <string>YahFortune/Nuovo fortune</string>
            </dict>
            <key>NSMessage</key>
            <string>makeFortuneService</string>
            <key>NSPortName</key>
            <string>YahFortunex</string>
            <key>NSReturnTypes</key>
            <array>
                <string>NSStringPboardType</string>
            </array>
        </dict>
    </array>>
```

La chiave `NSServices` avverte il sistema operativo che l'applicazione è in grado di fornire servizi. L'elenco dei servizi è indicato dallo array successivo, che nel mio caso è costituito da un solo elemento. Ogni singolo servizio è indicato attraverso un dizionario, composto da varie voci. In primo luogo, stabilisco il nome della voce che sarà inserita nel menu (il meccanismo key-string è utilizzato per l'internazionalizzazione). Nel menu *Servizi* comparirà quindi la voce `YahFortune`, dotata di un sottomenu (come indicato dal carattere /) denominato `Nuovo Fortune`.





Poi indico quale metodo verrà chiamato quando l'utente seleziona la voce. Detto meglio, quando si seleziona la voce `Nuovo fortune`, il sistema operativo richiederà all'applicazione l'esecuzione del seguente metodo:

```
-    (void)makeFortuneService:(NSPasteboard    *)pboard    userData:(NSString
*)userData error:(NSString **)error ;
```

dove il primo argomento riporta la selezione corrente, il secondo una eventuale stringa (specificata sempre all'interno di `Info.plist`) per caratterizzare meglio l'esecuzione del servizio, ed il terzo argomento è il posto dove, se il caso, riportare errori nell'esecuzione del servizio.

Con l'attributo `NSPortName` si attribuisce un identificatore al servizio. Deve corrispondere alla registrazione che si vedrà in seguito. Con `NSReturnTypes` si stabilisce il tipo di dati forniti dal servizio; nel mio caso, si tratta di una stringa. Questo tipo di dati è poi utilizzato da ogni applicazione per validare o meno la voce di menu del servizio. Dire che il servizio restituisce una stringa equivale ad avere la voce quasi sempre abilitata (in pratica in ogni situazione in cui è possibile incollare una stringa...).

PREPARAZIONE DI YAHFORTUNE

Per poter realizzare il servizio, devo modificare un po' il codice di YahFortune per poterlo riutilizzare il più possibile. La modifica principale, che a cascata richiede minori modifiche altrove, è la trasformazione del metodo `makeAFortune` della classe `frtWinCtrl` in una funzione all'interno del file `communUtils.m`. In questo modo posso eseguire la produzione di una stringa di fortune semplicemente utilizzando una funzione, slegandola dall'interfaccia dell'applicazione. Nell'operazione, le cose non cambiano molto:

```
NSString *
makeAFortune( NSBundle * fromBundle, NSMutableDictionary * currPrefs )
{
    ...
    // recupero il percorso dell'eseguibile all'interno dell'applicazione
    fortunePath = [ fromBundle pathForResource: @"fortune" ofType: @"" ] ;
    // costruisco un task apposito
    fortuneTask = [[NSTask alloc] init];
    ...
```

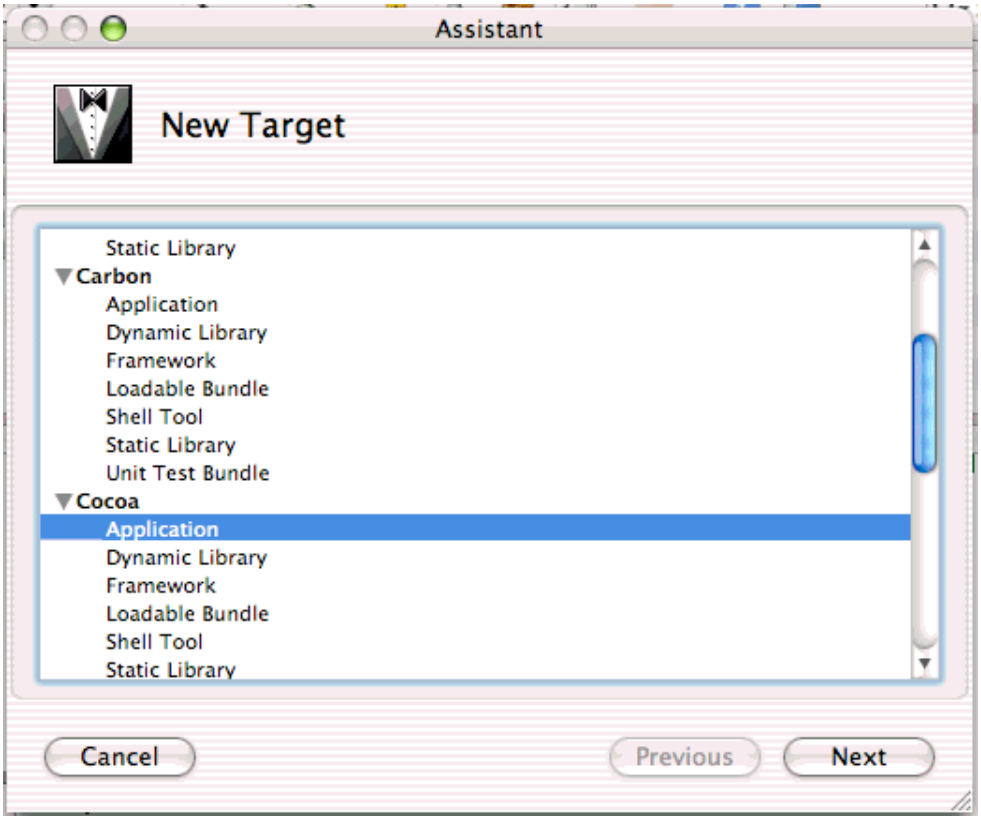
La presenza di una funzione modifica ovviamente il suo uso; ad esempio, all'interno del metodo `awakeFromNib` di `frtWinCtrl` si passa da

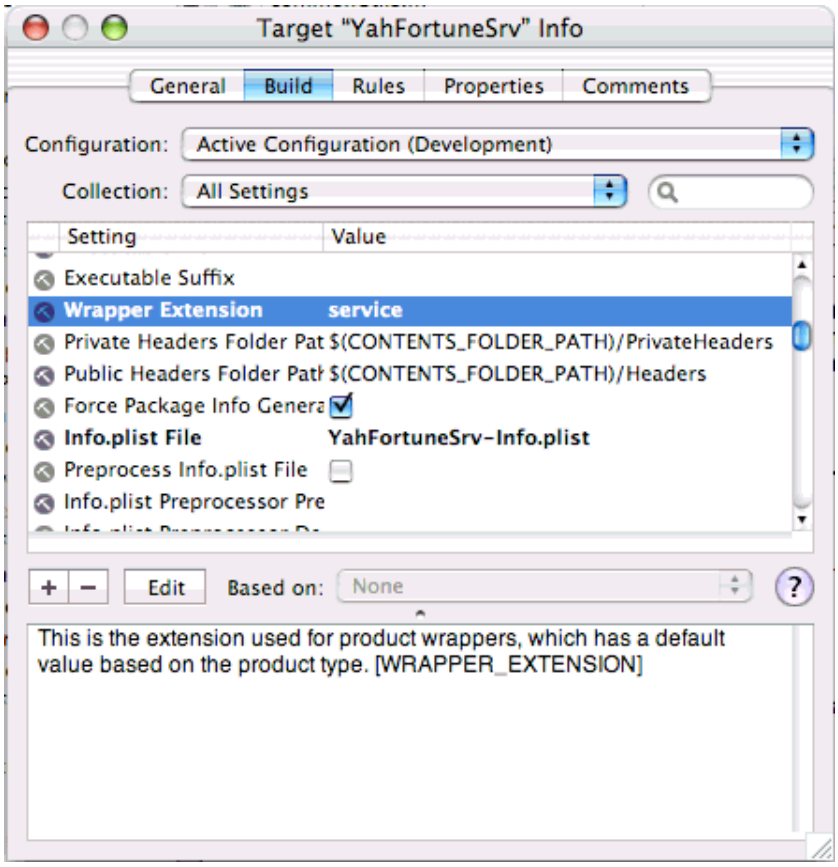
```
        [ self makeAFortune ] ;
```

a

```
        [ self writeFortuneOnfile: makeAFortune( [ NSBundle mainBundle ] ,
currPrefs) ];
```

E così via per tutti gli altri punti in cui era invocato il metodo.





A questo punto, sono pronto per creare all'interno del progetto un nuovo *Target* per la realizzazione del servizio. Aggiungo quindi il target *YahFortuneSrv*, scegliendo tra le varie possibilità quella di una applicazione Cocoa.

Ci sono due importanti modifiche da fare sulle opzioni di default del target. La prima consiste nel cambiare l'estensione del prodotto compilato da `.app` a `.service`. Per fare questo, devo aprire la finestra delle informazioni del target, e poi, nel tab *Build*, modificare la voce apposita.

La seconda operazione, che non sarebbe strettamente necessaria eseguire all'interno di XCode, ma che sveltisce il lavoro, è di spostare (o meglio, copiare) il prodotto `YahFortuneSrv.service` all'interno della cartella `~/Library/Services`. In questo modo, ad ogni compilazione, non si deve eseguire manualmente questa operazione dal Finder. Per fare questo, ho aggiunto una nuova fase alle operazioni di Build (c'è una apposita voce nel menu), una fase di copiatura di file. Ho definito la directory di destinazione nel pannello di informazioni relativo, e quale file copiare semplicemente trascinandolo sopra la cartella *Copy Files*.

Posso cominciare con lo scrivere il codice vero e proprio dell'applicazione.

CODICE DI SERVIZIO

Copiando brutalmente dall'esempio fornito da Apple, ecco il file principale:

```
#import <foundation foundation.h>
#import <cocoa cocoa.h>

#import "FortuneService.h"

int main (int argc, const char *argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    FortuneService *serviceProvider = [[FortuneService alloc] init];

    NSRegisterServicesProvider(serviceProvider, @"YahFortunex");

    NS_DURING
        [[NSRunLoop currentRunLoop] configureAsServer];
        [[NSRunLoop currentRunLoop] run];
    NS_HANDLER
        NSLog(@"%@", localException);
    NS_ENDHANDLER

    [serviceProvider release];
    [pool release];

    exit(0);     // insure the process exit status is 0
    return 0;     // ...and make main fit the ANSI spec.
}
```

Sostanzialmente, si costruisce una istanza di una classe (che sarà la destinataria del metodo `makeAFortune` chiamato attraverso la voce del menu), e poi si entra nel classico loop infinito tipico di una applicazione. All'interno di questo loop, si aspetta che qualcuno chiami il metodo. Da notare il collegamento con la voce del menu dei servizi, in cui l'applicazione si registra come fornitrice di servizi sulla porta con il nome presente all'interno del file `Info.plist`.

La classe `FortuneService` è piuttosto semplice (si fa per dire, ovviamente, ho penato un po'...). Il metodo principe è ovviamente il seguente:

```
- (void)makeFortuneService:(NSPasteboard *)pboard
            userData:(NSString *)userData
            error:(NSString **)error
{
    NSString *newString = nil;
    NSString *thePath = nil;
    NSArray *types;

            if ( thePath = getYahFortuneAppPath( yahFortuneAppPath,
PFC_yahf_prefFileName ) )
    {
        [ self setYahFortuneAppPath: thePath ];
           newString = makeAFortune( [NSBundle bundleWithPath: thePath] ,
getPrefFromFile( ) );
    }

    types = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:types owner:nil];
    [pboard setString:newString forType:NSStringPboardType];
    return;
}
```

La prima cosa da fare è di rintracciare il percorso dell'applicazione *YahFortune*, all'interno della quale risiede l'eseguibile fortune utilizzato per la produzione delle stringhe (questo perché ho voluto evitare di inserire una nuova copia del tutto all'interno dell'applicazione di servizio; mi sono complicato la vita, ma ho risparmiato quel mega di spazio su disco rigido...). Per svolgere questo compito, ho usato il metodo `getYahFortuneAppPath` della classe `yahFortunePref` (anzi, ho trasformato anche questo metodo in una funzione, l'ho spostato dalla classe nel file `commonUtils.m`).

Con il percorso dell'applicazione, è un gioco da ragazzi produrre una stringa `newString` con un nuovo fortune. Di seguito, ci sono le tre istruzioni che mettono il risultato a disposizione del mondo. Si prepara l'oggetto `NSPasteboard` che serve da comunicazione dicendo che al suo interno ci sarà appunto una stringa, e si inserisce la stringa al suo interno. Fine.

Però per fare funzionare le cose, ho penato parecchio, finché non ho spento tutto e riacceso (una delle regole fondamentali di ogni informatico). Infatti, i servizi sono esaminati dal sistema operativo allo startup; occorre quindi effettuare il `logout` e quindi il `login` per avere nel menu dei servizi l'apposita voce.

Che funziona. Quasi ovunque. *TextEdit*, *XCode*, ogni editor di testi funziona. Non funziona però, stranamente, con *Mail* (che, va detto, era lo scopo principale del servizio...). Chissà perché.

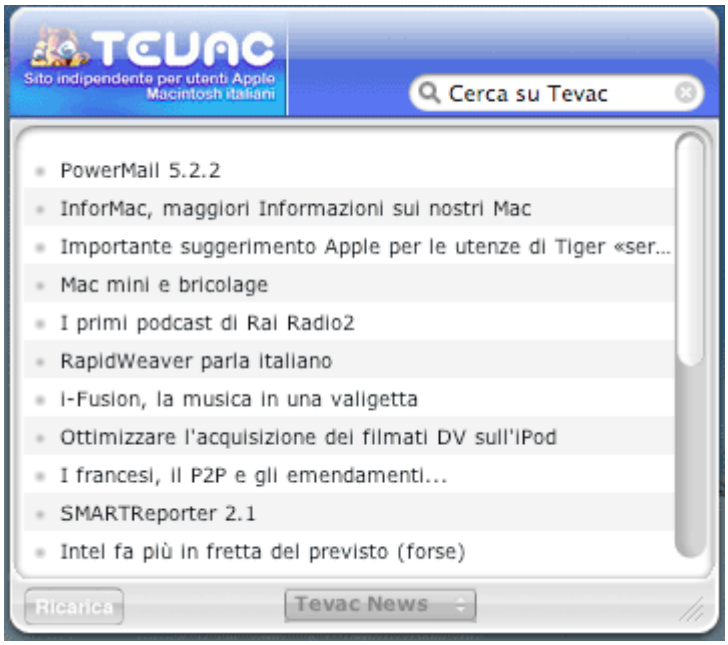
Capitolo 071 - Plugin x Widget = Facile!

Qualche tempo fa mi sono trovato a realizzare un widget (per accedere ai feed RRS del mai abbastanza lodato sito Tevac). Non parlo qui del widget in sé, ma di come ha realizzato un plugin in Cocoa per effettuare alcune operazioni all'interno del widget stesso. La cosa si è rivelata piuttosto semplice, anche se piuttosto semplice era il compito da svolgere.

Sorgenti: esempi Apple e tanto Google.

Prima stesura: 25 dicembre 2005 (!!!)

Il widget ed il plugin



Non parlo del widget. Del resto, il widget stesso e tutto il suo codice javascript, html e css è accessibile direttamente non appena lo avete disponibile (è sufficiente utilizzare il menu contestuale `Mostra Contenuto Pacchetto` una volta selezionato il file che lo rappresenta). Basti dire che il widget piglia un `feed rss` (ovvero, un `file xml`), lo legge, ne ricava una serie di elementi formati da titolo, indirizzo ed eventualmente una descrizione più estesa, e li presenta all'interno di una coreografica finestra.

Le funzioni della prima versione del widget erano completamente realizzate tramite Javascript. Purtroppo, soffriva di un problema non indifferente per la lingua italiana: le lettere accentate non erano correttamente visualizzate. Piuttosto che stare lì a combattere con Javascript, mi son detto: perché non fare un plugin in Cocoa? Nel farlo, oltre a pasticciare con la rete, avrei anche imparato, appunto, come fare un plugin per un widget.

Quindi, in questo breve ed avulso capitolo di Macocoa, parlo di come realizzare un

plugin per un widget. Il plugin è molto semplice: piglia un url, che dovrebbe rappresentare un file xml con all'interno un feed rss, e lo interpreta come una serie di elementi. Ciascun elemento è costituito da tre stringhe: un titolo, un link, ed una descrizione. Il plugin legge il file, interpreta le informazioni e le rende disponibili alla bisogna. Tutto qui.

MODEL

Quando si programma orientati agli oggetti, il paradigma MVC (Model, View, Controller) salta fuori quasi sempre. Anche nel caso di un plugin, abbiamo (benché sia assente la parte View, ovviamente a carico del widget) da considerare la parte di modellizzazione dei dati e la parte di controllo dell'interazione.

Comincio dalla parte di modellizzazione. È quasi giocoforza costruire una classe che rappresenti un elemento del feed rss:

```
@interface rssItem : NSObject {
    NSString    * theTitle ;
    NSString    * theLink ;
    NSString    * theDescr ;
}

- (id ) initWithTitle: (NSString*) theTit link: (NSString*) theLnk andDesc:
(NSString*) theDesc ;

- (NSString *)theTitle;
- (void)setTheTitle:(NSString *)value;

- (NSString *)theDescr;
- (void)setTheDescr:(NSString *)value;

- (NSString *)theLink;
- (void)setTheLink:(NSString *)value;
```

Nulla di più semplice: un oggetto con tre variabili d'istanza, appunto titolo, link e descrizione, e metodi per inizializzarlo e accedere ai singoli elementi.

Non offenderò la vostra intelligenza esaminando nei dettagli i triviali metodi sopra indicati, ma parlo solo del metodo seguente (che funzionicchia, non lo nascondo):

```
NSMutableString *
adjustString ( NSString * src )
{
    NSMutableString    * t = [ NSMutableString stringWithString: src ];
                    [t     replaceOccurrencesOfString:@"&"     withString:@"&"
options:NSLiteralSearch range:NSMakeRange(0,[t length])];
                   [t     replaceOccurrencesOfString:@"""     withString:@"\""
options:NSLiteralSearch range:NSMakeRange(0,[t length])];
    return ( t );
}
```

Ho introdotto questo metodo per trasformare alcune entità html che non ne volevano sapere. In realtà, ci sono altre entità che continuano a non volerne sapere (strani accenti e virgolette), ma non ho voglia di perderci troppo tempo.

Il passo successivo è la costruzione dell'insieme degli elementi a partire dallo url indicato. Mi serve un'altra classe, che è anche la classe principale di tutto il plugin, che svolge le funzioni restanti di modellizzazione, e le poche funzioni di controllo.

CONTROLLER

La classe `TevacRssRead` ha una sola variabile d'istanza:

```
@interface TevacRssRead : NSObject {
    NSMutableArray        * elementiRss ;
}
```

Si tratta di un array di oggetti `rssItem`, sopra introdotti. Per riempire questo array, ecco il metodo utilizzato. Assume come parametro lo url del file, ed esegue una serie di operazioni noiose che vado per ordine a commentare:

```
- (NSString *)
getRSSFeed: (NSString *)feedURL
{
    ...
    // trasformo la stringa in ingresso in un url
    theUrl = [ NSURL URLWithString: feedURL ];
    // prelevo il file dal server usando una codifica opportuna
    theStart = [ NSString    stringWithContentsOfURL: theUrl
                            encoding: NSISOLatin1StringEncoding
                            error: & theErr ];
    // controllo la presenza di errori attraverso il risultato
    if ( theStart == nil )
    {
            NSLog(@"TevacRssWidget plugin, getRSSFeed Error: %@", [theErr
description] );
        return ( nil );
    }
    // trasformo la stringa in un documento XML
    // non ho fatto tutto direttamente perche' costruendo un XML
    // direttamente dallo URL, trasforma subito le lettere strane
    theXml = [ [ NSXMLDocument alloc] initWithXMLString: theStart
                    options: NSXMLDocumentTidyXML error: & theErr ];
```

La prima parte del codice si occupa di leggere il file XML. In realtà, le prime due istruzioni eseguono già il grosso del lavoro. Parto dalla `NSString` in ingresso come argomento del metodo, e la trasformo in un comodo url da utilizzare con le classi Cocoa. Lo `NSUrl` appena costruito è dato in pasto ad un potente metodo di `NSString`, che costruisce un oggetto con tutto il contenuto del documento indicato dallo url. A questo punto, la variabile `theStart` contiene già tutte le informazioni necessarie. Il primo passo è di trasformarlo in un oggetto `NSXMLDocument`. E qui, bisogna fare un momento di pausa, anzi due.

Primo punto: un oggetto della classe `NSXMLDocument` è di una comodità straordinaria quando si tratta di esaminare documenti in formato XML; questa classe nasce a partire da Mac Os X 10.4. Il fatto non è un problema in questa situazione, in quanto i widget stessi sono presenti a partire da questa versione del sistema operativo; ovviamente, se avete versioni più datate, questa classe non è utilizzabile.

Per quei pochi sprovveduti che non conoscono XML (me compreso), basti dire che i documenti XML sono file di testo che sono i cugini più scaltri e potenti dei file HTML. Un file XML serve a comunicare informazioni in maniera strutturata, e la struttura stessa del messaggio è descritta all'interno del file attraverso dei classici *Tag*. Un esempio è in genere più illuminante di qualsiasi spiegazione. Ecco come si presenta il feed di Tevac:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<rss version="0.91">
    <channel>
        <title>Tevac RSS (solo titoli)</title>
        <link>
            http://www.tevac.com
        </link>
        <description>
            Open Source Mac Site
        </description>
        <language>
            en-gb
        </language>
        <item>
            <title>PowerMail 5.2.2</title>
            <link>
                http://www.tevac.com/article.php/2005122400425439
            </link>
        </item>
...
    </channel>
</rss>
```

Il file dichiara inizialmente che tipo di caratteri contiene e quale versione delle specifiche rss utilizza, poi presenta un canale di feed, del quale specifica nell'ordine il titolo, il link di riferimento ed una descrizione. Dichiara (sbagliando) in che lingua si sta ragionando, e poi passa ad elencare, item dopo item, le varie notizie presenti nel sito. In questo caso, le notizie sono presenti solo come titolo e link (altre versioni del file hanno anche una descrizione dell'articolo cui fanno riferimento). Maggiori informazioni presso la pagina di Tevac relativa.

Data la natura del file, la cosa più immediata per estrarre le informazioni è di cominciare, con pazienza, a lavorare con le stringhe ed estrarre i vari elementi delle informazioni. Ma perché lavorare quando se ne può fare a meno? La classe `NSXMLDocument` serve proprio a questo. Inizializzando tale classe con la stringa che rappresenta il contenuto del file, sarà la classe stessa ad eseminarne il contenuto, e costruire una struttura (necessariamente ad albero, per quelli tra voi che hanno una spruzzata di informatica nel proprio bagaglio culturale) di navigazione più adatta alla bisogna. Per vederla all'opera, basterà attendere qualche paragrafo.

Secondo punto: la classe `NSXMLDocument` presenta, tra i suoi metodi, il seguente:

```
- (id)initWithContentsOfURL:(NSURL *)url options:(unsigned int)mask
error:(NSError **)error
```

A che pro, quindi, utilizzare una istruzione per recuperare il file indicato dallo url in una `NSString`, e poi utilizzare tale `NSString` per costruire lo `NSXMLDocument`? Non si poteva fare tutto in una unica istruzione?

Certo. Solo che, per ragioni che non mi sono chiare, il singolo passaggio perde le lettere accentate, che è proprio il problema che ha portato alla nascita del plugin. Detto in altre parole: se leggo il file rss con una unica istruzione, le lettere accentate sono rappresentate da caratteri strani e diventa un problema recuperale; se eseguo la lettura in una istruzione, e la trasformazione in xml in una seconda istruzione, le lettere accentate rimangono. Tutto ciò rimane per me molto misterioso, posso supporre che abbia a che fare con la codifica dei caratteri e i protocolli di trasferimento dati, ma, in base all'esperienza derivante da lunghi anni di battaglie informatiche (quando va che tanto basta, non toccare, ché si guasta), lascio le cose come stanno e non indago oltre.

Posso finalmente proseguire con un po' di controlli sulla natura del file. È bene verificare che il file letto sia un file rss valido, e che contenga un canale di notizie:

```
    // preparo una stringa per il risultato
    theRes = [ [ NSMutableString alloc ] init ] ;
    // estraggo l'elemento root
    aNode = [theXml rootElement] ;
    // deve essere un "rss"
    if ( [ [ aNode name ] compare: @"rss" ] != NSOrderedSame )
    {
         NSLog( @"TevacRssWidget plugin, not a valid feed (rss) -> %@", [
aNode name ]);
        return ( nil );
    }
    // il prossimo figlio dovrebbe essere un "channel"
    aNode = [aNode nextNode] ;
    if ( [ [ aNode name ] compare: @"channel" ] != NSOrderedSame )
    {
         NSLog( @"TevacRssWidget plugin, not a valid feed (chn) -> %@", [
aNode name ]);
        return ( nil );
    }
    // a questo punto sembra che abbiamo un feed ragionevole
```

Direi che ci siamo: buttiamo via gli eventuali elementi presenti in precedenza, e passiamo a leggere gli elementi di tipo `item` presenti all'interno del canale.

È da notare come si naviga all'interno di un file xml strutturato. Si parte da un elemento `root`, radice, e si salta di nodo in nodo con il metodo `nextNode`, oppure, se si tratta di nodi allo stesso livello, con il metodo `nextSibling` (dove ogni nodo è un pezzo di informazione racchiuso tra tag). Se un nodo è strutturato, si accede ai vari elementi contenuti nel suo interno attraverso il vettore dei figli (metodo `children`).

```
    // rilascio l'attuale collezione
    [ elementiRss removeAllObjects ];
    // esploro l'albero xml, entro nel livello interno
    aNode = [aNode nextNode] ;
    // esploro finche' ci sono nodi
    while ( aNode )
    {
        // mi interessano solo gli item
        if ( [ [ aNode name ] compare: @"item" ] == NSOrderedSame )
        {
            // se entro qui, ho trovato un item
            rssItem            * aNewItem ;
            NSString        * locTitle = nil ;
            NSString        * locLink = nil ;
```

```
            NSString        * locDesc = nil ;
            NSXMLNode        * figghiu ;
            // esamino tutti i figli e metto da parte i valori
            NSEnumerator *en = [ [ aNode children ] objectEnumerator];
            while (figghiu = [en nextObject])
            {
                // mi interessano title, link e description
                        if ( [ [ figghiu name ] compare: @"title" ] ==
NSOrderedSame )
                         locTitle = [ NSString stringWithString: [ figghiu
stringValue ]] ;
                 if ( [ [ figghiu name ] compare: @"link" ] == NSOrderedSame
)
                          locLink = [ NSString stringWithString: [ figghiu
stringValue ] ];
                     if ( [ [ figghiu name ] compare: @"description" ] ==
NSOrderedSame )
                          locDesc = [ NSString stringWithString: [ figghiu
stringValue ] ];
            }
            // ho un altro elemento da inserire
            rowCount += 1 ;
               aNewItem = [[ rssItem alloc] initWithTitle: locTitle link:
locLink andDesc:locDesc ];
            [ elementiRss addObject: aNewItem ];
        }
        // passo al nodo successivo
        aNode = [ aNode nextSibling ];
    }
    // restituisco il numero di elementi presenti
    return ( [ NSString stringWithFormat: @"%d", rowCount] );
}
```

Per ogni elemento riconosciuto come `item`, provo a vedere se dispone di un titolo, di un link e di una descrizione. Con questi tre oggetti stringa poi costruisco un oggetto rssItem, che vado ad aggiungere alla mia collezione. Al termine delle operazioni, restituisco (sotto forma di stringa, piuttosto che di numero puro, che è difficile da trasferire indietro al widget) il numero di elementi presenti.

IL PLUGIN

In effetti, nulla di quanto visto finora ha un legame diretto con il fatto di stare in un plugin per widget. Nulla impedisce di utilizzare il metodo precedente all'interno di una applicazione generica per leggere un feed rss generico (anzi, all'inizio ho fatto proprio così, per sperimentare con url e xml). Adesso comincia la parte più specifica.

Parto, come al solito, dalla documentazione fornita da Apple. Questa dice che per fare un plugin occorre costruire un oggetto che risponda ad un po' di metodi (un protocollo!).

Il primo metodo è il seguente:

```
- (id) initWithWebView:(WebView*)webview
```

È chiamato all'inizializzione del widget (quando la view del view è inizializzata, verosimilmente), ed è utilizzabile per tutte le inizializzazioni del caso. Io ne approfitto per aggiustare il vettore degli elementi:

```
-(id)
initWithWebView: (WebView*)w
{
    // giusto per scrupolo
    self = [ super init ];
    if ( self )
    {
        elementiRss = [ NSMutableArray arrayWithCapacity: 0];
        [ elementiRss retain ];
    }
    return self;
}
```

Il secondo metodo da realizzare (sempre se si vuole che il plugin interagisca col widget, cosa che si suppone essere vera, altrimente serve a nulla) è:

```
- (void) windowScriptObjectAvailable:(WebScriptObject *)windowScriptObject
```

Con questo metodo, indico con quale nome è possibile chiamare il plugin attraverso istruzioni Javascript:

```
-(void)
windowScriptObjectAvailable:(WebScriptObject*)wso
{
    [ wso setValue:self forKey:@"TevacRSSPlugin"];
}
```

In altre parole, dall'interno del widget posso invocare i metodi e le variabili d'istanza delle classi presenti all'interno del plugin tramite il nome `TevacRSSPlugin`, come ad esempio (attenzione, il seguente è codice Javascript!!!):

```
    numofelem = TevacRSSPlugin.getRSSFeed( url );
```

Il passo successivo è la realizzazione del protocollo (informale: significa che non è proprio necessario realizzare tutti i metodi indicati, al limite la cosa non funziona) `WebScripting`. Il protocollo regolamenta come un plugin scritto in cocoa possa interagire con un linguaggio di scipting esterno.

Il primo metodo interessante nello scopo ma assolutamente noioso nella realizzazione mette a disposizione i metodi interni tramite nome:

```
+(NSString*)
webScriptNameForSelector:(SEL)aSel
{
    NSString    * selname = nil;
    if (aSel == @selector( getRSSFeed: ) )
    {
        selname = @"getRSSFeed";
    }
    else if (aSel == @selector( logMessage: ) )
    {
        selname = @"logMessage";
    }
    ...
    return selname;
}
```

Per ogni metodo presente nella classe, restituisco una stringa, che sarà quella da utilizzare all'interno di Javascript per chiamare il metodo corrispondente. Io mi sono limitato ad utilizzare gli stessi nomi.

I due metodi successivi regolano l'accesso ai metodi ed alle variabili d'istanza del plugin al resto del mondo:

```
+(BOOL)
isSelectorExcludedFromWebScript: (SEL)aSel
{
    // i miei due metodi sono accettabili
    if ( aSel == @selector(getRSSFeed:)
        || aSel == @selector(logMessage:)
        ...
        )
    {
        return NO;
    }
    return YES;
}

+(BOOL)isKeyExcludedFromWebScript:(const char*)k
{
    return YES;
}
```

Con il primo metodo, indico esplicitamente (rispondendo NO alla domanda se escluderli o meno dall'interfaccia) quali sono i metodi del plugin che possono essere chiamati dall'esterno; ovviamente, i metodi permessi devono avere una corrispondenza con i nomi indicati dal metodo `webScriptNameForSelector`. Con il secondo metodo, rispondendo sistematicamente YES, dico che in pratica qualsiasi variabile d'istanza del plugin non è direttamente accessibile dall'esterno. Così facendo, mi sono risparmiato la scrittura del metodo

```
+ (NSString *)webScriptNameForKey:(const char *)name
```

che è l'equivalente di `webScriptNameForSelector` per le variabili d'istanza.

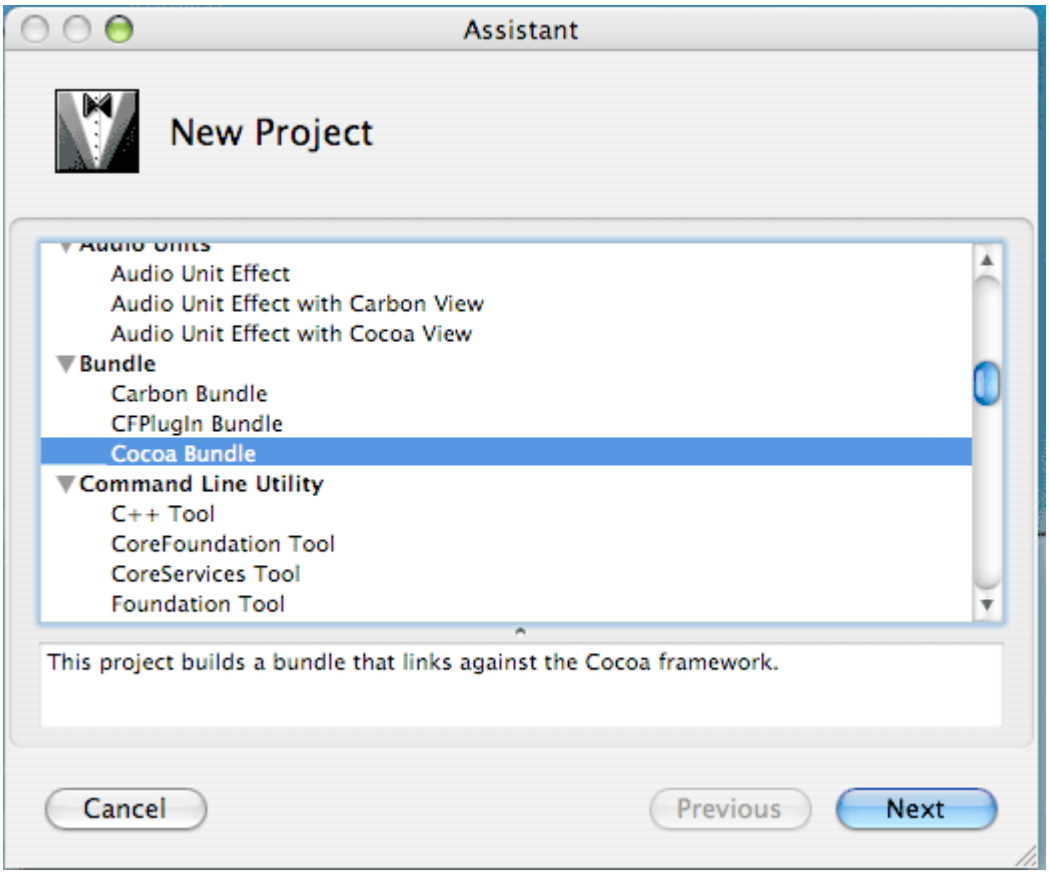
Per accedere alla variabili d'istanza, o meglio ai singoli elementi caricati dal feed rss, ho dei metodi espliciti che si basano sul numero d'ordine dell'elemento all'interno del vettore:

```
- (NSString *)
getTitleForItem: (NSString *) theIdx
{
    unsigned short    idx = [ theIdx intValue ];
    if ( [ elementiRss count ] < idx )
        return ( nil );
    return ( [ [ elementiRss objectAtIndex: idx] theTitle ] );
}
```

assieme ai molto simili `getLinkForItem` e `getDescForItem`.

Bene, i metodi fin qui visti sono tutti quelli necessari per partire. Ma solo adesso arriva la parte meno ovvia.

IL PROGETTO XCODE

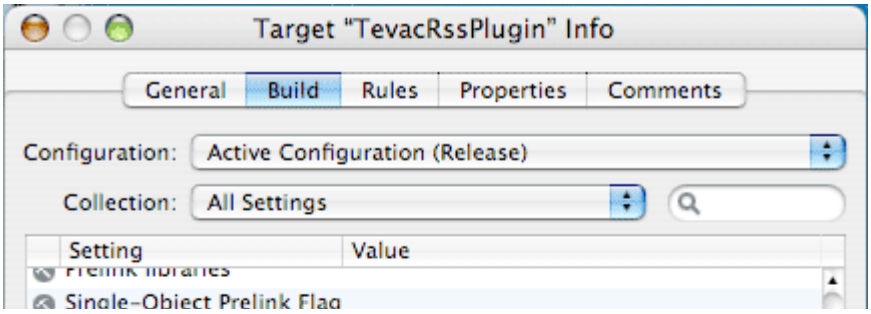


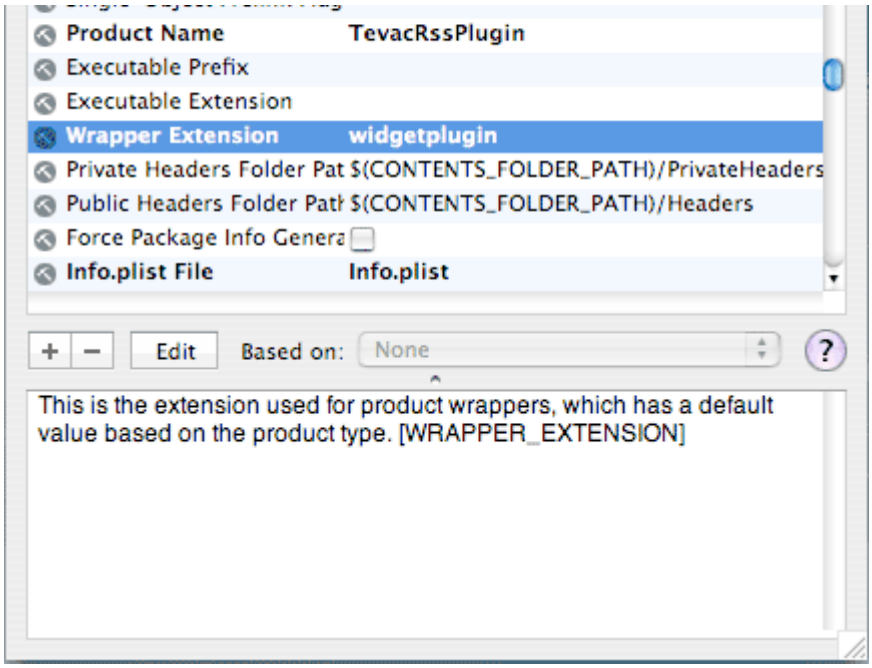
Un plugin non è un'applicazione Cocoa, non è un tool per linea di comando, non è alcuna delle voci presenti nella finestra che compare quando si costruisce un nuovo progetto con XCode. Cosa fare allora?

Si deve selezionare la voce generica `Cocoa Plugin`, e modificare un po' di cose.

In primo luogo, si deve modificare l'estensione del file prodotto dalla compilazione: si cerca nelle informazioni relative al target la `Wrapper Extension` e la si pone a `widgetplugin`. Visto che si accede alla rete per recuperare file, occorre inserire all'interno del progetto anche il framework `WebKit.framework`, che trovate (con un po' di fatica: `/System/Library/Frameworks/WebKit.framework`) all'interno delle librerie di sistema. Poi, all'interno del file `Info.plist`, occorre aggiungere la voce:

```
    <key>NSPrincipalClass</key>
    <string>TevacRssRead</string>
```





per indicare quale sia la classe da prendere come riferimento per il plugin (che poi sarebbe la classe che realizza il protocollo `WebScripting`). Va da sé che il risultato della compilazione va inserito all'interno della cartella del widget, al livello principale dello stesso.

Ci sono anche due accorgimenti all'interno del widget, modificando il file `Info.list` del widget stesso (attenzione, non quello del plugin, quello del widget): notificare al widget l'esistenza del plugin

```
    <key>Plugin</key>
    <string>TevacRssPlugin.widgetplugin</string>
```

e poi (ma questo solo perché il widget accede alla rete; e comunque sarebbe necessario, nel caso del widget di Tevac, per poter accedere alla rete esterna al computer)

```
    <key>AllowNetworkAccess</key>
    <true/>
```

Senza questi quattro infimi ed indispensabili dettagli, non funziona alcunché.

Finalmente, è possibile utilizzare il plugin all'interno del widget. Ecco ad esempio come fare a leggere un file rss e costruire la struttura delle ultime news dal sito:

```
        if (TevacRSSPlugin)
        {
            var theFeed ;
            ...
            xx = feedData[selectedFeed] ;
            numofelem = TevacRSSPlugin.getRSSFeed( xx[0]);
            var cnt = document.createElement ('div');
            for (var i = 0; i < numofelem ; ++i)
            {
                // prelevo l'elemento i-esimo
                var item1 = TevacRSSPlugin.getTitleForItem( i );
```

```
            var item2 = TevacRSSPlugin.getLinkForItem( i );
            var item3 = TevacRSSPlugin.getDescForItem( i );
            // costruisco una riga
            var row = createRow (item1, item2, item3, i );
            // la inserisco nel contenuto
            cnt.appendChild (row);
        }
        ...
    }
    else
    {
        DEBUG("Widget plugin not loaded.");
    }
```

Con tutto ciò, sono arrivato alla fine del plugin, dal comportamento soddisfacente per i miei miseri scopi, e spero adatto anche ai vostri (il widget è la prima opera con un mio contributo rilasciata al grande pubblico!!!).

Capitolo 072 - Key-Value coding

Comincio una serie di pezzi in cui non c'è codice strutturato, ma solo auto-istruzione. Serviranno a riprendere mano alla programmazione in Objective C e Cocoa, dal momento che è passato parecchio tempo dall'ultima volta che ho scritto una riga di codice funzionante.

Sorgenti: Riprendo, traduco e riassumo: Key-Value Coding Programming Guide di Apple.

Prima stesura: 29 aprile 2006

Intro a KVC

Il **Key Value Coding** KVC è un meccanismo che permette di accedere alle proprietà di un oggetto indirettamente, senza utilizzare direttamente i metodi accessor o la lettura/scrittura della corrispondente variabile d'istanza.

A quanto pare, è un meccanismo indispensabile per rendere una applicazione scriptabile (ovvero, pilotabile tramite applescript) con facilità. Ma il meccanismo KVC è utile anche per semplificare la scrittura delle applicazioni. Mi riferirò a tutto ciò come al paradigma KVC, inteso come la teoria che inquadra tutta l'operatività relativa a questi meccanismi per la manipolazione delle proprietà di un oggetto

Tutto quello che si deve fare è rispettare un protocollo (che è informale, ovvero non deve essere dichiarato esplicitamente) chiamato `NSKeyValueCoding`. Il protocollo, in essenza, è costituito da due metodi, uno per leggere la proprietà in base ad un nome, ed il secondo per impostarne il valore:

```
- (id)valueForKey:(NSString *)key
- (void)setValue:(id)value forKey:(NSString *)key
```

L'esempio classico è la produzione di dati per una tabella. Il trucco spesso utilizzato per semplificare la stesura dei metodi della classe dataSource è di dare alle colonne della tabella un nome uguale a quello della variabile. Il nome alle colonne è dato (ad esempio) tramite Interface Builder in sede di costruzione dell'interfaccia, impostato nel campo identifier. Recuperando tale nome all'interno del metodo che provvede alla fornitura dei dati, si può risalire in maniera molto semplice al valore dell'elemento, proprio utilizzando il nome e il metodo `valueForKey:`

```
- (id)
outlineView:                (NSOutlineView *) outlineView
    objectValueForTableColumn:    (NSTableColumn *) tableColumn
    byItem:                (id)item
{
    NSString * colId ;
    // recupero l'identificatore della colonna
    colId = [ tableColumn identifier] ;
    // in tutti gli altri casi, uso il metodo valueForKey
    return ( [ item valueForKey: colId ] ) ;
}
```

Mi pare di avere utilizzato estesamente questo trucco quasi ogni volta che dovevo fornire dati ad una tabella, anche se in effetti non ne ero completamente consapevole. Approfitto per approfondire la questione, soprattutto per arrivare alla scriptabilità dell'applicazione.

TERMINI

Attraverso KVC si può accedere al valore di tre diversi tipi di proprietà di un oggetto: gli attributi, relazione singola, e relazione a molti. Poiché i nomi italiani per gli ultimi due mi fanno ribrezzo e sono una mia approssimativa traduzione, utilizzo gli originali nomi inglesi: to-one relationship e to-many relationship.

Un attributo è una proprietà che si esprime direttamente attraverso un valore, di varia natura, ma che è legato strettamente all'oggetto di partenza. Sono quindi attributi tutte le variabili d'istanza esprimibili come variabili standard del linguaggio (ovvero numeri interi, floating point o altro), ma anche oggetti "semplici", quali `NSString`, `NSColor`, `NSNumber`, eccetera.

La proprietà to-one relationship collega l'oggetto di partenza con un altro oggetto, suscettibile anch'esso di essere dotato di proprietà.

La proprietà to-many relationship collega l'oggetto di partenza con una collezione di altri oggetti, ciascuno dei quali suscettibili di proprietà; è il caso classico di una variabile d'istanza di tipo `NSArray` (o simili).

Per fare un esempio, che poi userò estesamente anche più avanti, dichiaro le seguenti due classi:

```
@interface primaClasse : NSObject {
    unsigned long    varLong ;
    NSString        * altroTesto ;
}

@end

@interface secondaClasse : NSObject {
    unsigned long    varLong2 ;
    NSString        * testo ;
    NSMutableArray    * vettore ;
    primaClasse        * objinterno ;
}

@end
```

Nella classe `secondaClasse` si riconoscono gli attributi `varLong2` (facilmente perché numero scalare) e `testo` (un oggetto `NSString`); la to-one relationship `objinterno` e la to-many relationship `vettore`.

Per chiudere con le definizioni, abbiamo poi la chiave (`key`) ed il percorso chiave (`key path`). La chiave è il nome della proprietà, e quindi una stringa (un oggetto `NSString`). Il percorso chiave è invece una stringa che specifica un modo per recuperare una proprietà attraverso un percorso. Il percorso è una successione di chiavi separate da un punto: ad esempio `chiave1.chiave2` significa individuare in prima battuta la

proprietà `chiave1`. Questa dovrebbe individuare un oggetto, tra le proprietà del quale c'è `chiave2`. Il percorso chiave individua quindi il valore della proprietà `chiave2` dell'oggetto individuato dalla proprietà `chiave1`.

LEGGERE E SCRIVERE

Per sfruttare le potenzialità del paradigma KVC occorre realizzare i metodi del protocollo `NSKeyValueCoding`. Il protocollo è suscettibile di una realizzazione di default che richiede pochi accorgimenti nella stesura delle classi. Per il momento, esamino solamente l'interfaccia.

Per leggere una proprietà, ho già citato il metodo principe per accedere ad una proprietà attraverso il suo nome:

```
- (id)valueForKey:(NSString *)key
```

Se quindi ho un elemento `item` del quale mi interessa conoscere il valore della proprietà `nomeProp` mi basta scrivere:

```
val = [ item valueForKey: @"nomeProp" ];
```

Il metodo è così gentile da fornire sempre un oggetto, anche se il valore della proprietà è un tipo standard, come intero o floating point. Il metodo quindi costruisce un oggetto `NSNumber` o `NSValue` appropriato e lo inizializza al valore della proprietà.

Vale la pena notare che il paradigma KVC è utile quando non è noto in sede di compilazione il nome della proprietà desiderata; infatti, in tal caso, è molto più pratico ed efficiente accedere direttamente alla variabile, o meglio ancora, utilizzare il metodo accessor corrispondente. Quindi, in altre parole, in modo forse meno chiaro ma che per me rende meglio la potenza, il paradigma KVC è utile quando la proprietà desiderata non è nota al momento della compilazione, ovvero, tipicamente quando il nome della proprietà è stabilito dal contenuto di una variabile. Senza i meccanismi propri del KVC, si dovrebbe scrivere un metodo piuttosto pedissequo: esaminare il contenuto della variabile, e, a seconda del suo valore, attraverso una catena di if (visto che sarà raramente possibile utilizzare vettori o costrutti switch), distinguere tutti i vari casi possibili. Ad esempio, nel caso dei metodi per fornire dati ad una tabella, nel caso di utilizzo del trucco KVC, basta scrivere quelle poche righe viste in un esempio poco sopra; senza KVC, si deve scrivere qualcosa del tipo:

```
- (id)
outlineView:                (NSOutlineView *) outlineView
    objectValueForTableColumn:    (NSTableColumn *) tableColumn
    byItem:                (id)item
{
    NSString * colId ;
    // recupero l'identificatore della colonna
    colId = [ tableColumn identifier] ;
    if ( [ colId isEqual: @"colonna1" ] )
    {
        return ( [ item valorePerColonna1 ] );
    }
```

```
    if ( [ colId isEqual: @"colonna2" ] )
    {
        return ( [ item valorePerColonna2 ] );
    }
    // eccetera
}
```

Se la specifica della proprietà è nota attraverso un percorso chiave, il metodo da utilizzare è:

```
- (id)valueForKeyPath:(NSString *)keyPath
```

dove `keyPath` è una stringa con nomi separati da punti. Immagino che la versione standard di questo metodo (già realizzato da Apple per tutti noi) non faccia altro che esaminare i singoli costituenti del percorso, valutarli col metodo `valueForKey`, e passare all'elemento successivo.

Se infine si volessero recuperare un po' di proprietà in un colpo solo, esiste il metodo

```
-(NSDictionary *)dictionaryWithValuesForKeys:(NSArray *)keys
```

dove si passa un vettore di chiavi, e viene restituito un dizionario in cui ad ogni chiave è associato il valore della proprietà corrispondente.

Viene da chiedersi cosa accade quando la chiave o il percorso chiave specifica una proprietà inesistente (non si sa mai). Il protocollo afferma che in tal caso è chiamato il metodo

```
- (id)valueForUndefinedKey:(NSString *)key
```

il quale, normalmente, genera una eccezione (qualunque cosa sia: mi figuro un messaggio di errore un po' più complicato del solito). La cosa interessante è che, in una mia classe, posso sovrascrivere il metodo, e quindi restituire un valore di default, o qualcos'altro di piacevole.

Per assegnare valori alle proprietà, esistono i metodi corrispondenti a quelli appena visti:

```
- (void)setValue:(id)value forKey:(NSString *)key
- (void)setValue:(id)value forKeyPath:(NSString *)keyPath
- (void)setValuesForKeysWithDictionary:(NSDictionary *)keyedValues
- (void)setValue:(id)value forUndefinedKey:(NSString *)key
```

rispettivamente per impostare il valore di un proprietà attraverso il nome; attraverso un percorso chiave; per impostare contemporaneamente più proprietà; per evidenziare cosa succede in caso di non esistenza della proprietà indicata (ancora una volta, un'eccezione).

LA REALIZZAZIONE

La realizzazione standard del meccanismo KVC utilizza i metodi accessor. Se questi

sono fatti appropriatamente, in pratica non occorre scrivere una linea di codice. Infatti, il metodo `valueForKey` utilizza la stringa passata come argomento come nome del metodo accessor. Ovvero, se la proprietà si chiama `nomeProp`, il metodo `valueForKey` chiama il metodo nomeProp oppure `isNomeProp` per la lettura, ed il metodo `setNomeProp` per la scrittura. Se quindi scrivo i metodi accessor della classe rispettando questo schema, il protocollo `NSKeyValueCoding` è automaticamente soddisfatto, senza scrivere una linea aggiuntiva di codice.

Parentesi: che differenza c'è tra il metodo accessor `nomeProp` e `isNomeProp`? Le linee guida Apple dicono di usare `nomeProp` nel caso in cui la proprietà sia espressa attraverso un nome o una azione (verbo), e di utilizzare `isNomeProp` se la proprietà è espressa da un aggettivo. Tutto molto bello, ma il problema adesso è: quando utilizzare un nome e quando un aggettivo? Chiusa parentesi.

Per la stesura dei metodi accessor, la cosa più semplice è lasciare fare a XCode. La mia versione (la 2.2, ma la cosa è presente da un certo numero di versioni in qua) fornisce due script di supporto che svolgono tutto il lavoro. Basta selezionare le variabili d'istanza di una classe. Poi, accedendo al menu `Applescript` si seleziona la voce `Place Accessor Decls/Defs on Clipboard` del menu `Code` a seconda se si vogliono le dichiarazioni o le definizioni dei metodi. Ci si piazza poi nel posto giusto, e si Incolla il testo nella Clipboard. Nulla di più facile e completo. In effetti, in questa versione di XCode sono prodotti metodi accessor sofisticati anche per i vettori (ovvero, le to-many relationship), cosa che non ricordo accadesse con le versioni precedenti.

Ripredendo quindi le due classi d'esempio citate in precedenza, costruisco un nuova coppia di file .m/.h all'interno dei quali, pur non rispettando una delle regole base della programmazione object-oriented, dichiaro e definisco entrambe le classi. A parte scrivere le variabili d'istanza, lascio a XCode il compito di scrivere il resto del codice. Mi limito ad aggiungere un metodo (per ciascuna classe) che mi aiuta nelle operazioni di debugging e comprensione dei meccanismi:

```
- (NSString *) locDescr
{
        return ([ NSString stringWithFormat: @"%@ -- varLong: %10ld,
altroTesto: %@",
        [self description ], varLong, altroTesto ]);
}
```

Ricordo che il metodo description è fornito di serie con `NSObject` (e quindi a tutti i discendenti) come produttore di una stringa che descrive il contenuto dell'oggetto: il suo valore, se l'oggetto è semplice (un numero, una stringa), o qualcosa di più criptico negli altri casi.

ALLA PROVA

Come primo esempio di applicazione, faccio una cosa piuttosto pedestre:

```
        primaClasse        * obj1 ;

        NSLog( @"\n\t\t\t\t--- test %2d---", testDaEseguire );
        // costruisco l'oggetto
```

```
        obj1 = [[ primaClasse alloc] init ];
        // gli assegno le variabili d'istanza per nome
         [ obj1 setValue: [ NSNumber numberWithLong: 12] forKey: @"varLong"
];
        [ obj1 setValue: @"testo1" forKey: @"altroTesto" ];
        // vediamo come e' venuto fuori
        NSLog( [ obj1 locDescr ] );

        // accedo alle variabili per nome
        NSLog( @"varLong: %@", [ obj1 valueForKey: @"varLong" ]);
        NSLog( @"testo : %@", [ obj1 valueForKey: @"altroTesto" ]);
        NSLog( @"dict : %@", [obj1 dictionaryWithValuesForKeys:
                                [   NSArray   arrayWithObjects:   @"varLong",
@"altroTesto"]]);
```

Ho costruito un oggetto, ed ho assegnato le variabili d'istanza non tanto con metodi accessor, ma indirettamente con il paradigma KVC. L'unica cosa men che banale è l'ultima istruzione, in cui raccolgo le variabili indicate in un dizionario.

```
2006-04-29 17:02:15.252 test[943]
--- test 1---
2006-04-29 17:02:15.258 test[943] <primaclasse: 0x5033d0> -- varLong: 12,
altroTesto: testo1
2006-04-29 17:02:15.259 test[943] varLong: 12
2006-04-29 17:02:15.260 test[943] testo : testo1
2006-04-29 17:02:15.263 test[943] dict : {altroTesto = testo1; varLong =
12; }
```

Nel secondo esempio, ho provato a vedere cosa succede utilizzando nomi inesistenti. La prima realizzazione muore in maniera spettacolare non appena se ne provi l'esecuzione:

```
        primaClasse        * obj1 ;

        NSLog( @"\n\t\t\t\t--- test %2d---", testDaEseguire );
        obj1 = [[ primaClasse alloc] init ];
        [ obj1 setValue: @"testo1" forKey: @"chiavenonesistente" ];
          NSLog( @"varLong: %@", [ obj1 valueForKey: @"chiavenonesistente"
]);
```

Ciò che accade è la generazione di una eccezione. Poiché nel mio codice di esempio non esiste alcun agente in grado di trattarla, se eseguo in XCode mi si apre il debugger, da terminale si producono scritte inquietanti:

```
host189-191:~/Documents/Macocoa/devTest/mc072/build/Debug djzero $ ./test
2006-04-29 16:26:52.290 test[695]
--- test 2---
2006-04-29    16:26:52.341    test[695]    ***    Uncaught    exception:
<nsunknownkeyexception> [<primaclasse 0x5033b0> setValue:forUndefinedKey:]:
this   class   is   not   key   value   coding-compliant   for   the   key
chiavenonesistente.
Trace/BPT trap
```

In effetti, l'ambiente run-time riconosce il maldestro tentativo ed impedisce operazioni ulteriori.

La seconda versione di questo esempio prova allora a gestire l'eccezione. Riporto il codice così come mi è venuto, riservandomi di approfondire l'interessante argomento

della gestione eccezioni:

```
        primaClasse        * obj1 ;

        NS_DURING        // sezione standard

        NSLog( @"\n\t\t\t\t--- test %2d---", testDaEseguire );
        // costruisco l'oggetto
        obj1 = [[ primaClasse alloc] init ];
        // provo ad impostare un valore
        [ obj1 setValue: @"testo1" forKey: @"chiavenonesistente" ];
         // qui tanto non ci arrivo, vista l'eccezione che si e' scatenata
sopra
          NSLog( @"varLong: %@", [ obj1 valueForKey: @"chiavenonesistente"
]);

        NS_HANDLER        // handler delle eccezioni
        // localException sembra essere una variabile definita all'interno
           // del meccanismo delle eccezioni che contiene il motivo dei
problemi
        NSLog( @"ECCEZIONE: %@", localException );

        NS_ENDHANDLER    // chiusa gestioni
```

In questo modo, l'esecuzione dell'applicazione non causa traumi; l'impossibilità di lavorare con la variabile indicata da un nome inesistente è riconosciuta facilmente e produce sostanzialmente lo stesso testo visto sopra, ma in maniera più controllata:

```
2006-04-29 17:02:39.714 test[944]
--- test 2---
2006-04-29  17:02:39.720  test[944]  ECCEZIONE:  [<primaclasse  0x503500>
setValue:forUndefinedKey:]: this class is not key value coding-compliant
for the key chiavenonesistente.
```

Provo quindi a realizzare (per la classe `secondaClasse`) il metodo del protocollo che gestisce proprio questi comportamenti strani. Scrivo il metodo seguente (niente di più che un metodo civetta per vedere che è invocato, una volta presente):

```
- ( id) valueForUndefinedKey: (NSString *) key
{
    return ( @"xxx" ) ;
}
```

Così facendo, il pezzo di codice seguente non produce disastri:

```
        secondaClasse        * obj1 ;

        NS_DURING        // sezione standard

        NSLog( @"\n\t\t\t\t--- test %2d---", testDaEseguire );
        // costruisco l'oggetto
        obj1 = [[ secondaClasse alloc] init ];
        // provo a leggere una variabile inesistente
          NSLog( @"varLong: %@", [ obj1 valueForKey: @"chiavenonesistente"
]);
        // ci riesco, ma il valore non ha senso...

        NS_HANDLER        // handler delle eccezioni
```

```
        NSLog( @"ECCEZIONE: %@", localException );

        NS_ENDHANDLER    // chiusa gestioni
```

ma solo una risposta di default non impegnativa:

```
2006-04-29 17:03:42.004 test[945]
--- test 3---
2006-04-29 17:03:42.010 test[945] varLong: xxx
```

OPERATORI SU VETTORI ED INSIEMI

Una cosa interessante che si può fare con i meccanismi KVC è di applicare un dato operatore a tutti gli oggetti che fanno parte di un vettore. Il costrutto sotto esame appare complicato a prima vista:

```
keyPathToArray.@operator.keyPathToProperty
```

La stringa `keyPathToArray` è un percorso chiave che specifica un elemento di tipo NSArray (o assimilabile). Se l'oggetto cui si applica è già un vettore, si omette la prima parte. La stringa `keyPathToProperty` specifica un percorso chiave che individua una determinata proprietà (ma ci deve essere da qualche parte una specifica che attraversa un vettore). Infine, la stringa `operator` determina quale operazione deve essere effettuata sui valori della proprietà individuata. Tra le varie possibilità sono presenti quelle tipiche: il valore massimo, il valore minimo, la media, il numero di elementi.

Con qualche esempio si capisce meglio. Partiamo dall'uso semplice di `valueForKeyPath`, tornando alle classi di esempio viste in precedenza.

```
        primaClasse            * obj1 ;
        secondaClasse        * obj2 ;

        NSLog( @"\n\t\t\t\t--- test %2d---", testDaEseguire );
        // costruisco l'oggetto
        obj1 = [[ primaClasse alloc] init ];
        // gli assegno le variabili d'istanza per nome
         [ obj1 setValue: [ NSNumber numberWithLong: 111] forKey: @"varLong"
];
        [ obj1 setValue: @"testo1" forKey: @"altroTesto" ];

        // costruisco un altro oggetto
        obj2 = [[ secondaClasse alloc] init ];
              [ obj2 setValue: [ NSNumber numberWithLong: 222] forKey:
@"varLong2" ];
        [ obj2 setValue: @"testo2" forKey: @"testo" ];
        // to-one relationship
        [ obj2 setValue: obj1 forKey: @"objinterno" ];
        // vediamo cosa e' saltato fuori
        NSLog( [ obj2 locDescr ] );
          // accesso alle variabili di obj1 attraverso obj2 e la specifica
del path
                      NSLog( @"varLong: %@", [ obj2 valueForKeyPath:
@"objinterno.varLong" ]);
          [ obj2 setValue: @"testoxxx" forKeyPath: @"objinterno.altroTesto"
```

```
];
                        NSLog(   @"testo1:   %@",   [   obj2   valueForKeyPath:
@"objinterno.altroTesto" ]);
```

costruisco il primo oggetto, e lo inizializzo tramite KVC. Passo al secondo oggetto, e lo inizializzo sempre con KVC. Tra i valori che assegno, c'è una una to-one relationship, che coinvolge i due oggetti. A questo punto, posso leggere ed impostare i valori delle variabili interne sempre per nome, sfruttando il percorso completo.

```
2006-04-29 17:03:55.575 test[946]
--- test 4---
2006-04-29 17:03:55.581 test[946] <secondaclasse: 0x504ee0> -- varLong2:
222, testo: testo2 arrayDim: 0
2006-04-29 17:03:55.584 test[946] varLong: 111
2006-04-29 17:03:55.585 test[946] testo1: testoxxx
```

Fin qui, è andato tutto bene. Mi complico la vita con la to-many relationship. Per prima cosa un segmento di codice che costruisce il tutto:

```
        primaClasse            * obj1 ;
        secondaClasse        * obj2 ;
        NSMutableArray        * locArr ;
        unsigned short        ii ;
        id        val ;

        NSLog( @"\n\t\t\t\t\t\t\t\t--- test %2d---", testDaEseguire );
        // costruisco un oggetto e gli assegno proprieta'
        obj2 = [[ secondaClasse alloc] init ];
              [ obj2 setValue: [ NSNumber numberWithLong: 222] forKey:
@"varLong2" ];
        [ obj2 setValue: @"testo2" forKey: @"testo" ];
        // costruisco un vettore di oggetti di tipo primaClasse
        locArr = [ NSMutableArray arrayWithCapacity: 5 ];
        for ( ii = 0 ; ii < 5 ; ii ++ )
        {
            // costruisco l'oggetto
            obj1 = [[ primaClasse alloc] init ];
            // gli assegno le variabili d'istanza per nome
             [ obj1 setValue: [ NSNumber numberWithLong: (100 + ii)] forKey:
@"varLong" ];
            [ locArr addObject: obj1 ];
        }
        // to-many relationship
        [ obj2 setValue: locArr forKey: @"vettore" ];
        NSLog( [ obj2 locDescr ] );
```

Adesso, ho un vettore di oggetti `primaClasse`, all'interno di ciascuno dei quali la variabile `varLong` ha un valore differente (i cinque valori 100, 101, 102, 103 e 104). Allora:

```
val = [ locArr valueForKeyPath:@"@sum.varLong" ];
```

Con questa istruzione si calcola la somma di tutti i valori di `varLong` che sono presenti negli oggetti di tipo `primaClasse` raccolti nella variabile `locArray`. Il risultato dovrebbe essere 510.

Invece con questa istruzione si calcola il numero di elementi del vettore `vettore`.:

```
val = [ obj2 valueForKeyPath:@"vettore.@count" ];
```

Oppure si può calcolare la media dei cinque valori presenti:

```
val = [ obj2 valueForKeyPath:@"vettore.@avg.varLong" ];
```

Per finire, l'istruzione

```
val = [ obj2 valueForKeyPath:@"vettore.@unionOfObjects.varLong" ];
```

produce un array di elementi; questo array si ottiene unendo tutti gli elementi `varLong` degli oggetti presenti nel vettore `vettore` dell'oggetto `obj2`. In effetti ciò che viene prototto sul terminale mi da ragione:

```
2006-04-29 17:04:12.791 test[947]
--- test 5---
2006-04-29 17:04:12.799 test[947] <secondaclasse: 0x503fc0> -- varLong2:
222, testo: testo2 arrayDim: 5
2006-04-29 17:04:12.804 test[947] somma: 510
2006-04-29 17:04:12.805 test[947] quanti: 5
2006-04-29 17:04:12.807 test[947] media: 102
2006-04-29 17:04:12.808 test[947] unione: (100, 101, 102, 103, 104)
```

Capitolo 073 - Universal binary

Trasformo un po' d'applicazioni in Universal Binary

Sorgenti: documentazione Apple

Prima stesura: 8 luglio 2006

BINARI UNIVERSALI

Come dovrebbe essere noto, Apple ha deciso di equpaggiare i propri computer con processore Intel piuttosto che continuare ad affidarsi all'architettura PowerPC (di Motorola/Freescale/IBM). La transizione pone qualche problema agli sviluppatori, a causa della differente architettura dei processori. Come al solito, tuttavia, Apple semplifica la transizione, sia per l'utente comune attraverso l'interprete Rosetta, sia per lo sviluppatore, fornendo gli strumenti necessari.

È invece meno noto che sono da poco possessore di un iMac intel (con XCode 2.3); è quindi giocoforza affrontare la questione della migrazione, ovvero la creazione di applicazioni Universal Binary (UB).

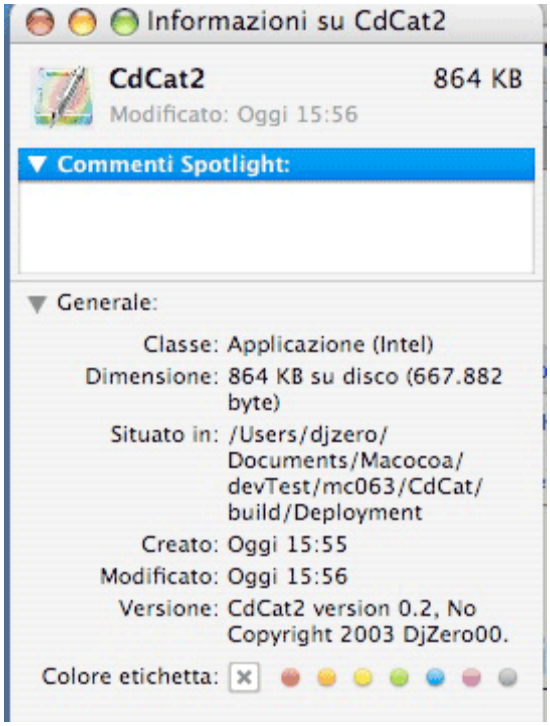
Parto dalla documentazione Apple, ovvero `Universal Binary Programming Guidelines, second edition`. Le assunzioni che il documento prevede siano soddisfatte per i programmi da rendere UB sono molto semplici: devono essere applicazioni sviluppate per Macintosh con Carbon, Cocoa o Java. Rientrano in tutto le applicazioni che mi appresto a rendere UB, ovvero, `CdCat` (rilascio del capitolo 63), lo `screensaver` (capitolo 64), `cocoaurl` (capitolo 65) e `YahFortune` (capitolo 70). Ognuna di queste applicazioni ha delle peculiarità che permettono di testare la conversione in maniera, credo, piuttosto completa.

CONVERTO IL CATALOGO

Comincio con CdCat. Il punto di partenza è l'applicazione originale, che mi informa essere una normale applicazione PCC.

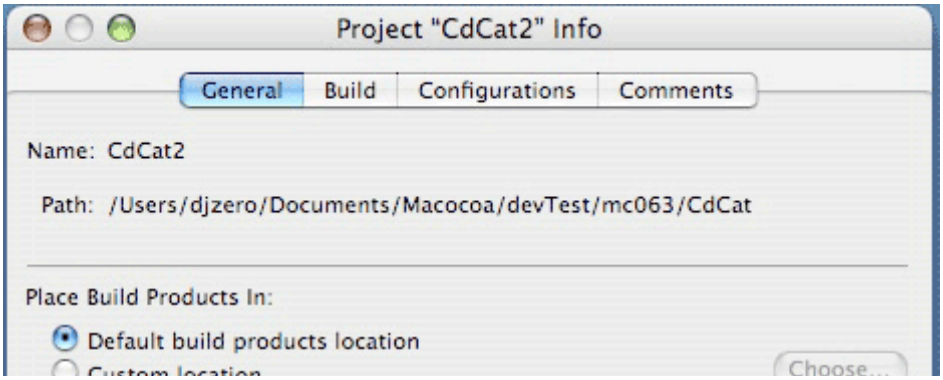
Va detto che funziona tranquillamente così com'è anche sullo iMac intel, senza colpo ferire. Tuttavia, voglio rendere il tutto UB. Importo il progetto e, dopo qualche traversia di conversione del file del progetto, ricompilo tutto senza cambiare nulla.

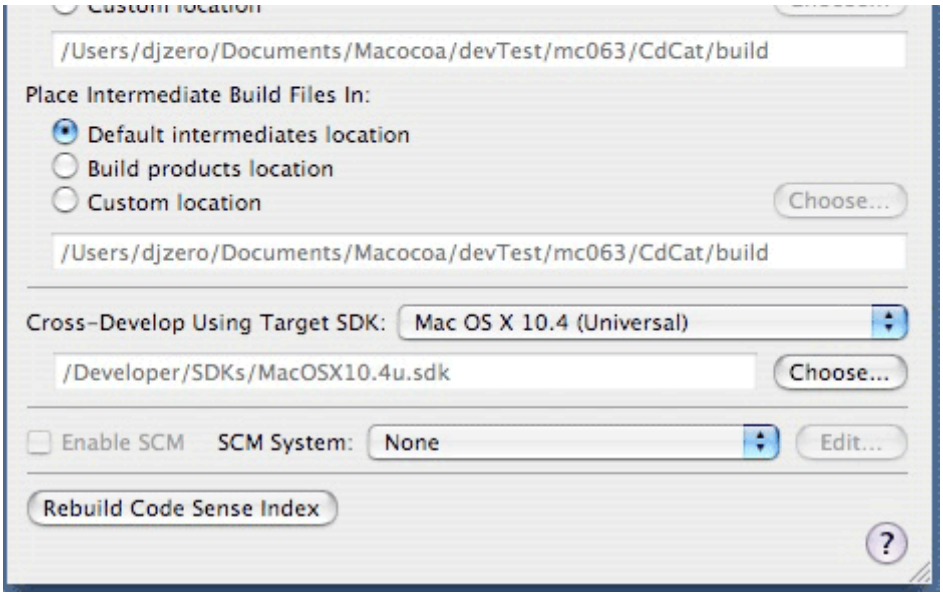
Ma il prodotto non è UB, è semplicemente intel.



Si vede che non è così banale. Leggo il documento Apple, e scopro di dover fare qualcosa. Mi figuro che le stesse operazioni funzionino anche su macchine PowerPC, purché, ovviamente, la configurazione sistema (10.4) e Xcode (2.3) siano soddisfatte.

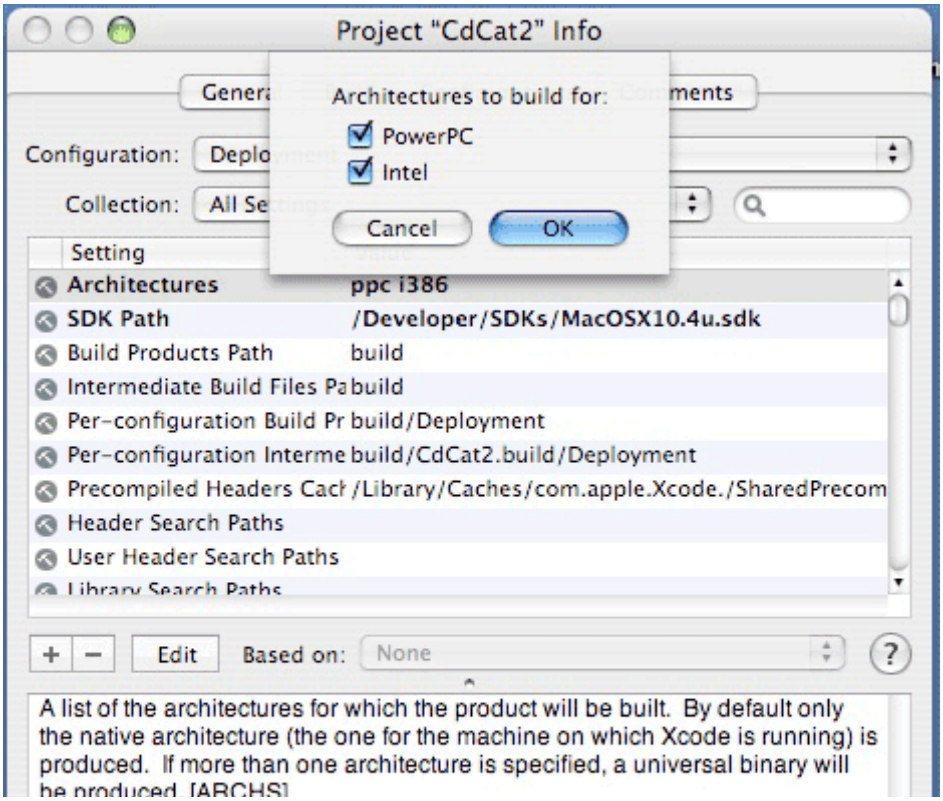
In primo luogo, devo essere sicuro di utilizzare la versione 4.0 del compilatore GCC. Essendo di default così in Xcode 2.3, devo far nulla. Il progetto però va configurato per lavorare in `Cross-development`, cioè per produrre del codice non necessariamente per la macchina sulla quale sto lavorando. Si cerca quindi il pannello delle informazioni del progetto, e nel tab `General` cerco la sezione apposta.

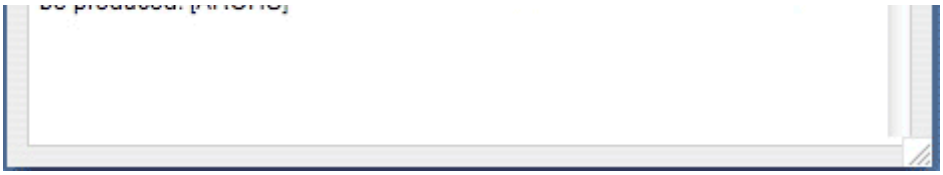




Seleziono il cross development per `Mac OS X 10.4 (Universal)`, in modo da essere sicuro. XCode mi minaccia con un messaggio di avviso, ma la documentazione mi dice di andare avanti tranquillo.

Il passo successivo è di cambiare le impostazioni dell'architettura. Rimango nello stesso pannello ma cambio tab, e vado in `Build`. Qui devo effettuare i cambiamenti sia per la configurazione `Development` (ma anche no, dipende se voglio generare codice per PPC anche in sede di sviluppo) sia e soprattutto per la configurazione `Deployment`. Ricordo che la configurazione Development utilizza alcuni accorgimenti (ZeroLink, Fix and Continue, eccetera) che permettono di velocizzare le fasi di sviluppo. La configurazione Deployment invece costruisce l'applicazione per la distribuzione, elimina gli accorgimenti visti sopra e cerca di ottimizzare il codice.





Seleziono insomma la riga `Architectures` e uso il tasto `Edit` poco sotto. Mi si apre un piccolo pannellino dove spuntare le architetture di destinazione del codice prodotto. Di default è acceso (per me che lavoro su intel) solo Intel, spunto anche PPC.

A quanto pare, ho finito. Vedo se è vero. Ricompilo tutto con la configurazione di Deployment, ed ottengo un eseguibile. Questa volta le informazioni sul file dicono che è UB.

Le dimensioni sono aumentate, come dovrebbe essere giusto (bisognerà pur conservare da qualche parte il codice da eseguire su intel e quello per PPC), ma non sono raddoppiate come ingenuamente si potrebbe pensare. Guardando all'interno dell'applicazione (da Finder, menu contestuale, Mostra Contenuto pacchetto) si nota che nessun file è duplicato, nemmeno l'eseguibile vero e proprio. Il codice per Intel dovrebbe essere lì dentro: in effetti le dimensioni di questo file sono quasi raddoppiate.

L'applicazione funziona normalmente, leggo scrivo e stampo. Fin qui, tutto bene.

La prova fondamentale tuttavia è di pigliare questa applicazione, ed i file prodotti, e sperimentare su un PPC. Sposto tutto sul mio vecchio powerbook G3 e vedo che accade. Tutto funziona. Molto bene.

E TUTTI GLI ALTRI

Andiamo avanti. Salvaschermo. Rifaccio le stesse operazioni (cross-development e selezione dell'architettura). Provo su intel, funziona. Provo su PPC, funziona. Bene. CocoaUrl. Funziona su entrambe le piattorme.

YahFortune. Qui ci sono diversi target. Li compilo tutti, e tutti sembrano funzionare correttamente (tuttavia, ho il sospetto, di fronte a certi comportamenti bizzarri che

compaiono di tanto in tanto, ci sia qualche problema, più dell'applicazione che della trasformazione in UB...).

Insomma, la transizione in UB è parsa una passeggiata.

PUNTI NOTEVOLI

Ma quali sono le cose che potrebbero andare storte (e perché qui non sono andate storte)?

Essenzialmente, mi pare di capire che ci possano essere due tipi di problemi: fare di conto e l'ordine dei byte. PowerPC ed Intel sono due architetture piuttosto differenti (CISC vs RISC, per cominciare), che gestiscono le istruzioni, i dati e la memoria in maniera differente.

Le istruzioni in PPC sono sempre lunghe 4 byte, sono poche, semplici e ben ottimizzate. Le istruzioni su Intel sono di dimensioni variabile, sono tante, e qualche volta fanno lunghi giri per essere eseguite compiutamente (si fa per dire). Ciò che su PPC veniva raggiunto con eleganza ed efficienza, Intel lo fa di mera forza bruta. Lungi tuttavia dall'attribuire coloritura etica e morale sulla faccenda, passiamo ai due punti importanti.

La prima questione sono i numeri; benché rispondenti in entrambi i casi a vari standard, la trattazione differisce in qualche minuzia. Ad esempio, la divisione per zero (operazione matematicamente non lecita) produce il valore zero su PPC, senza troppi patemi. Su Intel, è una mezza tragedia, causando eccezione e blocco del programma (e qui mi pare che Intel abbia più ragione...). Pertanto, se il programma da trasformare utilizza estesamente calcoli in cui c'è il rischio di una divisione per zero, le strategie per evitarla sono teoricamente differenti nei due casi. In pratica, un buon programmatore ha già affrontato e risolto il problema evitando in generale la divisione per zero. Ugualmente, potrebbero esserci problemi quando si fanno confronti con i numeri floating point (che, a parità di valori interessati, potrebbero dare risultati diversi sulle due architetture), ma anche qui un buon programmatore ha già risolto il problema in modo indipendente dalla piattaforma. Lungi dall'essere una sagace programmatore, non ho riscontrato nei miei casi questo tipo di problemi perché i numeri hanno avuto poco impatto nelle applicazioni fin qui sviluppate.

L'ordine dei byte è invece il problema più grosso. Tutto nasce da come i dati composti da più di un byte sono conservati in memoria. Si può utilizzare la strategia `big-endian`, in cui i byte più significativi vengono prima di quelli meno significativi (come succede nel PPC), oppure la strategia `little-endian`, opposta, dove i byte meno significativi sono messi davanti (ovviamente, Intel). Non mi pare di ricordare ragioni tecniche per cui un metodo di allineamento sia migliore dell'altro, ragione per cui il mondo informatico si è diviso (a volte con toni acerrimi) tra big-endiani e little-endiani. Per inciso, il nome deriva da un episodio del libro di Jonathan Swift "I viaggi di Gulliver", in cui i lillipuzziani erano in guerra con un'altra popolazione (di pari dimensioni ma di nome differente). Le due popolazioni infatti erano in conflitto per il modo con cui utilizzavano le uova a scopi alimentari, con gli uni che le aprivano dalla parte piccola (i little-endiani), e gli altri che li aprivano dalla parte più grossa (i big-endiani). Queste profonde ed importanti differenze nella visione delle cose del

mondo li ha portati ad essere nemici e combattere per lunghi anni (e quando qualcuno mi cita i viaggi di gulliver come libro per bambini, rido della sua pochezza culturale e gli faccio notare come Swift sia uno scrittore che scandagliava l'animo umano come pochi).

Tornando ai nostri byte, l'importanza dell'ordine si ha ogni volta che si organizzano in memoria strutture dati più grosse di un byte (sempre) ed ogni volta che si ha a che are con flussi lineari di dati, ovvero file su disco e connessioni di rete (che mi pare di ricordare siano quasi sempre big-endian). Per cui, ogni volta che si ci si avvicina al cuore del trasferimento di dati, occorre prestare attenzione a come si gestiscono questi byte. Confesso che all'inizio pensavo di dovermi sorbirmi tutta questa faccenda, e di dover esaminare il codice alla ricerca di sezioni dove i dati erano gestiti. Invece, dal momento che utilizzo Cocoa e non mi sono avventurato nella gestione diretta di flussi di dati, ma ho lasciato fare tutto a Objective C e le classi come NSCoding, devo fare proprio nulla e lasciare tutto intonso. Il che è un risultato non da poco.

Capitolo 074 - Debugging

Introduco brevemente il meraviglioso mondo del Debugger.

Sorgenti: documentazione Apple e non solo.

Prima stesura: 3 settembre 2006.

GDB e XCode

Un debugger è una applicazione di ausilio a chi sviluppa del software; in effetti, una dell occupazioni che impiega maggiormente il tempo di uno sviluppatore è `debuggare` (verbo bruttissimo che utilizzo in mancanza di un equivalente e significativo verbo italiano), ovvero trovare gli errori logici di esecuzione dell'applicazione. Un debugger serve a mostrare cosa sta succedendo all'interno di un'altra applicazione mentre è in esecuzione, oppure per diagnosticare cosa sia successo negli istanti precedenti ad un crash. Un debugger è quindi una applicazione insostituibile per ogni sviluppatore, che può semplificare il lavoro esaminando una istruzione alla volta. Molti sono usi infarcire il proprio codice con delle scritture su console (che so, attraverso printf o NSLog) per capire cosa sta succedendo in un dato momento dell'esecuzione. Tutto ciò è svolto in maniera egregia e meno invasiva dal debugger, senza problemi.

I primi debugger erano semplicemente un metodo per eseguire le istruzioni dell'applicazione passo-passo, istruzioni che all'inizio erano in linguaggio macchina. In pratica, una sorta di interprete del linguaggio macchina. Il salto successivo, e drastico, è stato quando i debugger hanno cominciato a lavorare direttamente sul codice sorgente, in formato simbolico.

A cosa serve un debugger

Le operazioni principali svolte da un debugger sono: il lancio in esecuzione di una applicazione e l'esecuzione un passo alla volta delle istruzioni che lo compongono; il controllo di esecuzione dell'applicazione, con la possibilità di arrestare l'esecuzione della stessa in occasione di particolari condizioni; l'esame dei valori delle variabili, della memoria e di quant'altro appartenente all'applicazione; la modifica (!) al volo del valore di variabili, del contenuto della memoria e, in tali casi, perfino l'esecuzione di codice diverso da quello dell'applicazione.

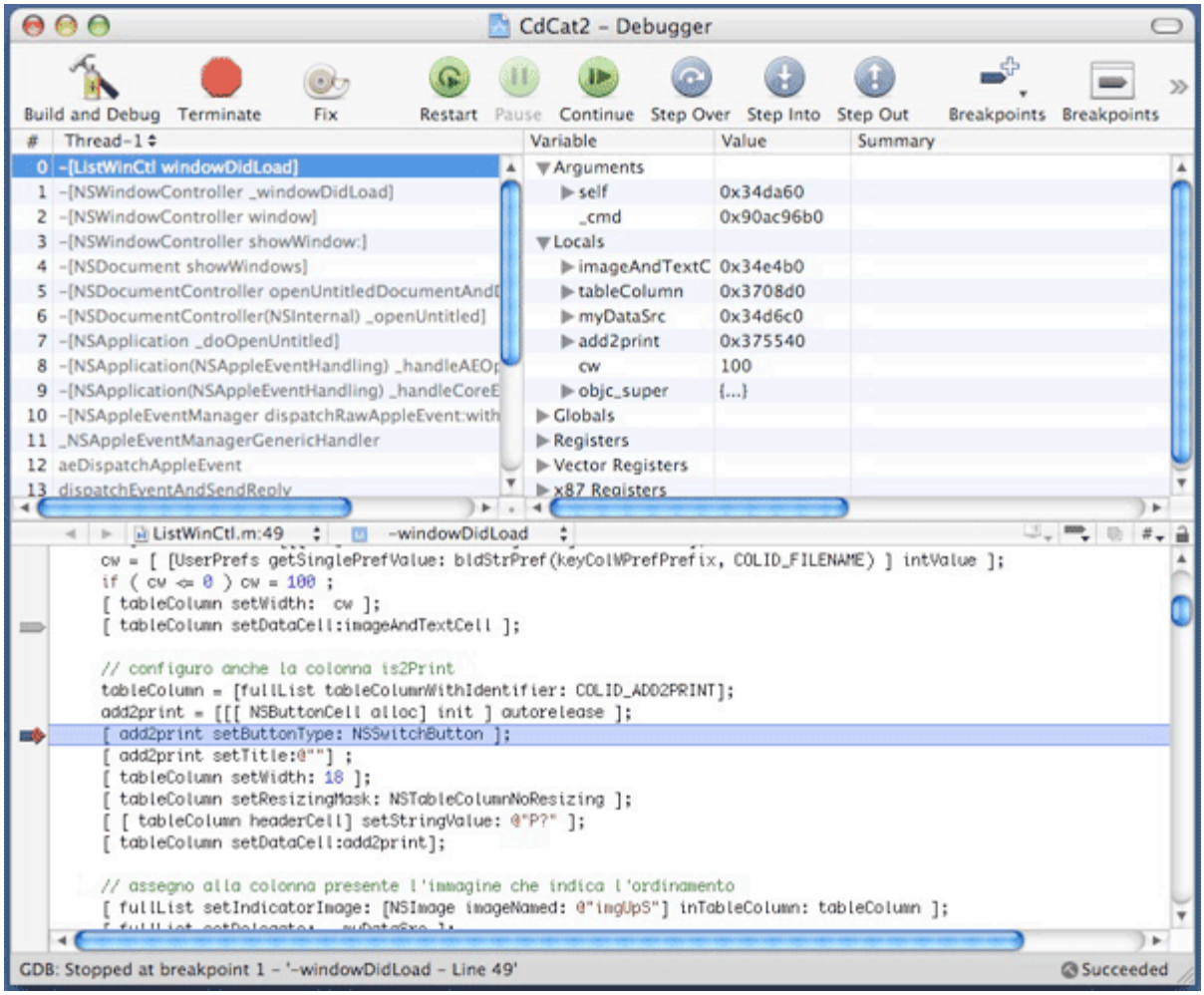
All'interno di XCode è presente il debugger `GDB` (il debugger open-source del progetto GNU). Normalmente, GDB è un debugger che funziona da linea di comando; tuttavia, Apple è così gentile da averlo dotato di una comoda interfaccia grafica (come spesso accade agli strumenti di sviluppo di XCode). Inoltre, lo integra all'interno dell'ambiente di sviluppo in modo da poter tranquillamente dimenticare molte delle peculiarità della linea di comando. Da notare che GDB è in grado di debuggare più o meno tutti i principali linguaggi, con opportune estensioni per ciascuno di essi. Ad esempio, per Objective C, c'è il supporto per gli oggetti e i messaggi verso gli oggetti.

Per poter utilizzare pienamente GDB, occorre in primo luogo che il progetto si compili

compiutamente senza errori. Il debugger non serve a trovare gli errori di compilazione, ma quelli di esecuzione. In secondo luogo, occorre che il compilatore produca, durante il processo di compilazione, una serie di informazioni aggiuntive. Questo avviene automaticamente quando il progetto è compilato in modalità `Debug` o `Development`. Non avviene (e quindi una sessione di debugging non può aver luogo) se il progetto è compilato in `Release` o `Deployment`. La scelta tra questi due modi avviene tramite il menu `Project`, con la `Active Build Configuration`. C'è poi un intero menu, `Debug` appunto, dedicato alle funzioni che si possono svolgere all'interno di GDB. La prima e fondamentale è appunto di attivare il debugger e contestualmente lanciare in esecuzione l'applicazione. A parte il menu, è possibile inserire nella toolbar di XCode una icona apposita, cosa che ho subito fatto, in quanto il Debugger è il secondo strumento più utilizzato dal processo di sviluppo (il primo, ovviamente, il compilatore).

Lanciando l'applicazione, non sembra succedere nulla di diverso da quanto accade lanciando in normale esecuzione l'applicazione, a parte l'apertura di una nuova finestra propria del debugger. Tuttavia, la finestra, divisa in tre sezioni, non presenta informazioni utili. Questo perché non è presente alcun breakpoint.

NON MI ROMPETE



Un `breakpoint` (letteralmente: punto di rottura) è il concetto principe del debugger. Si tratta di un punto dove si vuole che il programma in esecuzione si blocchi e sospenda

ogni attività; normalmente, un breakpoint coincide con una linea del programma sorgente, opportunamente indicata. L'indicazione di un breakpoint avviene portandosi sulla riga che si intende evidenziare, e utilizzare l'apposita voce di menu (`Debug`, e poi `Add breakpoint...`). Dovrebbe comparire, sulla barra sinistra della finestra del codice, una specie di linguetta scura, indicante appunto la presenza di un breakpoint attivo. Alternativamente, il breakpoint si può assegnare facendo clic direttamente un quella zona sulla sinistra della finestra (chiamata `Gutter`; se non è presente, bisogna attivarla esplicitamente nelle preferenze dell'editor di XCode). Facendo nuovamente clic, la linguetta diventa grigia: significa che il breakpoint è presente, ma non è attivo. L'assegnazione di un breakpoint può avvenire prima di lanciare il debugger, ma anche durante. L'operazione è la stessa, ed il breakpoint ha subito effetto.

GDB esegue l'applicazione normalmente, tranne quando incontra un breakpoint. In tal caso, sospende l'esecuzione dell'applicazione, giusto prima dell'esecuzione dell'istruzione indicata dal breakpoint, e ferma tutto (dove per tutto si intende anche eventuali altri thread o task o quant'altro, anche se questo è un comportamento che dipende da debugger a debugger).

Ed è proprio quando siamo in questo momento sospeso nel tempo di esecuzione dell'applicazione che la potenza del debugger si esplica nella sua pienezza. In primo luogo, si popolano i tre pannelli della finestra del debugger. A meno di configurazione esotiche, nel pannello in basso si trova il codice dell'applicazione, con la riga sospesa bene evidenziata. In alto a destra, sono presentate la variabili in quel momento accessibili all'applicazione (e quindi, le variabili locali, gli argomenti del metodo, e le globali ed altro che non mi interessa). Tra gli argomenti del metodo c'è anche la variabile self, che punta precisamente all'oggetto il cui metodo è attualmente sospeso. Il pannello in alto a sinistra mostra, per i vari thread presenti, qual è la situazione a livello di chiamate, ovvero la catena di metodi e funzioni che hanno portato alla situazione corrente.

CAMBIARE LE VARIABILI

In questa situazione sospesa, si può controllare il valore delle variabili, e verificare se sono consistenti con quello che ci si aspetta a qual punto dell'esecuzione. Ora, molte variabili sono normali variabili (interi, float, eccetera), ma molte altre sono oggetti, o meglio, puntatori ad oggetti. è per questo che nella seconda colonna del pannello delle variabili ci sono valori molto strani (che cominciano con 0x): si tratta degli indirizzi di memoria dove si trovano effettivamente gli oggetti. Queste variabili hanno però un triangolino davanti al nome, che permette di espandere il contenuto dell'oggetto. Ad esempio, espandendo la variabile `self`, si trovano (oltre a una serie di altre cose sconosciute, evidentemente ereditate dalle varie superclassi) le variabili d'istanza dell'oggetto. Inoltre, per alcuni degli oggetti, quelli di tipo più "semplice" (come `NSString`) la colonna `Summary` ne presenta il valore effettivo o comunque una indicazione un po' più interessante del contenuto. Il primo elemento di ogni classe espansa è il campo `isa`, che identifica il tipo di oggetto (la sua classe). è così possibile verificare durante l'esecuzione la classe di un oggetto altrimenti identificato col termine generico id.

Selezionando una variabile, si può attivare un menu contestuale (che poi è il menu `Variabile View`, sempre all'interno di `Debug`), dove ci sono un paio di voci

interessanti di cui voglio parlare.

La prima è la `Print Description to Console`, che scrive sulla console (che è un'altra finestra che si può aprire, e corrisponde in pratica al terminale dove è partito GDB) una descrizione dell'oggetto. In pratica, è l'applicazione del metodo

```
-(NSString*) debugDescription
```

all'oggetto selezionato. Ora, tutti gli oggetti ereditano da `NSObject` questo metodo, che aggiunge qualche informazione in più sull'oggetto e che i programmatori accorti possono sovrascrivere per le proprie classi, in modo che all'interno del debugger si abbia una rappresentazione più leggibile del loro oggetto. Ancora per esempio, per gli array, i dictionary e le collezioni in genere, il metodo `description` presenta il contenuto della collezione. Se si intende sovrascrivere `debugDescription`, è bene farlo a partire dal metodo

```
- (NSString) description
```

dall'uso molto simile. Molti oggetti realizzano `debugDescription` semplicemente chiamando `description`. Il metodo `debugDescription` dovrebbe essere quindi una versione più ricca di `description`, a cui si dovrebbe rifare per le informazioni di base.

La seconda voce è `Edit Value`, che permette appunto di modificare il valore della variabile stessa (a dire il vero questa caratteristica si ottiene anche facendo doppio clic sopra il valore). Sebbene cambiare l'indirizzo di memoria dell'oggetto non sia un'operazione consigliata all'inesperto, modificare il valore di una variabile (intera, o stringa) è comodo per recuperare situazioni compromesse da un errore, per scatenare un errore e vedere come si comporta l'applicazione, cose del genere.

PASSO DOPO PASSO

A questo punto, conviene esaminare alcuni elementi della toolbar (che corrispondono a svariate voci del menu `Debug`). Mi interessano i tre pulsanti verdi ed i tre pulsanti il cui nome comincia per Step.

I pulsanti verdi regolano l'esecuzione generale dell'applicazione. Se l'applicazione è ferma (ad esempio, perché è appena scattato un breakpoint), il pulsante `Pause` è disabilitato, mentre sono attivi `Restart` e `Continue`. Come dovrebbe essere ovvio, il pulsante Restart riavvia completamente da zero l'applicazione; il pulsante Continue invece fa proseguire l'applicazione, che potrà fermarsi al prossimo breakpoint (sempre se ne esiste uno). Se invece l'applicazione sta funzionando normalmente, è attivo il solo pulsante Pause, che causa l'interruzione immediata dell'applicazione. Generalmente, la fermata avviene in mezzo al codice, per cui non si vedrà (a meno di fortune sfacciate) il codice Objective C, ma dell'incomprensibile linguaggio macchina.

Torno ora ai tre pulsanti Step: `Step Over`, `Step Into` e `Step Out`. Il loro effetto è di eseguire una (o più istruzioni) e fermarsi prima di continuare. In particolare, Step Over (salta sopra) esegue una istruzione, esattamente quella indicata dalla riga corrente. Se l'istruzione è la chiamata di un metodo, di una funzione, o anche più di una, insomma,

si tratta di una istruzione complessa, sono eseguite tutte le operazioni indicate e l'esecuzione si arresta alla successiva riga di codice nel metodo/funzione corrente. Diverso è il comportamento del pulsante Step Into (salta dentro), che esegue una ed una sola istruzione: se quindi la riga contiene una chiamata ad una funzione o metodo, si salta dentro la funzione o metodo chiamato. Infine, Step Out (salta fuori) esegue tutte le istruzioni del metodo o funzione corrente, e si ferma all'istruzione successiva, ovvero all'interno della funzione o metodo che ha chiamato il metodo corrente.

UNA SESSIONE DI DEBUG

La combinazione di questi tre meccanismi di step, assieme ai breakpoint, fornisce uno strumento fondamentale per indagare il comportamento del codice. Faccio un esempio di una procedura di debug, piuttosto generica, che mostra come utilizzare questi strumenti.

Supponiamo di essere riusciti a circoscrivere un problema all'interno di un metodo `mioMetodo`. Allora, per debuggare `mioMetodo`, si mette un breakpoint alla prima istruzione. Si lancia in esecuzione l'applicazione, si fa qualcosa e ad un certo punto il debugger interrompe le operazioni in corrispondenza del breakpoint. La prima cosa da fare è verificare se i dati sono consistenti, ovvero se le variabili di istanza hanno dei valori ragionevoli, se la variabile self è quella che ci aspettiamo, se gli eventuali parametri della chiamata del metodo sono quelli previsti. Se qualcosa non torna, significa che non è tanto `mioMetodo` ad essere errato, quanto il fatto che le istruzioni precedenti hanno provocato confusione. Si deve quindi individuare il metodo chiamante (c'è il pannello in alto a sinistra che mostra appunto la catena delle chiamate) e debuggare quello. Se invece tutto ci pare normale, cominciamo ad eseguire le istruzioni una alla volta. La prima volta, faccio sempre Step Over (l'idea è di fidarsi in prima battuta di tutte le funzioni o metodi chiamati da `mioMetodo`). Ad ogni passo, controllo variabili e quant'altro, per vedere se succede qualcosa di strano. Se arrivo alla fine del metodo, e tutto va bene, forse ci siamo sbagliati ad imputare a `mioMetodo` la colpa dell'errore. Tuttavia, prima di assolverlo del tutto, faccio continuare l'applicazione (lasciando attivo il breakpoint) per vedere se eventuali chiamate successive di `mioMetodo` causano problemi. Se neppure questo succede, ma il problema rimane, allora assolvo `mioMetodo` e cerco altri colpevoli. Se invece `mioMetodo` mostra segni di commettere errori, la colpa può essere direttamente sua (perché alcune istruzioni sono palesemente sbagliate, perché una variabile ha un valore sbagliato, eccetera), oppure la colpa è di uno dei metodi/funzioni chiamate, diciamo `metodoInterno`. In questo caso, me ne dovrei accorgere perché l'operazione di Step Over sopra `metodoInterno` provoca qualche effetto collaterale indesiderato (valore restituito, modifica delle variabili, eccetera): la colpa è allora di `metodoInterno`. Tolgo il breakpoint su `mioMetodo` e ne metto un altro su `metodoInterno`. Faccio ripartire l'applicazione e passo a debuggare `metodoInterno`.

L'esempio sembra un po' noioso, ma dopo un po' ci si fa la mano, e si va via veloci.

È SOLO L'INIZIO

Quanto scritto in questo capitolo è solo la punta di un iceberg. GDB è un debugger molto potente, e l'interfaccia fornita da XCode non rende piena giustizia. Conviene in

primo luogo esplorare un po' tutte le possibilità fornite dall'interfaccia, e poi cominciare a studiare la linea di comando. Si scopriranno così moltissime cose, preziose ed anche indispensabili per trovare gli errori più infidi. La documentazione (sia Apple che altra) non manca. La seguente arruffata lista dovrebbe solamente darvi un'idea.

I breakpoint, che ho semplicemente presentato come un meccanismo per fermare l'esecuzione di una applicazione in un punto prefissato, sono molto di più. E' possibile associare ad ogni breakpoint una condizione, di modo che si attivi solo in particolari occasioni. Ad esempio, il breakpoint potrebbe attivarsi solamente se il valore di una variabile è nullo. Se è diverso da zero, nessun breakpoint. è possibile associare ad ogni breakpoint un'azione (che so, scrivere il valore di una variabile all'interno di un file); una cosa piuttosto comoda (ma solo in occasioni speciali) è di permettere il proseguimento automatico: l'applicazione si ferma sul breakpoint, attende diciamo tre secondi, poi continua. Se si ha l'occhio veloce, si riesce a vedere come cambiano i valori delle variabili, e si velocizza il riconoscimento die problemi.

Si può definire un `watchpoint`, ovvero arrestare l'applicazione ogni volta che una variabile viene letta o scritta.

Con l'applicazione sospesa, è possibile vedere il contenuto della memoria, vedere il contenuto delle variabili, eseguire delle istruzioni per modificare il comportamento del programma, invocare metodi e funzioni presenti, saltare istruzioni o comunque saltare in altre parti dell'applicazione.

Si può inserire un breakpoint direttamente da programma, con le istruzioni Debugger() oppure DebugStr(stringa), che fermano l'esecuzione dell'applicazione quando si incontrano nel codice.

Si può (provare a) modificare il codice, compilarlo e caricarlo nel debugger senza dover uscire dalla sessione di debug, e verificare l'effetto che fa (non ho mai provato, a dire il vero).

Capitolo 075 - DownQueue

Ricomincio a programmare riprendendo cURL.

Sorgenti: documentazione Apple, ripesco vecchi capitoli, di tutto un po'.

Prima stesura: 29 dicembre 2006.

C'ERA UNA CODA

Nel capitolo 65 avevo realizzato un wrapper Cocoa per il comando Unix cURL, adatto al trasferimento di file tramite protocollo http e ftp. Ripendo in mano l'idea e provo a fare una nuova applicazione, semplice ed inutile, ma come al solito educativo per il programmatore.

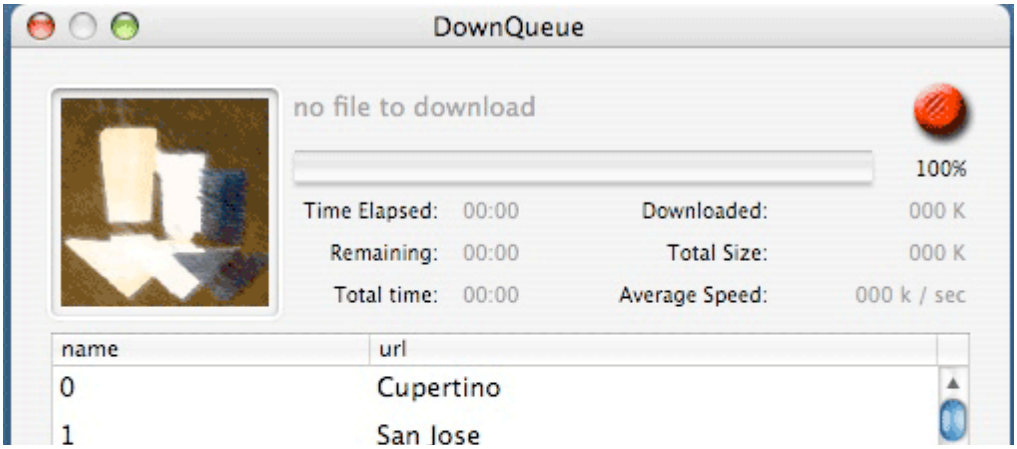
I REQUISITI

L'idea è di trasferire una lista di file, non tanto in parallelo (come ad esempio avviene con Safari o con altri programmi che funzionano da acceleratori del trasferimento; un giretto su VersionTracker o siti similari ve ne fornirà qualcuno migliore di quello che andrò a produrre), quanto in serie, uno dopo l'altro. La cosa nasce da una mia esigenza personale, che consiste nel mettere da parte una lista di file (di dimensioni generose), che poi sono trasferiti con comodo durante la notte o comunque mentre non sono presente davanti al computer (tranquilli, nulla di illegale in tutto ciò).

L'ambizione è di poter costruire un'applicazione completa, e fruibile da altre persone che non siano me stesso, non tanto per ricavarci gloria e denaro, quanto per prefissarmi la presenza di localizzazioni, help, e varie altre piccolezze che fanno di un pezzo di codice una applicazione a tutti gli effetti.

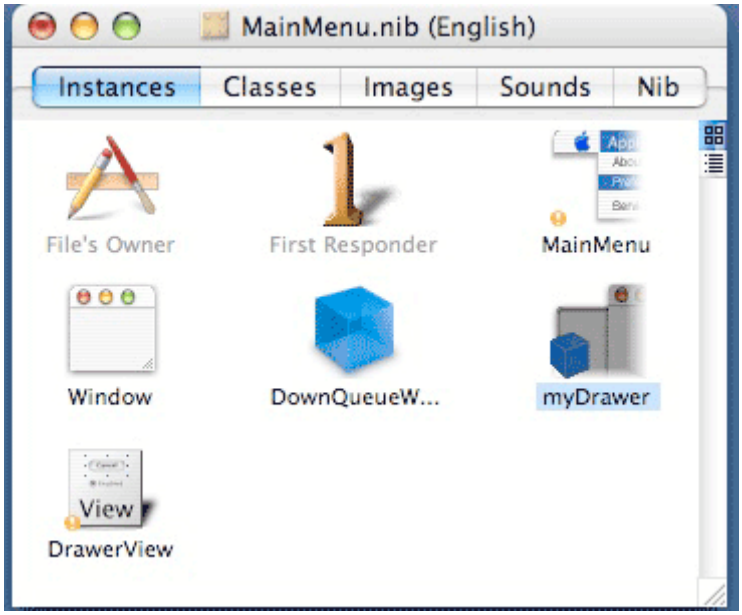
L'INTERFACCIA

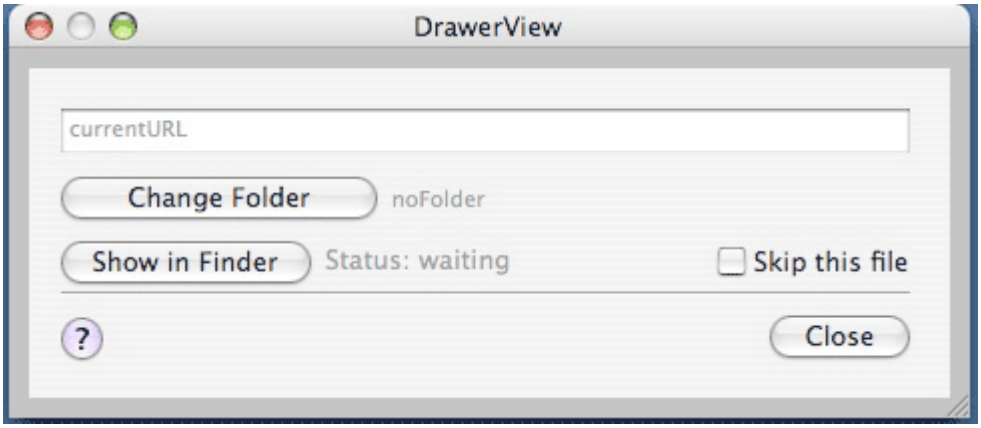
In base a questi requisiti, incomincio a rendere le cose un po' più specifiche, andando a disegnare un'interfaccia utente. Comincio da quella in inglese, ricordandomi che poi dovrò farne anche una versione in italiano.



Ci sono un'ampia zona sulla sinistra (un'immagine che sarà anche l'icona dell'applicazione) sulla quale si possono droppare dei collegamenti per inserirli nella lista dei trasferimenti. A completare la fascia superiore le informazioni relative al file in corso di trasferimento, con nome, stato di avanzamento, ed anche tutte le informazioni che si potranno ricavare in termini di tempo trascorso e residuo, e byte trasferiti. Una immagine rossa, gialla o verde rende conto dello stato corrente del trasferimento: rosso indica nessun trasferimento in corso, giallo indica trasferimenti attivi ma coda vuota, ed infine verde indica trasferimento in corso.

La parte inferiore della finestra contiene invece la lista dei file da trasferire e un pulsante di Aiuto. Ho aggiunto un classico controllore specifico per la finestra (`DownQueueWinCtrl`) e outlet a iosa per collegare tutti gli elementi dell'interfaccia.





Per visualizzare informazioni sui singoli file, ho pensato di utilizzare un `Drawer`, anche perché non ne ho mai utilizzato uno e voglio vedere come si fa. Per aggiungere un

drawer ad una finestra, non ho nemmeno letto una riga di documentazione. Ho utilizzato `Interface Builder`, aggiungendo un oggetto `NSDrawer` direttamente all'interno del `MainMenu.nib`. Quest'operazione non ha però introdotto una zona dove disegnarne il contenuto; bisogna quindi aggiungere anche una `NSView`. L'ho fatto droppando sopra l'icona dell'oggetto `myDrawer` una CustomView prelevata dalla pelette degli oggetti di Interface Builder. L'operazione, contestualmente, imposta l'outlet `contentView` di `myDrawer` alla view appena inserita, e mostra tale view in una finestra a parte di Interface Builder (che non corrisponde però ad una finestra dell'applicazione). Bisogna anche collegare l'outlet `parentWindow` di `myDrawer` alla finestra madre (nel mio caso, l'unica finestra operativa dell'applicazione) e, nel mio caso, il delegate per lasciare fare al controllore della finestra alcune operazioni. In effetti, il collegamento tra finestra e drawer si effettua dall'altro lato non tanto dalla finestra, quanto attraverso il controllore della finestra (il mio `DownQueueWinCtrl`). Anche tutti i controlli del drawer sono stati collegati tramite outlet a `DownQueueWinCtrl`, come pure questo controllore è il riferimento per le operazioni scatenate dai controlli del drawer. Così facendo la classe controller diventa piuttosto piena di outlet, e forse valeva la pena separare o semplificare la gestione in altro modo. Tuttavia, data la semplicità dell'applicazione, lascio tutto così.

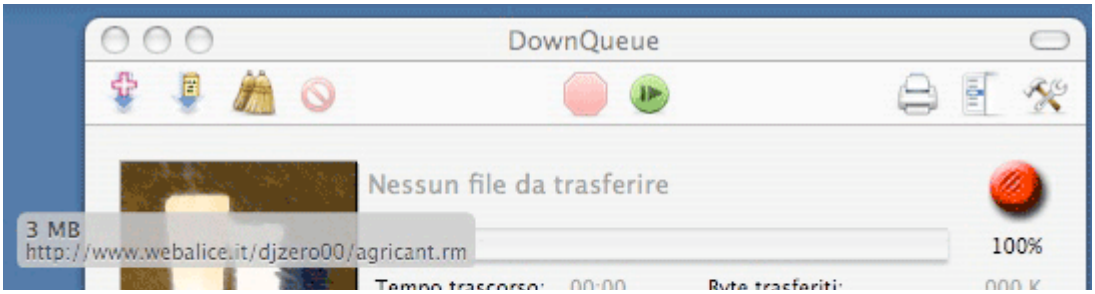
Oltre agli outlet, all'interno della classe controllore inserisco anche una variabile d'istanza `NSMutableArray`, destinata a contenere l'elenco di tutti gli URL da trasferire (e le informazioni accessorie). In effetti, per gli URL da trasferire ho costruito una classe apposita, `Url2DL`, che contenga appunto le informazioni relative allo URL, la destinazione del file trasferito, e di atre informazioni sullo stato corrente del trasferimento. In pratica quindi, una prima bozza dell'applicazione consiste nella visualizzazione di una lista di oggetti `Url2DL` all'interno di una finestra.

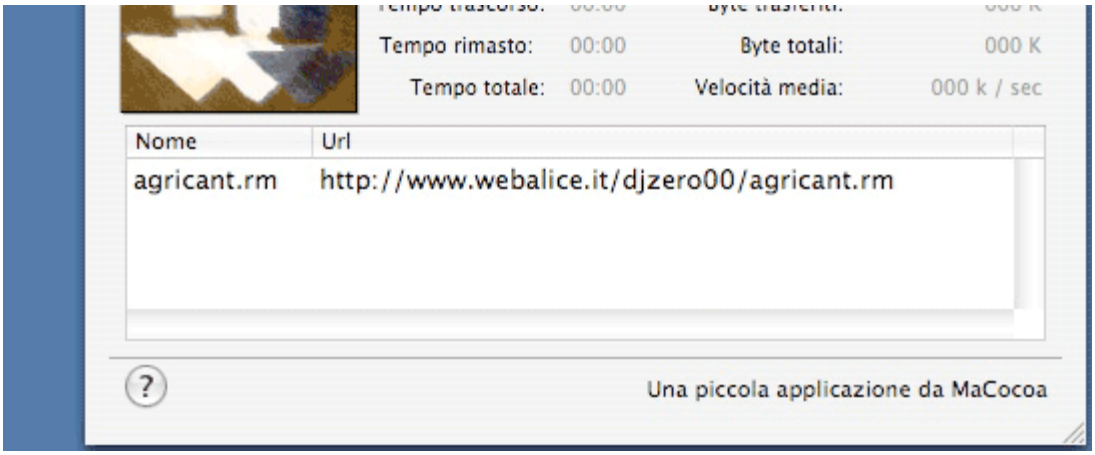
Detta così, la cosa si presenta piuttosto facile. In realtà, le cose da fare sono parecchie, e comincio a metterle in un qualche ordine.

DROP SULL'IMMAGINE

Comincio dal realizzare le funzioni di drop sull'immagine.

Costruisco una nuova classe, la `DropImage`, che è una sottoclasse di `NSImageView`, dove aggiungo appunto la gestione del drop. Ho aggiunto una variabile d'istanza, e l'ho classificata come `IBOutlet`. Fatto questo, entro in interface builder, ed attribuisco all'immagine che voglio rendere sensibile al drop proprio `NSImageView` come `Custom Class`. Effettuo anche il collegamento verso il controllore della finestra come outlet. In questo modo l'immagine saprà indirizzare al posto giusto gli URL che gli vengono droppati sopra.





Per quanto riguarda i metodi che gestiscono l'operazione di droppatura, non faccio altro che ricopiare ed adattare i metodi che avevo già scritto nel capitolo 19. Riassumendo e riscrivendo, i metodi da considerare sono sei.

Si comincia con `draggingEntered`, quando qualcosa trascinato al mouse entra nella zona sensibile (nel mio caso, sopra l'immagine). Il metodo serve a stabilire cosa che ne farà il destinatario degli oggetti trascinati.

Ho introdotto una specie di giochino grafico sull'immagine, che ne modifica leggermente la rappresentazione. Normalmente infatti l'immagine presenta una cornice in rilievo `NSImageFrameGrayBezel`; quando ci entra qualcosa, la cornice diventa `NSImageFramePhoto`. L'effetto è che l'immagine sembra gonfiarsi un poco quando ci si passa sopra con qualcosa in trascinamento. Il metodo dice che degli oggetti trascinati sopra ci farà nulla nella maggior parte dei casi; solo quando gli oggetti sono di tipo `NSURLPboardType` (in pratica, degli URL ben formati), ne farà qualcosa (un collegamento, `NSDragOperationLink`, un collegamento, che in pratica può significare molte cose...).

Altri due metodi sono facili da fare. Il primo, `draggingUpdated`, è attivo finchè si rimane sopra l'immagine, e l'altro, `draggingExited`, è chiamato quando si esce dall'immagine senza aver rilasciato il mouse (in pratica, facendo nulla). Ovviamente, il primo metodo non fa che ridabire l'operazione che sarà svolta, mentre il secondo rimette a posto lo stile della cornice dell'immagine.

Il primo metodo serio è `prepareForDragOperation`, che prepara le operazioni conseguenti al drop, accettando il compito nel caso di URL.

Il metodo che fa in pratica tutto il lavoro, `performDragOperation`, si limita ad invocare un metodo del controllore della finestra (ecco dove viene utile lo outlet) indicandogli quale URL aggiungere alla lista.

Infine, a conclusione delle operazioni, si rimette a posto l'aspetto dell'immagine con metodo `concludeDragOperation`.

Fino a qui, tutto bene e tutto chiaro (almeno per me). Ho già un primo problema. Quando trascino un URL sopra l'immagine, il punto di partenza dell'operazione di trasferimento è perfettamente chiaro. Completamente ignoto, invece, il punto di arrivo del trasferimento; in altre parole, non ho idea di dove andare a salvare il file risultato

dell'operazione di trasferimento. Del resto, l'operazione di drop è piuttosto naturale, ma non permette in modo naturale la specifica della destinazione. Mi giovo di alcuni valori di default e del fatto che molto spesso ci si aspetta che il file trasferito abbia lo stesso nome del file di partenza.

AGGIUNGERE UN FILE

L'azione scatenata da un drop riuscito sull'immagine richiama il metodo seguente (della classe `DownQueuWinCtrl`):

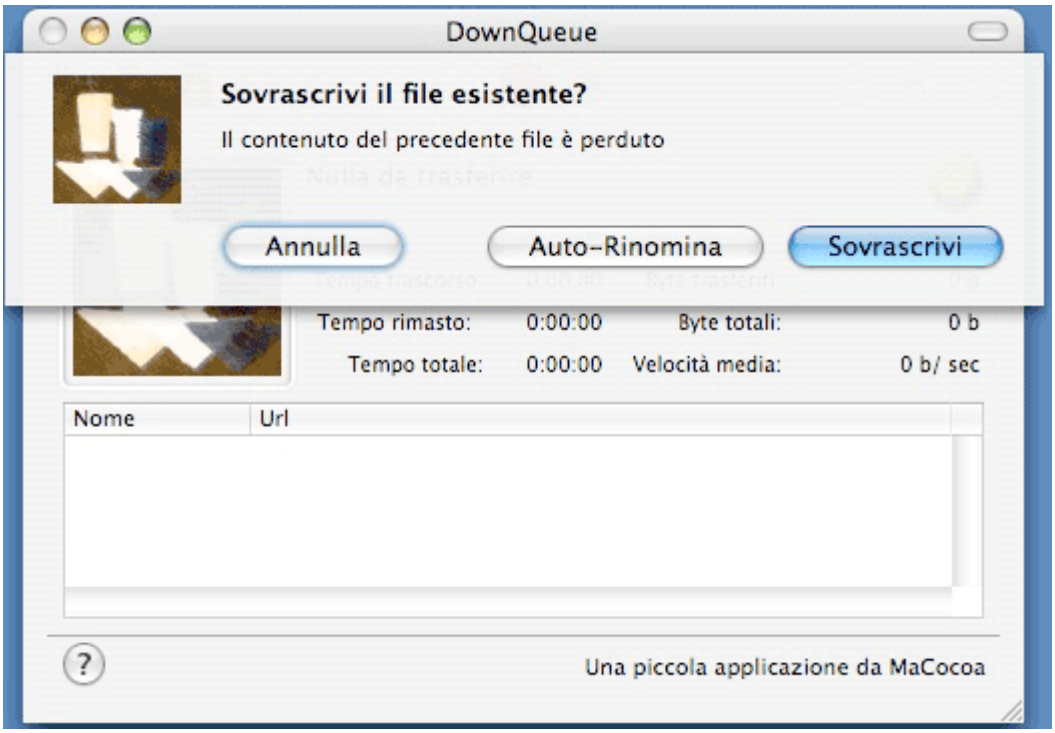
```
- (void) addClipFile2Download: (NSPasteboard *) pstBoard
{
    NSArray        * pList ;
    NSEnumerator     * enumerator ;
    NSString        * anUrl ;
    // e ne estraggo solo gli elementi url
    pList = [ pstBoard propertyListForType: NSURLPboardType ] ;
    enumerator = [ pList objectEnumerator ];
    // gli spazzolo tutti
    while ( anUrl = [enumerator nextObject] )
    {
            NSString       *  tmpURL  =  [[  NSURL  URLWithString:  anUrl  ]
absoluteString ];
        // se ho una stringa, e questa non e' vuota
        if ( tmpURL && [ tmpURL length] )
        {
            // costruisco un nuovo elemento
            Url2DL     * theNewElem = [[ [ Url2DL alloc] initWithUrl: tmpURL
andDLFolder: defaultDLDir ] autorelease ];
            // e lo aggiungo
            [ self addFile2Download: theNewElem ];
        }
    }
}
```

Ci sono le solite operazioni che recuperano le informazioni dalla pasteboard; anche se non riesco a capire come un utente normale possa droppare una lista di url, gestisco la pasteboard come se fosse in grado di restituire un array di URL. Per ogni URL ben formato che trovo, costruisco un elemento della classe `Url2DL` (ci arrivo in un attimo), e poi lo aggiungo alla lista degli elementi, attraverso il metodo seguente:

```
- (void )
addFile2Download: (Url2DL *) theNewElem
{
    if ( [ self shouldAdd2Queue: theNewElem ])
    {
        [ theURLlist addObject: theNewElem ];
        [ theQueue reloadData ] ;
        [ self performCurl ];
    }
}
```

Mi sono infatti accorto che potrebbero esserci problemi di varia natura quando gestisco un nuovo URL da trasferire. Ho così condizionato l'effettivo inserimento alla verifica di un metodo `shouldAdd2Queue` che effetta qualche controllo aggiuntivo sulla natura del file (più che altro, sulla sua possibile destinazione):

```
- (BOOL) shouldAdd2Queue: (Url2DL *) theNewElem
{
    NSString * finalFile ;
    // se la sovrascittura e' attiva, va tutto bene
    if ( checkOverwrite == NO )
        return ( YES ) ;
    // se arrivo qui, devo controllare eventuali sovrascritture
    // nome completo del file destinazione
                    finalFile     =     [[     theNewElem     theDestFolder]
stringByAppendingPathComponent: [ theNewElem theName] ];
    // vedi se esiste gia' un file con questo nome
    if ([ [ NSFileManager defaultManager] fileExistsAtPath: finalFile ] )
    {
        // si, chiedo cosa fare
        NSAlert *alert = [[NSAlert alloc] init];        // tre possibilita'
         [alert addButtonWithTitle: NSLocalizedString(@"Overwrite", @"DQWC:
alert overwrite, overwrite btn")];        // sovrascirvo
        [alert addButtonWithTitle: NSLocalizedString(@"AutoRename", @"DQWC:
alert overwrite, autorename btn")];        // piglio un nome a caso
          [alert addButtonWithTitle: NSLocalizedString(@"Cancel", @"DQWC:
alert overwrite, cancel btn")];            // lascio stare
         [alert setMessageText: NSLocalizedString(@"Overwrite the existing
file?", @"DQWC: alert overwrite, msgTxt")];
          [alert setInformativeText: NSLocalizedString(@"The content of the
previous file is lost.", @"DQWC: alert overwrite, infoTxt")];
        [alert setAlertStyle:NSWarningAlertStyle];
        [ theNewElem retain ];
           [ alert beginSheetModalForWindow: [ self window] modalDelegate:
self
                                                                didEndSelector:
@selector(checkOverWrite:returnCode:contextInfo:)    contextInfo:    (void*)
theNewElem ];
        return ( NO );
    }
    return ( YES );
}
```



Tra le preferenze dell'applicazione, ci sarà anche la possibilità di ignorare eventuali sovrascritture di un file (il flag `checkOverwrite`). è tuttavia meglio controllare se per caso il trasferimento del file indicato non vada a sovrascrivere un file già presente. Costruisco quindi il path completo della destinazione, e controllo se non esista già un file con lo stesso nome. Nel caso, preparo un messaggio di avvertimento che lascia all'utente la scelta definitiva di sovrascrivere o meno il file.

In definitiva, il metodo visto restituisce `YES` se si può procedere tranquillamente (nessuna sovrascrittura, oppure chissenefrega se c'è una sovrascrittura); risponde `NO` se l'utente decide fare altro.

Le opzioni che si possono scegliere sono racchiuse all'interno del metodo di chiusura del dialogo:

```
- (void)
checkOverWrite:(NSAlert          *)alert          returnCode:(int)returnCode
contextInfo:(void *)contextInfo
{
    Url2DL    * theNewElem = (Url2DL *) contextInfo ;
    // overwrite, faccio nulla
    if ( returnCode == NSAlertFirstButtonReturn)
    {
        // overwrite, faccio nulla
        [ theURLlist addObject: theNewElem ];
        [ theQueue reloadData ] ;
        [ self performCurl ];
    }
    if (returnCode == NSAlertSecondButtonReturn)
    {
        // autorename
        short        i = 0 ;
        // piglio l'estensione e la metto da parte
        NSString    * finalFile ;
        NSString    * newName ;
        NSString    * extension = [ [ theNewElem theName] pathExtension];
        // il nuovo nome e' quello vecchio con aggiunta in fondo
        // contnuo a cambiare l'estensione numerica...
        do
        {
            // proviamo con una estensione numerica progressiva
             NSString    * numext = [ NSString stringWithFormat: @"-%d", ++
i ];
            // nuovo nome del file
                  NSString      * newshname = [ [ [ theNewElem theName]
stringByDeletingPathExtension] stringByAppendingString: numext] ;
                   newName = [ newshname stringByAppendingPathExtension:
extension];
                             finalFile = [[ theNewElem theDestFolder]
stringByAppendingPathComponent: newName ];
        }
        // ... finche' ci sono file con lo stesso nome
             while ([ [ NSFileManager defaultManager] fileExistsAtPath:
finalFile ] ) ;

        // imposto il nuovo nome
        [ theNewElem setTheName: newName ];
        [ theURLlist addObject: theNewElem ];
        [ theQueue reloadData ] ;
        [ self performCurl ];
    }
```

```
    // negli altri casi, faccio nulla
    [ theNewElem release ];
    [alert release];
}
```

Se l'utente decide di sovrascrivere comunque, si inserisce l'elemento nella coda. La seconda opzione è di lasciare all'applicazione il compito di rinominare il file. Ciò avviene aggiungendo un numero progressivo tra il nome e l'estensione del file, in modo da avere nomi molto simili, eppure diversi (si tratta della stessa strategia realizzata da Safari, ad esempio; tuttavia, a differenza di Safari, qui ho pasticciato le cose, per cui se si aggiungono nella coda file con lo stesso nome, non esistendo già tali file nel file system dell'utente, saranno comunque sovrascritti...; presumo che ci saranno modifiche in questa parte del codice, ma non ora).

Infine, se l'utente decide di soprassedere, nessun file è aggiunto alla coda, e tutto rimane come prima.

ELEMENTI IN CODA

Le informazioni relative ad un singolo elemento nella coda sono racchiuse all'interno di una classe:

```
@interface Url2DL : NSObject {
    NSString            * theURL ;
    NSString            * theDestFolder ;
    NSString            * theName ;
    unsigned short        dlStatus ;
}
```

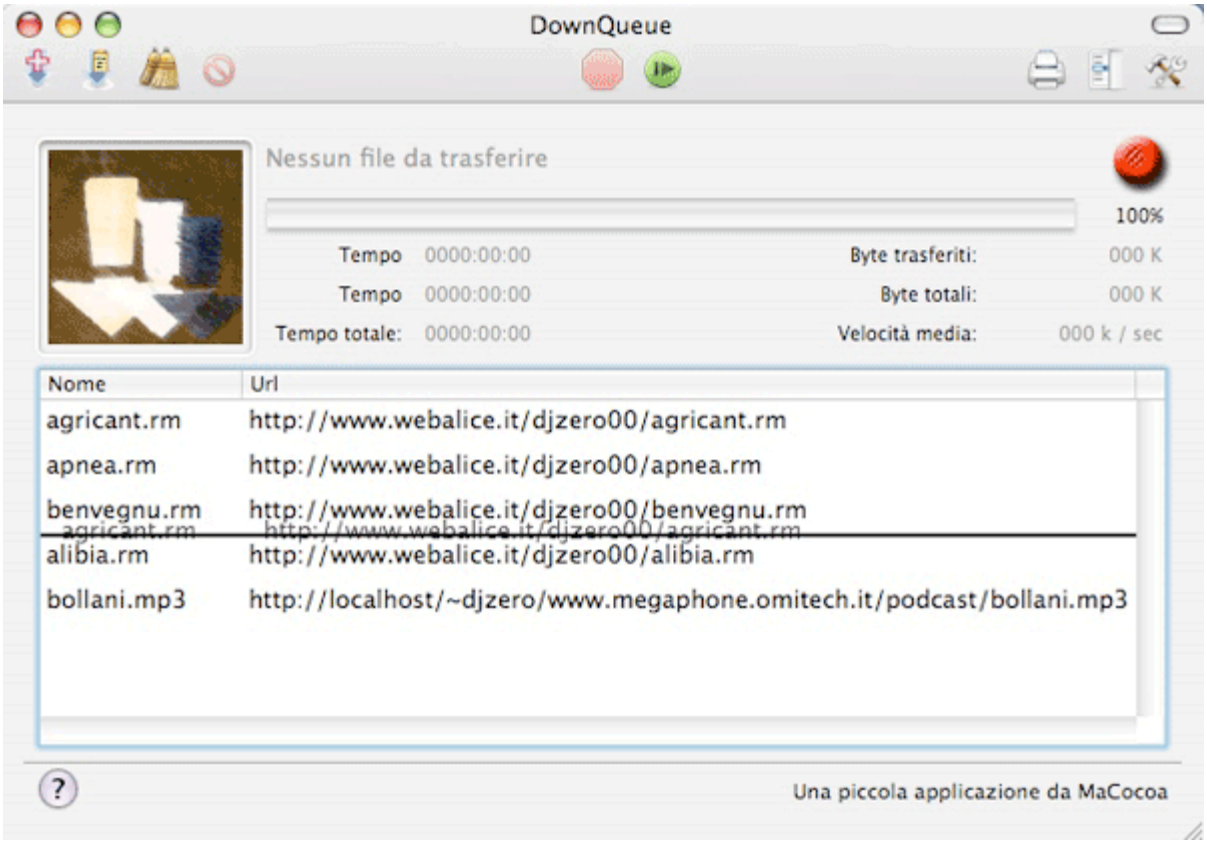
Le variabili d'istanza sono l'URL completo da dove avviene il trasferimento, mentre la specifica della destinazione è divisa nel nome e nel percorso sul disco fisso. Infine, una variabile intera discrimina lo stato corrente del trasferimento: terminato, in trasferimento, in attesa, sospeso, parziale ed in errore. Questa variabile sarà responsabile della modalità di visualizzazione di queste informazioni nella coda.

I metodi associati alla classe sono i soliti metodi standard di gestione (inizializzazione, accessor, eccetera). Segnalo solo i due metodi per il salvataggio su file ed il recupero dei dati da file, ovvero `encodeWithCoder` e `initWithcoder`, metodi che permettono il salvataggio di una coda su di un file per poter effettuare i trasferimenti in un secondo momento o su altro computer. Anche il metodo `description` è degno di interesse (ne parlavo il capitolo passato a proprosito del debugging): riporta in forma amena le informazioni contenuta in una istanza di `Url2DL`.

Non voglio inoltre tediare il lettore con la riproposizione dei soliti metodi per la sorgente dati della tabella che presenta la lista (per comodità, ho usato `DownQueueWinCtrl` stesso come sorgente dei dati). Mi limito a segnalare il seguente metodo, che colora in maniera diversa la riga a seconda dello stato del trasferimento:

```
- (void)
tableView:(NSTableView *)aTableView
    willDisplayCell:(id)aCell
    forTableColumn:(NSTableColumn *)aTableColumn
```

```
    row:(int)rowIndex
{
    switch ( [[theURLlist objectAtIndex: rowIndex] dlStatus]) {
    case    URLSTS_FINISHED :        // gia' scaricato
        [ aCell setTextColor: [ NSColor darkGrayColor]];
        break ;
    case    URLSTS_PARTIAL :
        [ aCell setTextColor: [ NSColor orangeColor]];
        break ;
    case    URLSTS_SUSPENDED :        // sospeso
        [ aCell setTextColor: [ NSColor lightGrayColor]];
        break ;
    case    URLSTS_DOWNLOADING :        // in corso di scaricamento
        [ aCell setTextColor: [ NSColor brownColor]];
        break ;
    case    URLSTS_WAITING :        // in attesa di essere scaricato
        [ aCell setTextColor: [ NSColor blackColor]];
        break ;
    case    URLSTS_DWLDERROR :        // in attesa di essere scaricato
        [ aCell setTextColor: [ NSColor redColor]];
        break ;
    }
}
```



Una caratteristica interessante e nuova nella gestione della tabella è la possibilità di riordinare gli elementi presenti nella lista. Ci sono due momenti da tenere presenti: configurare la `NSTableView` in modo che possa gestire le operazioni di drag and drop all'interno di se stessa, e poi riordinare la sorgente dati in modo da riflettere i cambiamenti effettuati.

Di default, la `NSTableView` non è soggetta al drag and drop interno. Occorre specificarlo esplicitamente, ad esempio nel metodo awakeFromNib della finestra che

contiene la tabella:

```
#define        localDragType        @"localDragType"
[     theQueue     registerForDraggedTypes:     [NSArray     arrayWithObject:
localDragType] ];
```

A questo punto, la `NSTableView` riceve tutti i messaggi tipici del drag and drop. In particolare, come si trova nelle guide di riferimento sull'argomento fornite da Apple, ci sono alcuni metodi da mettere a posto. Il primo è utilizzato per mettere da parte le informazioni alla partenza delle operazioni di drag. Il metodo fornisce come argomento l'elenco degli indici delle righe interessate (nel nostro caso, avendo scelto che la selezione della tabella può essere sempre e sola una riga, sarà un solo valore) e la pasteboard dove salvare i dati. Per semplicità, i dati salvati consistono in pratica nel solo indice della riga selezionata.

```
- (BOOL)
tableView:(NSTableView *)tv
        writeRowsWithIndexes:(NSIndexSet *)rowIndexes
        toPasteboard:(NSPasteboard*)pboard
{
NSData *data = [NSKeyedArchiver archivedDataWithRootObject: rowIndexes ];
[pboard declareTypes:[NSArray arrayWithObject:localDragType] owner:self];
[pboard setData: data forType:localDragType];
return YES;
}
```

Durante le operazioni di drag and drop, occorre assicurare il sistema che le cose stanno procedendo bene: è compito del metodo seguente.

```
- (NSDragOperation)
tableView:(NSTableView*)tv
        validateDrop:(id <NSDraggingInfo>)info
        proposedRow:(int)row
        proposedDropOperation:(NSTableViewDropOperation)op
{
return NSDragOperationEvery;
}
```

Vista la semplicità della cosa, non sono effettuate operazioni, ma solamente si conferma il proseguimento delle operazioni.

La parte conclusiva delle operazioni è svolta dal metodo successivo.

```
- (BOOL)
tableView:(NSTableView *)aTableView
        acceptDrop:(id <NSDraggingInfo>)info
        row:(int)row
        dropOperation:(NSTableViewDropOperation)operation
{
NSPasteboard* pboard = [info draggingPasteboard];
NSData* rowData = [pboard dataForType:localDragType];
NSIndexSet*            rowIndexes            =            [NSKeyedUnarchiver
unarchiveObjectWithData:rowData];
int dragRow = [rowIndexes firstIndex];

    Url2DL    * elem = [ theURLlist objectAtIndex: dragRow];
```

```
    [ elem retain ];    // il successivo remove elimina l'oggetto !
    [ theURLlist removeObjectAtIndex: dragRow] ;
    // aggiusto il conteggio delel righe, ora che ho eliminato un elemento
    if ( row > 0 )
        row -= 1 ;
    [ theURLlist insertObject: elem atIndex: row ];
    [ elem release ];    // ripristino il giusto retainCount
    [ theQueue reloadData ] ;
    return ( TRUE );
}
```

Si comincia recuperando le informazioni sulla sorgente delle operazioni (quindi, il numero della riga `dragRow` che è stata inizialmente trascinata). Come argomento del metodo, è fornito l'indice della riga `row` sulla quale è avvenuto il drop. Si tratta adesso di riordinare la sorgente dati `theURLlist` in modo da riflettere la modifica effettuata. Per farlo, estraggo l'elemento e lo cancello dalla lista. Devo effettuare una operazione di `retain`, altrimenti andrebbe perduto per sempre, essendo la lista l'unico punto che trattiene l'elemento in questione. Adesso inserisco l'oggetto nella nuova posizione (cui ho sottratto uno per tenere conto del fatto che ho appena tolto un elemento...). La classica richiesta di ridisegnare la lista chiude il metodo. Metodo che restituisce `TRUE` per indicare che tutto è andato bene.

TOOLBAR ED ALTRE AZIONI

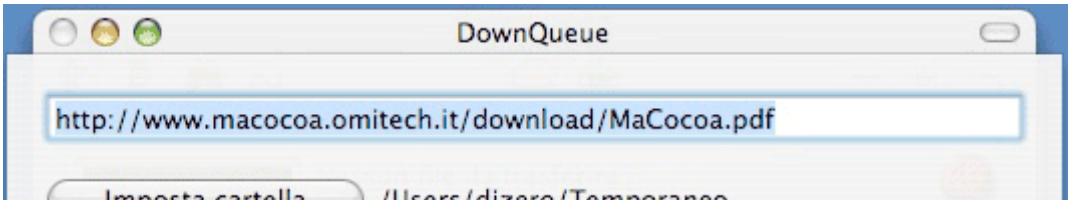
Una toolbar semplifica l'interazione con l'applicazione in maniera drastica, per cui ho inserito molti comandi nel suo interno.

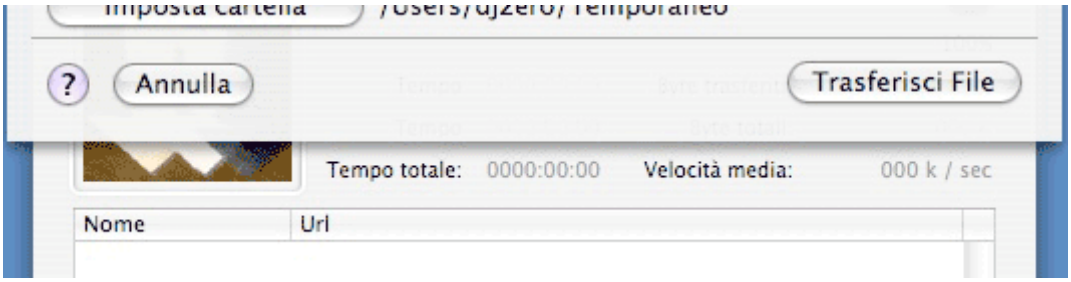
Piuttosto che discutere a lunga sulla sua realizzazione, che non presenta grosse novità, illustro più in dettaglio le varie operazioni che si possono effettuare attraverso i pulsanti della toolbar e le voci di menu corrispondenti.

L'aggiunta di un elemento alla lista dei trasferimenti si può al momento effettuare solo tramite drag and drop sulla immagine presente sull'interfaccia. Ecco altre tecniche.

Si può aggiungere un elemento alla lista attraverso specifica diretta dell'indirizzo. Allo scopo, apro una finestrella ausiliaria.

```
- (IBAction)addFile2Queue:(id)sender
{
    // costruisco lo sheet e lo apro
    SetDLFileWinCtrl *tmpCVCtrl = [[ SetDLFileWinCtrl alloc] init ] ;
    [ tmpCVCtrl setTheCaller: self ] ;
[NSApp beginSheet: [ tmpCVCtrl window ]
modalForWindow: [ NSApp mainWindow ]
modalDelegate: nil
            didEndSelector: nil
contextInfo: nil];
}
```





La finestra (o meglio, il suo controllore) appartiene alla classe `SetDLFileWinCtrl`, che funziona come una sheet. Questo significa che la finestra si apre sopra la finestra principale, intercettando tutti gli eventi relativi alla finestra (e, dal momento che la mia applicazione ha una sola finestra, dell'intera applicazione). Da notare come alla finestra impongo come variabile d'istanza un riferimento alla classe chiamante. In questo modo, dall'interno dei metodi di `SetDLFileWinCtrl` posso richiamare i metodi propri di `DownqueueWinCtrl`, ad esempio per aggiungere un file alla lista dei trasferimenti.

La finestra modale presenta semplicemente un campo per immettere l'indirizzo completo del file da trasferire ed un pulsante per selezionare la directory locale di destinazione. Due pulsanti, uno per acconsentire al trasferimento e l'altro per annullarlo, completano il tutto.

Il primo pulsante scatena l'esecuzone del seguente metodo:

```
- (IBAction)downloadButton:(id)sender
{
    // assegno un nuovo url da scaricare
    Url2DL    * theNewElem ;
    // butto via la finestra
    [[self window] setIsVisible: FALSE ];
    // ho finito di lavorare modale con la sheet
    // il codice di ritorno e' la riga selezionata
    [NSApp endSheet: [self window] returnCode: 0 ];
    // predispongo i valori dell'oggetto
     theNewElem = [[ [ Url2DL alloc] initWithUrl: [ theURL stringValue]
andDLFolder: [ theDestFolder stringValue] ] autorelease ];
    // lo aggiungo alla lista
    [ theCaller addFile2Download: theNewElem ];
}
```

Semplicemente, si costruisce un elemento della classe `Url2DL` con i valori ottenuti dall'utente, e lo si aggiunge alla lista.

Più articolata la selezione della directory di destinazione: in prima battuta interviene il metodo seguente, che mostra il tipico pannello di selezione directory

```
- (IBAction)setDLDestination:(id)sender
{
    // per scegliere uan directory, paradossalmente si usa il pannello
    // per scegliere un file... opportunamente configurato
    NSOpenPanel    *sPanel = [ NSOpenPanel openPanel ] ;
    // preparo il pannello di salvataggio file
     [ sPanel setTitle: NSLocalizedString(@"addFile: Saving into folder",
@"SDLFWC:, save file panel, titolo") ];
    [ sPanel setPrompt: NSLocalizedString(@"addFile: Select the destination
folder", @"SDLFWC:, save file panel, prompt") ];
```

```
    [ sPanel setCanChooseDirectories: YES ];
    [ sPanel setCanChooseFiles: NO ];
    [ sPanel beginSheetForDirectory:nil file:nil types: nil modalForWindow:
[ self window]
                                                    modalDelegate:      self
didEndSelector:@selector(openPanelDidEnd:returnCode:contextInfo:)
contextInfo:nil ];
}
```

Questo metodo, tuttavia, passa la palla alla gestione standard, e lascia la gestione del valore di ritorno (la directory scelta) al metodo delegato:

```
-   (void)openPanelDidEnd:(NSOpenPanel   *)panel   returnCode:(int)returnCode
contextInfo:(void *)contextInfo
{
    if ( returnCode == NSOKButton )
    {
        // se e' andato bene, prelevo il nome del file
        NSString *aFile = [ panel filename ] ;
        // e lo assegno alla variabile interna
        [ theDestFolder setStringValue: aFile ] ;
    }
}
```

Questo metodo, se è stato premuto il pulsante di scelta, recupera il percorso della directory e l'assegna al campo della finestra (da dove il metodo `downloadButton` la preleverà quando necessario).

Un giochino interessante avviene all'apertura della finestra, all'interno del metodo `windowDidLoad`

```
- (void)
windowDidLoad
{
    // se in clipboard c'e' un indirizzo, uso quello
    NSPasteboard    * pstBoard = [NSPasteboard generalPasteboard];
      NSArray                  * pList = [ pstBoard propertyListForType:
NSURLPboardType ] ;
    if ( [ pList count] )
    {
            NSString       * tmpURL = [[ NSURL URLWithString: [ pList
objectAtIndex: 0 ] ] absoluteString ];
        [ theURL setStringValue: tmpURL ];
    }
    // altrimenti, un url di default
    else
    {
                                            [     theURL     setStringValue:
@"http://www.macocoa.omitech.it/download/MaCocoa.pdf" ] ;
    }
    // metto la directory di default
    [ theDestFolder setStringValue: [ theCaller defaultDLDir] ];
}
```

Se all'interno degli appunti si trova un indirizzo URL ben formato, ebbene, questo indirizzo è utilizzato come punto di partenza per la scelta dell'utente. Altrimenti, è utilizzato un indirizzo di default (un po' banale). La directory di destinazione iniziale è ricavata ancora una volta attraverso la classe chiamante; sarà un valore di default nelle

preferenze dell'utente.

In effetti, questa cosa di recuperare dagli appunti un indirizzo ben formato mi piace così tanto che ho aggiunto un metodo apposta, in modo da associare l'azione ad un pulsante della toolbar e ad una voce di menu:

```
- (IBAction)addFileFromClip:(id)sender
{
    // recupero il testo dalla clipboard
    NSPasteboard    * pstBoard = [NSPasteboard generalPasteboard];
    // e ne estraggo solo gli elementi url
    [ self addClipFile2Download: pstBoard];
}
```

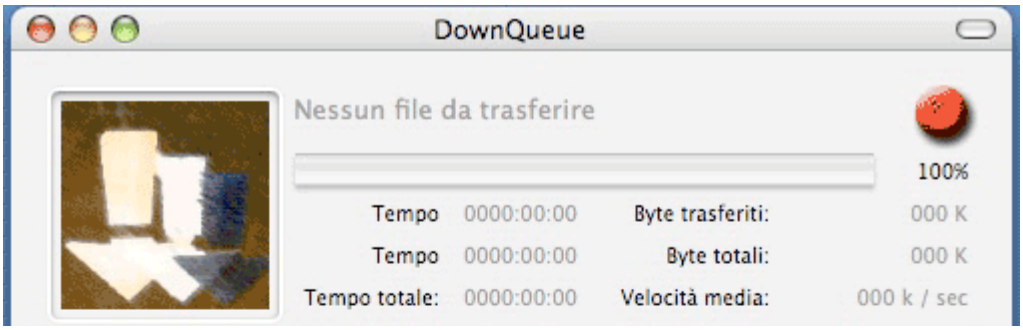
Oltretutto, la cosa è piuttosto semplice. Dopo aver individuato la pasteboard corretta, è sufficiente invocare lo stesso metodo utilizzato per il drag and drop sull'immagine (che utilizza sempre concetti legati alle pasteboard) per effettuare l'operazione.

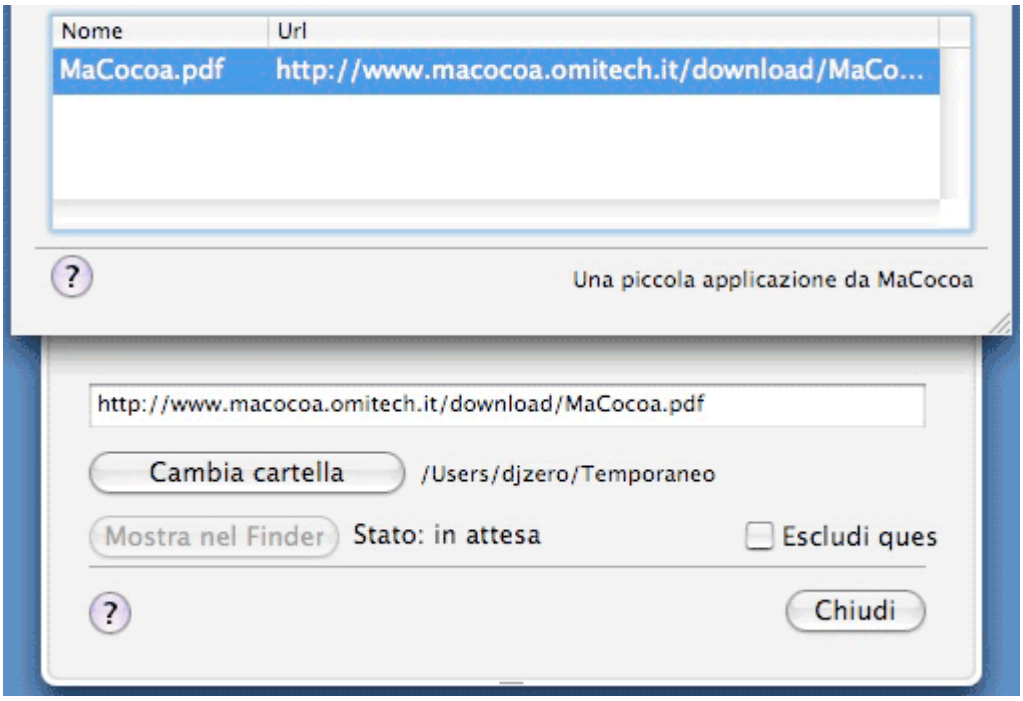
Per eliminare un elemento dalla coda, la cosa è piuttosto semplice. Basta selezionare l'elemento nella coda, ed utilizzare l'apposito pulsante della toolbar o voce di menu.

```
- (IBAction)deleteItem:(id)sender
{
    // recupero l'elemento
      Url2DL            * theElem = [theURLlist objectAtIndex: [ theQueue
selectedRow] ] ;
    // se e' in corso di scaricamento, faccio nulla
    if ( [ theElem dlStatus] == URLSTS_DOWNLOADING )
        return ;
    // elimino l'elemento selezionato
    [theURLlist removeObjectAtIndex: [ theQueue selectedRow] ] ;
    // riaggiorno un po' tutto
    [ theQueue reloadData ] ;
    if ( [ theDrawer state ] == NSDrawerOpenState )
        [ self refreshDrawer ];
    [ self saveListStatus ];
}
```

IL DRAWER

La presenza nel metodo precedente di una istruzione che riguarda il drawer mi consente di introdurre il concetto. Ho già descritto come lo abbia definito in Interface Builder, aggiungendolo al file nib direttamente dalla palette, ed associandoci una `NSView` isolata (non contenuta all'interno di una finestra). A quanto pare, è tutto ciò che è necessario fare per la sua gestione.





Un drawser ha uno stato, che può essere visibile o invisibile, ed una sua disposizione preferita (alto, basso, destra, sinistra), ovvero il lato da cui uscirà quando se ne richiede la visualizzazione. Si tratta di una direzione preferita, in quanto il lato di uscita dipende anche dalla geometria della finestra a cui fa riferimento. Ho scelto la direzione "basso", ma in realtà il drawser uscirà dal lato inferiore solo se c'è abbastanza spazio per visualizzarlo (almeno in parte), altrimenti uscirà dal lato superiore. Uguale sorte se avessi scelto sinistra o destra, esce sempre dal lato dove trova spazio.

Ho stabilito un paio di modi con cui visualizzarlo. Direttamente da voce di menu/pulsante della toolbar, oppure facendo un doppio clic su di un elemento della lista. Nel primo caso, il metodo è direttamente collegato come action al pulsante/menu; nel secondo, occorre esplicitamente indicare la cosa, ad esempio all'interno del metodo `awakeFromNib` della finestra

```
    [ theQueue setDoubleAction: @selector(showInfo:) ];
```

In entrambi i casi, il metodo `showInfo` funziona allo scopo:

```
- (IBAction)showInfo:(id)sender
{
    if ( [ theDrawer state ] == NSDrawerClosedState )
    {
        [ self refreshDrawer ];
        [ theDrawer open ] ;
    }
    else if ( [ theDrawer state ] == NSDrawerOpenState )
        [ theDrawer close ] ;
}
```

La gestione è bistabile, ovvero, il comando alterna tra lo stato visibile e lo stato invisibile. In entrambi i casi, il drawer compie una amorevole animazione nell'uscire o nel rientrare nel lato preferito.

Il drawer contiene le informazioni associate ad un `Url2DL`, quindi URL, nome e cartella destinazione. il metodo refreshDrawer serve proprio ad aggiustare i valori a seconda della selezione corrente in lista.

```
- (void)
refreshDrawer
{
    // se non c'e' alcuna riga selezionata
    if ( [ theQueue selectedRow] == -1 )
    {
        // metto tutto in condizioni nulle
        [ drwURL setEnabled: NO ] ;
             [ drwURL setStringValue: NSLocalizedString(@"DrwDefaultURL",
@"DQWC: drw defautl url") ] ;
                                      [    drwDLFolder    setStringValue:
NSLocalizedString(@"DrwDownloadFolder", @"DQWC: drw download folder") ] ;
        [ drwSkipThisFile setEnabled: NO ] ;
        [ drwShowFinder setEnabled: NO ] ;
        [ drwChangeFolder setEnabled: NO ] ;
           [ drwFileSts setStringValue: NSLocalizedString(@"DrwFileStatus",
@"DQWC: file sts, nessuno") ];
        [ drwFileSts setEnabled: NO ] ;
    }
    else
    {
        // recupero l'elemento selezionato
            Url2DL      * theElem = [theURLlist objectAtIndex: [ theQueue
selectedRow] ] ;
        // imposto le informazioni
        [ drwURL setEnabled: YES ] ;
        [ drwURL setStringValue: [ theElem theURL ] ] ;
        [ drwDLFolder setStringValue: [ theElem theDestFolder ] ] ;

        switch ( [ theElem dlStatus] ) {
        // gestione dei vari casi ...
        // ...
        case    URLSTS_DOWNLOADING :        // in corso di scaricamento
            [ drwURL setEnabled: NO ] ;
            [ drwSkipThisFile setEnabled: NO ] ;
            [ drwChangeFolder setEnabled: NO ] ;
            [ drwShowFinder setEnabled: NO ] ;
            [ drwSkipThisFile setState: NO ] ;
                 [ drwFileSts setStringValue: NSLocalizedString(@"Status:
Downloading", @"DQWC: file sts, downloading") ];
            break ;
        case    URLSTS_WAITING :        // in attesa di essere scaricato
            [ drwURL setEnabled: YES ] ;
            [ drwChangeFolder setEnabled: YES ] ;
            [ drwSkipThisFile setEnabled: YES ] ;
            [ drwShowFinder setEnabled: NO ] ;
            [ drwSkipThisFile setState: NO ] ;
                 [ drwFileSts setStringValue: NSLocalizedString(@"Status:
Waiting", @"DQWC: file sts, in attesa") ];
            break ;
        // gestione degli altri casi ...
        }
    }
}
```

Il metodo è piuttosto prolisso. Nel caso non ci siano selezioni attive, provvede a disabilitare tutti i controlli e a inserire stringhe neutre al posto di nome ed url. Con un

elemento selezionato, si recuperano le informazioni, mentre i controlli presenti sono gestiti a seconda dello stato del trasferimento. Ad esempio, se il trasferimento è in corso, non è possibile cambiare URL e nome, anzi, non è possibile fare alcuna modifica. Se invece l'elemento è in attesa di essere scaricato, è possibile modificare ogni singolo attributo (url, destinazione, eccetera) ed anche manipolarne lo stato (ad esempio, escludendolo dal trasferimento, ma mantenendolo nella lista).

All'interno del drawer sono infatti presenti alcuni pulsanti che permettono di modificare alcune caratteristiche dell'elemento. Per modificare lo URL, si opera direttamente sulla stringa che lo rappresenta. Perché l'applicazione riconosca le modifiche in questo campo di testo, bisogna che ne sia informata. Una tecnica classica è di attribuire un delegato al campo di testo (l'ho fatto in Interface Builder, ed ho assegnato sempre la classe `DownQueueWinCtrl`), e di intercettare alcune notifiche, o meglio, la notifica `controlTextDidChange`. In questo modo, ogni volta che il testo all'interno del campo è modificato dall'utente, l'ambiente operativo informa della cosa il delegato, che può agire di conseguenza:

```
- (void) controlTextDidChange:(NSNotification *) aNotification
{
    NSTextField *quale = [ aNotification object ];
    // verifico che sia il mio campo
    if ( [quale tag] == 11 )
    {
        // recupero l'elemento
            Url2DL      * theElem = [theURLlist objectAtIndex: [ theQueue
selectedRow] ] ;
        // e gli cambio l'url
        [ theElem setTheDestFolder: [ drwURL stringValue] ];
        // salvo i dati
        [ self saveListStatus ];
    }
}
```

Nel caso, nulla di speciale: controllo che sia proprio il campo di testo ad essere stato modificato (verifica un po' paranoica, ma tant'è), e modifico la variabile d'istanza dell'elemento Url2DL corrente con il nuovo valore

Per modificare la cartella di destinazione, c'è un pulsante del tutto simile a quello visto nella finestra di immissione di un nuovo URL (e che svolge in pratica le medesime operazioni). C'è poi un pulsante di spunta, che permette di escludere il file dal trasferimento (in pratica, modifica la variabile d'istanza `dlStatus` di `Url2DL` da uno stato di attesa ad uno di sospensione, e viceversa). Grazioso poi il pulsante che permette, per i file già scaricati, di individuarli nel Finder (nel caso non si riesca a trovarli, o per raggiungerli rapidamente in una selva di finestra aperte).

```
- (IBAction) drwShowInFinder:(id)sender
{
    // recupero l'elemento
      Url2DL            * theElem = [theURLlist objectAtIndex: [ theQueue
selectedRow] ] ;
    // il nome completo del file
          NSString          *  finalFile   =   [[   theElem   theDestFolder]
stringByAppendingPathComponent: [ theElem theName] ];
    // ed uso l'apposito metodo di NSWorkspace
              [   [NSWorkspace   sharedWorkspace]   selectFile:   finalFile
```

```
inFileViewerRootedAtPath: [ theElem theDestFolder] ];
}
```

Una volta determinato il path completo, giungendo percorso e nome del file, si utilizza un metodo della classe `NSWorkspace` per aprire una nuova finestra del Finder, quella appunto che contiene il file trasferito.

SUPPORTO ALL'APPLICAZIONE

Qui e lì nel codice compare un'istruzione che invoca il metodo `saveListStatus`. L'idea è di mantenere da qualche parte l'elenco dei file in corso di trasferimento, e di ripristinare la lista tra una chiusura dell'applicazione e la successiva riapertura; una specie di salvataggio della sessione in corso, insomma. Il posto deputato dove scrivere queste cose è la cartella `Application Support` dell'utente corrente (la si trova all'interno della cartella `Libreria`. In effetti, ci sono diverse cartelle Application Support in giro per il sistema, ma a me interessa solo quella specifica dell'utente). Molte applicazioni professionali in effetti scrivono informazioni per il proprio funzionamento in quella cartella, piuttosto che utilizzare altri posti in giro per il sistema.

Mescolo il tutto con le caratteristiche legate alle preferenze, e descrivo il metodo `init` del controllore:

```
- (id )
init
{
    NSArray            * paths ;
    NSMutableArray     * tmpArray ;
    NSString        * destPath ;
    NSUserDefaults     * defaults = [NSUserDefaults standardUserDefaults];

    self = [ super init ];
    // inizializzo un po' di variabili
    currItem = nil ;
    downLoadStartTime = TickCount() ;
    theCurlServer = nil ;
    [ self setTheURLlist: [NSMutableArray array]] ;
    // queste dovrei pigliarle dalle preferenze
    destPath = [ defaults stringForKey: UD_DefaultDLFolder ] ;
    if ( destPath )
        [ self setDefaultDLDir: destPath ] ;
    else
        [ self setDefaultDLDir: @"/Users/Shared" ] ;
    checkOverwrite = [ defaults boolForKey: UD_CheckOverwrite ] ;
    autoCleanQueue = [ defaults boolForKey: UD_AutoCleanQueue ] ;
    startAtStartup = [ defaults boolForKey: UD_StartAtStartup ] ;
    downloadStatus = startAtStartup ;
```

Fino a qui, tutto piuttosto tranquillo, Ci sono un po' di inizializzazioni sparse, la maggior parte delle quali prelevate dalle preferenze utilizzando la classe `NSUserDefaults` (vedi poi).

```
    // guardo se ce' gia' la directory "Application Support"
                                                    paths                =
NSSearchPathForDirectoriesInDomains(NSApplicationSupportDirectory,
```

```
NSUserDomainMask, YES);
    if ([paths count] == 0)
    {
        // non c'e' neppure la cartella "Application support"...
        NSAlert *alert = [[NSAlert alloc] init];
           [alert addButtonWithTitle: NSLocalizedString(@"AppSuppFld1 Ok",
@"DQWinCtrl: alert OK manca cartella app supp")];
          [alert setMessageText: NSLocalizedString(@"AppSuppFld1 AppError",
@"DQWinCtrl: msgtxt manca cartella app supp")];
        [alert setInformativeText:
                          NSLocalizedString(     @"Can't find the application
Support folder. Quitting DownQueue",
                                            @"DQWinCtrl: infoTxt manca cartella
app supp") ];
        [alert setAlertStyle:NSCriticalAlertStyle];
          [ alert beginSheetModalForWindow: [ self window] modalDelegate:
self
                                                                  didEndSelector:
@selector(stopMe:returnCode:contextInfo:) contextInfo: nil ];
        return ( self );
    }
```

Comincia la saga della cartella Application Support. In primo luogo si controlla se esiste: a meno di una installazione pura del sistema operativo, dovrebbe essere già presente. Non lo fosse, segnalo la cosa con un bel dialogo di avvertimento, e poi chiudo l'applicazione (non provo neppure a costruire la cartella, insomma).

```
    // la mia cartella all'interno di Application Support
                         destPath      =      [[paths      objectAtIndex:0]
stringByAppendingPathComponent:@"DownQueue"];
    // il nome del file nella mia cartella
                       [     self     setDefaulfFileList:     [     destPath
stringByAppendingPathComponent:@"initQueue.plist"] ];
    // vedo se esiste il file
    if ([ [ NSFileManager defaultManager] fileExistsAtPath: defaulfFileList
] )
    {
        // il file esiste, lo leggo
                   tmpArray  =  [NSKeyedUnarchiver  unarchiveObjectWithFile:
defaulfFileList ];
        // ci sono perfino degli elementi
        if ( tmpArray )
            // ci inizializzo la lista
            [ self setTheURLlist: tmpArray ];
    }
```

Questo dovrebbe essere il pezzo di codice standard di ogni esecuzione. Costruisco il path di un file, che si chiama `initQueue.plist`, all'interno della cartella `DownQueue` interna alla cartella `Application Support`. Con questo path, vado a leggerne il contenuto, che è una proprety list con la quale inizializzo la lista dei trasferimenti `theURLList`.

Va tuttavia tenuto presente che la prima volta che si lancia l'applicazione, la cartella e tanto meno il file sono presenti.

```
    else
    {
        // il file non esiste
```

```
        BOOL    isdir = YES;
        // vedo se esiste almeno la cartella (prima esecuzione)
          if ([ [ NSFileManager defaultManager] fileExistsAtPath: destPath
isDirectory: & isdir] == NO || isdir == NO)
        {
            // non c'e' la cartella, la creo
              if ( [ [ NSFileManager defaultManager] createDirectoryAtPath:
destPath attributes:nil] == NO )
            {
                // non riesco a creare la cartella...
                NSAlert *alert = [[NSAlert alloc] init];
                                                    [alert  addButtonWithTitle:
NSLocalizedString(@"AppSuppFld2 Ok", @"DQWinCtrl: alert ok manca cartella
appsupp/DQ")];
                     [alert setMessageText: NSLocalizedString(@"AppSuppFld2
AppError", @"DQWinCtrl: msgTxt manca cartella appsupp/DQ")];
                      [alert setInformativeText: NSLocalizedString(@"Cannot
create Application Support folder. Quitting DownQueue",
                                                      @"DQWinCtrl: infoTxt manca
cartella appsupp/DQ") ];
                [alert setAlertStyle:NSCriticalAlertStyle];
                [alert runModal] ;
                          [ alert beginSheetModalForWindow: [ self window]
modalDelegate: self
                                                                didEndSelector:
@selector(stopMe:returnCode:contextInfo:) contextInfo: nil ];
                return ( self );
            }
        }
        // a questo punto provo a creare il file
              if  (  [  [  NSFileManager  defaultManager]  createFileAtPath:
defaulfFileList contents:nil attributes:nil ] == NO )
        {
            // non riesco a creare il file
            NSAlert *alert = [[NSAlert alloc] init];
             [alert addButtonWithTitle: NSLocalizedString(@"SuppFileCrt Ok",
@"DQWinCtrl: alert ok fallita creazione file lista")];
                    [alert  setMessageText:  NSLocalizedString(@"SuppFileCrt
AppError", @"DQWinCtrl: msgTxt fallita creazione file lista")];
               [alert setInformativeText: NSLocalizedString(@"Cannot create
Queue file list. Quitting Downqueue",
                                                                    @"DQWinCtrl:
infoTxt fallita creazione file lista") ];
            [alert setAlertStyle:NSCriticalAlertStyle];
             [ alert beginSheetModalForWindow: [ self window] modalDelegate:
self
                                                                didEndSelector:
@selector(stopMe:returnCode:contextInfo:) contextInfo: nil ];
            return ( nil );
        }
}
```

Se il file non esiste, vado in primo luogo a verificare se per caso non esiste neppure la cartella, sintomo di prima esecuzione dell'applicazione (o di un utente che cancella cartelle più o meno a caso). Se necessario, la creo nuova. In tutte queste operazioni, mi giovo dei metodi della classe `NSFileManager`, che appunto fornisce comode funzioni adatte alla bisogna. Concludo quindi con la creazione del file (vuoto) nella posizione corretta.

Se queste operazioni per qualche motivo non riescono (il file system deve essere in tal caso in condizioni ben disgraziate), l'applicazione mostra un dialogo informativo e

termina l'esecuzione dell'applicazione.

```
    // ripulisco la lista dei file presenti
    [ self adjustURLList ];
    return ( self );
}
```

Le ultime operazioni svolte consistono nell'aggiustare la lista dei trasferimenti, in accordo con le preferenze.

```
- (void) adjustURLList
{
    NSEnumerator     * enumerator ;
    Url2DL            * theNewElem ;
    // spazzolo i dati
    enumerator = [ theURLlist objectEnumerator ];
    while ( theNewElem = [enumerator nextObject] )
    {
        // se sono in downloading, li cambio in waiting
        if ( [ theNewElem dlStatus] == URLSTS_DOWNLOADING )
            [ theNewElem setDlStatus: URLSTS_WAITING ];
         if ( autoCleanQueue && ( [ theNewElem dlStatus] == URLSTS_FINISHED
) )
            [ theURLlist removeObject: theNewElem ];
    }
}
```

Qui sono esaminati tutti gli elementi in lista, e sono messi nello stato di attesa trasferimento. Se ci sono poi elementi il cui trasferimento è terminato, e l'utente ha scelto di mantenere pulita la lista, allora questi elementi sono eliminati.

A questo punto si può ritornare al metodo `saveListStatus`, da cui sono partito. Ogni volta che la lista dei trasferimenti subisce modifiche (perché è stato aggiunto o eliminato un elemento, o perché un elemento ha cambiato il proprio stato), conviene salvare il tutto nel file, invocando appunto il metodo citato:

```
- ( void) saveListStatus
{
         if   (  [NSKeyedArchiver   archiveRootObject:  theURLlist   toFile:
defaulfFileList ] == NO )
        NSLog(@"Errore archiviazione 1");
}
```

I metodi di `NSKeyedArchiver` producono dei file che sono delle property list. Sono file binari, che però possono essere facilmente consultati con l'applicazione `Property List Editor`, disponibile una volta installati i `Developer's Tool` (e quindi, visto l'argomento di questo sito, dovreste già avere installato). Non che si capisca moltissimo, ma intanto...

A questo punto, viene naturale la possibilità di salvare la lista dei trasferimenti. Facile associare i metodi per la scrittura e la lettura alle voci di menu classiche `Open` e `Save`.

```
- (IBAction)listSave:(id)sender
{
    NSSavePanel    *sPanel = [ NSSavePanel savePanel ] ;
```

```
    // personalizzo il pannello di salvataggio
      [ sPanel setTitle: NSLocalizedString(@"Save list", @"DQWC: list save
panel, titolo") ];
    // i file salvati avranno l'estensione dql
    [ sPanel setRequiredFileType: @"dql" ];
      [ sPanel beginSheetForDirectory:nil file:nil modalForWindow: [self
window] modalDelegate: self
          didEndSelector:@selector(savePanelDidEnd:returnCode:contextInfo:)
contextInfo: nil ] ;
}

-   (void)savePanelDidEnd:(NSSavePanel   *)sheet   returnCode:(int)returnCode
contextInfo:(void *)contextInfo;
{
    if ( returnCode == NSOKButton )
    {
        // recupero il nome del file
        NSString *aFile = [ sheet filename ] ;
        // archivio l'intera struttura dati
        [NSKeyedArchiver archiveRootObject: theURLlist toFile: aFile ];
    }

}
```

Il salvataggio avviene in due passaggi. Nel primo si invoca il dialogo tipico, nella modalità modale (scusate il gioco di parole); nel secondo, si procede alla scrittura su file usando la stessa primitiva di `NSKeyedArchiver`.

Per quanto riguarda la lettura, si procede anche qui in due passaggi:

```
- (IBAction)listLoad:(id)sender
{
    NSOpenPanel        *oPanel = [ NSOpenPanel openPanel ] ;
    // imposto un po' di caratteristiche del dialogo
     [ oPanel setTitle: NSLocalizedString(@"LoadList Title", @"DQWC: list
load panel, titolo") ];
     [ oPanel setPrompt: NSLocalizedString(@"LoadList Prompt", @"DQWC: list
load panel, prompt") ];
    // niente directory, un solo file alla volta
    [ oPanel setCanChooseDirectories:NO ];
    [ oPanel setCanChooseFiles: YES ];
    [ oPanel setAllowsMultipleSelection:NO ];
    [ oPanel setResolvesAliases:YES ];
        [ oPanel beginSheetForDirectory:nil file:nil types: [ NSArray
arrayWithObject: @"dql" ] modalForWindow: [ self window]
                                                    modalDelegate:      self
didEndSelector:@selector(openPanelDidEnd:returnCode:contextInfo:)
contextInfo:nil ];
}

-   (void)openPanelDidEnd:(NSOpenPanel   *)panel   returnCode:(int)returnCode
contextInfo:(void *)contextInfo
{
    if ( returnCode == NSOKButton )
    {
        // ... recupero il nome del file selezionato
        NSArray *filesToOpen = [ panel filenames ];
        NSString *aFile = [ filesToOpen objectAtIndex: 0 ];
        // ed uso il template per costruire la view
        // estraggo i dati dal file
          [ self setTheURLlist: [NSKeyedUnarchiver unarchiveObjectWithFile:
```

```
aFile ]];
        [ self adjustURLList ];
        [ self performCurl ];
        [ theQueue reloadData ] ;
    }
}
```

La logica utilizzata è esattamente la stessa (preparazione del dialogo, esecuzione delle operazioni sottese); da notare che caricare un nuovo file significa buttare via la lista corrente (la variabile `theURLlist` è brutalmente assegnata con i nuovi valori), e non ho neppure inserito un dialogo che informi l'utente della cosa.

CHIUSURA DELL'APPLICAZIONE

Prima di arrivare al cuore dell'applicazione (ovvero, come effettuare il trasferimento), aggiungo qualche parola sulla presenza di un dialogo delle preferenze, in cui appunto l'utente può stabilire alcune caratteristiche dell'applicazione. Ci sono pulsanti di spunta ed altri controlli per stabilire il percorso di default dove salvare i file trasferiti, il comportamento ell'applicazione nei confronti delle (possibili) sovrascritture di file, se la coda dei trasferimento debba essere tenuta pulita (eliminazione automatica dei trasferimenti completati) o meno, se il trasferimento parte automaticamente al lancio dell'applicazione o meno. tutti questi valori sono caricati dalle preferenze all'apertura dell'applicazione, replicati nel dialogo delle preferenze per poterli modificare, riscritti poi nelle preferenze alla chiusura dell'applicazione:

```
- (void)windowWillClose:(NSNotification *)theNotification
{
    NSUserDefaults    * defaults = [NSUserDefaults standardUserDefaults];
    [ defaults setObject: defaultDLDir forKey: UD_DefaultDLFolder ] ;
    [ defaults setBool: checkOverwrite forKey: UD_CheckOverwrite ] ;
    [ defaults setBool: autoCleanQueue forKey: UD_AutoCleanQueue ] ;
    [ defaults setBool: startAtStartup forKey: UD_StartAtStartup ] ;
    [ defaults synchronize];
    [NSApp terminate:self];
}
```

Questo metodo è chiamato quando la finestra dell'applicazione viene chiusa, vuoi per l'uscita dall'applicazione, vuoi perché l'utente la chiude.

Da notare come con questo metodo forzo comunque la chiusura dell'applicazione nel secondo caso.

Capitolo 076 - Continua DownQueue

La libreria cURL

Riprendo adesso `cURL` come applicazione di appoggio per il trasferimento di file. Esplorando meglio il sito, scopro l'esistenza della libreria `libcURL`, che svolge le stesse funzioni di cURL, ma a livello di libreria. In altre parole, non è più necessario utilizzare il meccanismo dei pipe per eseguire l'applicazione da linea di comando e pescare il risultato dell'esecuzione in modo da visualizzarlo. è sufficiente eseguire alcune chiamate a funzioni C rese appunto disponibili dalla libreria stessa. Il comando cURL stesso non è altro che una piccola interfaccia (a linea di comando) costruita attorno alla libreria. La libreria, così come il comando cURL, sono presenti nell'installazione standard del sistema operativo. Tuttavia, per poter utilizzare le funzioni, occorre indicare al progetto XCode di utilizzare tale libreria. Ciò avviene aggiungendo (tramite drag and drop, ad esempio), il file `libcurl.dlyd` (al'interno della cartella `/usr/lib`) all'elenco dei framework del progetto.

Il sito di cURL fornisce abbondante documentazione ed esempi in proposito all'uso. Ho seguito il tutorial presente nel sito ed ho cominciato a costruire una classe, `CurlServer`, che potesse funzionare da thread per il trasferimento dei file.

Per poter inizializzare la classe e la libreria, occorre effettuare un paio di operazioni notevoli:

```
- (id) initWithCaller: (id) localCaller
{
    self = [ super init ];
    [ self setTheCaller: localCaller] ;
    stopDownload = NO ;
    // ne approfitto per inizializzare curl
    curl_global_init(CURL_GLOBAL_DEFAULT);
    curl = curl_easy_init();
    if ( curl )
        return ( self ) ;
    // se non ho curl, e' inutile continuare
    return ( nil ) ;
}
```

Il metodo deve chiamare un paio di funzioni per inizializzare correttamente la libreria. In particolare, la funzione `curl_easy_init` fornisce una variabile di tipo `CURL` che deve essere utilizzare ogni volta che si accede alla funzioni della libreria. È per tanto una variabile d'istanza della classe.

Già che ci sono, il metodo `dealloc` deve chiudere pulitamente la libreria:

```
- (void) dealloc
{
    // chiudo curl
```

```
    curl_global_cleanup();
    [super dealloc];
}
```

La parte più difficile di tutta la faccenda è rappresentata dalle `callback`, ovvero da funzioni che sono chiamate dalla libreria e che devono essere realizzate dall'utilizzatore della libreria per avere qualcosa di funzionante.

Sostanzialmente, occorre scrivere la funzione che si occupa di trattare i dati ricevuti nel trasferimento; se la funzione non fosse presente, i dati ricevuti verrebbero scritti sulla console e quindi sarebbero a tutti gli effetti perduti. Questa funzione deve essere fatta più o meno così:

```
size_t function( void *ptr, size_t size, size_t nmemb, void *stream);
```

La funzione è chiamata da libcURL ogni volta che nel trasferimento sono disponibili dei dati che conviene siano salvati da qualche parte. Sono presenti `size` elementi, presenti in memoria a partire dal puntatore `ptr`, con ciascun elemento di dimensione `nmemb` (in pratica, significa che sono disponibili `size * nmemb` byte). La funzione deve restituire il numero di byte che sono stati effettivamente trattati, che deve coincidere con quelli arrivati, altrimenti il trasferimento si blocca.

Interessante l'ultimo parametro della funzione, il puntatore `stream`, che è a cura dell'utilizzatore della libreria. C'è infatit la possibilità di impostare tale puntatore affinché riconduca ad informazioni di varia natura, ed è di uso strategico in tutta la faccenda. Infatti qui stiamo parlando di funzioni C, che non hanno alcun rapporto con metodi ed altro. Quindi, scrivendo la funzione callback, sarà una normale funzione C e non un metodo della classe `CurlServer`. Questo significa che, ad esempio, non è possibile accedere alle variabili d'istanza della classe. Ecco quindi che posso usare il puntatore `stream` per passare alla funzione callback tutte le informazioni che le occorrono per ben funzionare.

Nella realizzazione minimale. la callback è sufficiente conosca solo il nome del file dove scrivere i dati. Quindi, normalmente, all'interno di `stream` è conservato il puntatore al file. In effetti all'inizio avevo scritto le cose in questo modo; poi però mi sono accorto che sarebbe comodo, all'interno della funzione callback, accedere alla classe `CurlServer`. Alla fine, ho costruito una struttura dati che comprende entrambe le cose:

```
typedef    struct    _cbData {
    CurlServer        * myself ;        // un puntatore all'oggetto
    NSFileHandle    * writtenFile ;    // il puntatore al file corrente
}
CalldackData, * CallbackDataPtr ;
```

Da qualche parte, in preparazione al trasferimento, in uno dei metodi di `CurlServer`, farò più o meno così:

```
    CalldackData    cbData ;
    ...
    cbData.myself = self ;
    cbData.writtenFile = <puntatore al file dove scrivere i dati> ;
```

In questo modo, dall'interno della funzione callback, posso accedere facilmente sia alla classe `CurlServer` sia al file dove scrivere i dati.

Ecco quindi come ho messo in piedi la faccenda:

```
size_t    WriteCallback(
                void    *buffer,        /* puntatore ai dati */
                  size_t     size,              /* dimensione in byte di un
elemento */
                    size_t     nmemb,                /* numero di elementi
presenti*/
                void    * cbDataHdl )    /* struttura dati privata */
{
    unsigned long long    prevSize, delta ;
    // recupero puntatori all'oggetto ed al file
    NSFileHandle    * fh = ((CallbackDataPtr) cbDataHdl )->writtenFile ;
    CurlServer    * ohmy = ((CallbackDataPtr) cbDataHdl )->myself ;
    // se devo fermare tutto, imbroglio curl
    if ( [ ohmy stopDownload ])
        // questo dovrebbe fermare tutto
        return ( 0 ) ;
    // guardo la dimensione corrente del file
    prevSize = [ fh seekToEndOfFile ];
    // scrivo i dati sul file
     [ fh writeData: [ NSData dataWithBytes: buffer length: (size * nmemb)
]];
    // calcolo i byte aggiunti dalla scrittura
    delta = [ fh seekToEndOfFile ] - prevSize ;
    // dovrebbero essere uguali a (size * nmemb); non lo fossero
    // curl automaticamente produce un errore
    return ( delta );
}
```

Ricordo che questa funzione è chiamata direttamente dalla libreria, che ha riempito per me di quattro argomenti della funzione; i primi tre, con i dati giunti per il trasferimento, il quarto, con i dati che ho predisposto io. Nel campo `writtenFile` c'è una variabile `NSFileHandle` che individua il file dove scrivere i dati. Mi posiziono alla fine del file, scrivo i dati, e verifico quanti dati sono stati scritti, in maniera molto semplice. Uso invece le variabili d'istanza dell classe `CurlServer` per, eventualmente, fermare il trasferimento (mi è sembrata questa la maniera più pulita per farlo, ma sono sicuro che ci sono altre strade).

Già che ci sono, ho scritto una seconda funzione callback, non indispensabile, ma molto utile per l'interfaccia utente. Infatti, una funzione fatta più o meno come segue è utile per informare l'utente sullo stato delle cose

```
int          progress( void * stream, double dltotal, double dlnow, double
ultotal, double ulnow) ;
```

Questa funzione è chiamata dalla libreria libcURL periodicamente (diciamo una volta al secondo), sia che ci siano nuovi dati sia che il trasferimento sia fermo per traffico. Il primo argomento è il solito puntatore definito dall'utilizzatore, mentre gli altri quattro argomenti dicono quanti byte `dlnow` sono stati trasferiti in download su un totale di `dltotal`, oppure quanti byte `ulnow` sono stati trasferiti in upload su un totale di `ultotal` (se i dati non sono noti o non pertinenti, valgono zero). La funzione deve ritornare zero se tutto va bene, altro in caso di problemi, con conseguente bloccaggio del

trasferimento).

La funzione che ho scritto è fatta così:

```
int         ProgressFuncCallback(
                void    * cbDataHdl,    /* struttua dati privata */
                double    t,                /* byte totali da download */
                double    d,                /* byte finora ricevuti */
                double    ultotal,        /* byte totali in upload */
                double    ulnow)            /* byte finora inviati */
{
    // recupero puntatori all'oggetto
    CurlServer    * ohmy = ((CallbackDataPtr) cbDataHdl )->myself ;
    // informo il chiamante dello stato avanzamento
    // lo faccio in questo modo per evitare conflitti di grafica coi thread
    [ [ ohmy theCaller] performSelectorOnMainThread:@selector(advanceMsg:)
                        withObject:[  NSArray  arrayWithObjects:  [  NSNumber
numberWithDouble: d], [ NSNumber numberWithDouble: t], nil ]
             waitUntilDone: NO ];
    if ( [ ohmy stopDownload ])
        // questo dovrebbe fermare tutto
        return ( 1 ) ;
    return ( 0 );
}
```

Qui sfrutto pesantemente il fatto che all'interno del parametro `cbDataHdl` è presente un riferimento alla variabile `CurlServer` per far eseguire alcuni metodi. E soprattutto, per dire alla classe che ha fatto partire il tutto (ovvero, il caller di `CurlServer`, quindi `DownQueueWinCtrl`) di eseguire il metodo `advanceMsg` che, si vedrà poi, aggiusta l'aspetto dell'interfaccia mostrando lo stato di avanzamento dei lavori.

GESTIONE DEI TRASFERIMENTI

Ora ce ho scritto le callback, che svolgono il lavoro effettivo, devo solo par partire tutto il meccanismo. Questo avviene all'interno del metodo principale della classe `CurlServer`

```
- (void) getFileWithCurl: (Url2DL*) theItem
{
    CURLcode        res;
    NSFileHandle    * fh ;
    NSString    * finalFile ;
    CalldackData    cbData ;

    // un pool per l'autorelease, visto che siamo in un thread
pool = [[NSAutoreleasePool alloc] init];
    // nome completo del file destinazione
    finalFile = [[ theItem theDestFolder] stringByAppendingPathComponent: [
theItem theName] ];
    // dico al chiamante che sto iniziando
       [  theCaller  performSelectorOnMainThread:@selector(startingDownload)
withObject: nil waitUntilDone: NO ];
    [ theItem setDlStatus: URLSTS_DOWNLOADING ];
    // costruisco il file di destinazione SOVRASCRIVENDO qualsiasi cosa
     // il controllo dovrebbe essere fatto dal chiamante al momento della
scelta del file
    [ [ NSFileManager defaultManager] createFileAtPath: finalFile contents:
nil attributes: nil ];
    fh = [ NSFileHandle fileHandleForUpdatingAtPath: finalFile ];
```

```
    // nel caso di problemi, mi fermo
    if ( fh == nil)
    {
        // informo il chiamante del problema
        [ theCaller performSelectorOnMainThread:@selector(failedDownload:)
                        withObject: NSLocalizedString(@"Fail creating file",
@"curlServer: errore NSFileHandle")
                waitUntilDone: NO ];
    }
```

Poiché questo metodo fa parte di un `thread`, bisogna costruire un `autorelease pool` per gestire gli oggetti in memoria. Dopo di che, le cose filano lisce: si costruisce il nome del file destinazione, e quindi anche il file stesso, producendo un oggetto del tipo `NSFileHandle` (che sarà poi utilizzato dalla funzione callback). Se per qualche motivo la creazione del file non ha successo, si informa la classe chiamante (`DownQueueWinCtrl`) della cosa. Altrimenti, comincia il ballo.

```
    else
    {
        // preparo la struttura dati
        cbData.myself = self ;
        cbData.writtenFile = fh ;
        // preparo le opzioni per il comando curl
        // lo url da scaricare
        curl_easy_setopt(curl, CURLOPT_URL, [[ theItem theURL] UTF8String ]
);
        // la funzione di callback per la scrittura dei dati
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void*) & cbData );
        // opzioni per lo stadio avanzamento
        curl_easy_setopt(curl, CURLOPT_VERBOSE, FALSE);
        curl_easy_setopt(curl, CURLOPT_NOPROGRESS, FALSE);
        // la funzione di callback per lo stato di avanzamento
                        curl_easy_setopt(curl,   CURLOPT_PROGRESSFUNCTION,
ProgressFuncCallback);
        curl_easy_setopt(curl, CURLOPT_PROGRESSDATA, (void*) & cbData );
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, TRUE );
        // eseguo le operazioni di download
        res = curl_easy_perform(curl);
```

Qui preparo la struttura dati per le funzioni callback; poi, configuro il comportamento di cURL attraverso una successione di chiamate alla funzione `curl_easy_setopt`. Sono così individuati, nell'ordine: lo url del file da trasferire (che va trasformato in una stringa masticabile dalla libreria, che è in C, e non come `NSString`, tipico di Cocoa); la funzione di callback per la scrittura dei dati, e la predisposizione del quarto argomento della funzione stessa; si evita la scrittura di informazioni non necessarie sullo standard output (`CURLOPT_VERBOSE`) e si inibiscono le funzioni interne alla libreria per tenere conto dell'avanzamento lavori (`CURLOPT_NOPROGRESS`); infatti, l'avanzamento lavori è gestito direttamente attraverso la funzione di callback ed il suo argomento; infine, impostando a vero il parametro `CURLOPT_FOLLOWLOCATION` si dice alla libreria di seguire gli eventuali reindirizzamenti dello Url principale (cosa che ho scoperto accadere spesso quando scarico i file che mi interessano; cosa sia e come funzioni il reindirizzamento, bisognerebbe chiederlo a qualcuno più esperto di me nei protocolli di rete).

A questo punto, la chiamata alla funzione `curl_easy_perform` esegue magicamente

il tutto, ovvero fa partire il trasferimento. La funzione continua a lavorare; chiama periodicamente la funzione `ProgressFuncCallback` k per comunicare lo stato di avanzamento, e la funzione `WriteCallback` per la scrittura dati quando ne sono disponibili di nuovi; termina quando l'intero processo di trasferimento si è concluso(oppure si è fermato in errore per qualche motivo).

```
        // verifico il risultato
        if(CURLE_OK != res)
        {
              NSString     * locerr = NSLocalizedString( @"Fail downloading
file", @"curlServer: risultato curl non ok");
              // mi sono fermato per qualche motivo, ristabilisco lo stato
dell'elemento
            [ theItem setDlStatus: URLSTS_PARTIAL ];
            // se non ho fermato volontariamente
            // oppure sono fermo ma con errori strani
            if (stopDownload == NO)
            {
                // informo il chiamante
                                                                [    theCaller
performSelectorOnMainThread:@selector(failedDownload:)
                          withObject: [ NSString stringWithFormat: @"%@: %d :
%s", locerr, res, curl_easy_strerror(res)]
                       waitUntilDone: NO ];
                [ theItem setDlStatus: URLSTS_DWLDERROR ];
            }
                      else  if  (  (res  !=  CURLE_WRITE_ERROR)  &&  (res  !=
CURLE_ABORTED_BY_CALLBACK) )
            {
                // informo il chiamante
                                                                [    theCaller
performSelectorOnMainThread:@selector(failedDownload:)
                          withObject: [ NSString stringWithFormat: @"%@: %d :
%s", locerr, res, curl_easy_strerror(res)]
                       waitUntilDone: NO ];
                [ theItem setDlStatus: URLSTS_DWLDERROR ];
            }
        }
```

Quando si esegue questo pezzo di codice, significa che la funzione `curl_easy_perform` che ha gestito il trasferimento è terminata in uno stato di errore. La cosa potrebbe essere stata voluta dall'utente, oppure dovuta ad un oggettivo errore nel processo di trasferimento. Qui ho distinto i due casi, ma le operazioni eseguite sono le stesse (la chiamata del metodo `failedDownload` di `DownQueueWinCtrl`).

```
        // negli altri casi, e' andato tutto bene
        else
            // metto a posto le info dell'oggetto
            [ theItem setDlStatus: URLSTS_FINISHED ];
```

Questo pezzo di codice invece è eseguito quando il trasferimento è andato a buon fine. Mi limito a cambiare lo stato dell'elemento.

```
        // se ho un file aperto, lo chiudo in ogni caso
        if( fh )
            [ fh closeFile ];
    }
    // fine dello scaricamento, ed anche del thread
```

```
        [  theCaller  performSelectorOnMainThread:@selector(endingDownload)
withObject: nil waitUntilDone: NO ];
    [ pool release ];
    [ NSThread exit ] ;
}
```

Le operazioni conclusive sono la chiusura del file (che altrimenti rimarrebbe aperto provocando qualche scompiglio a livello di sistema operativo) e la chiusura del thread.

GESTIONE DELLE OPERAZIONI

Una parte che mi ha fatto penare un po' è stata la comunicazione tra il thread del trasferimento ed il thread principale della finestra. Intanto, il metodo di `DownQueueWinCtrl` da cui parte tutto è `performCurl`. è chiamato senza argomenti ogni volta che la lista dei trasferimenti è modificata in qualche modo per aggiunta di elementi) oppure, all'inizio delle operazioni (all'interno di `awakeFromNib`)

```
- (void)
performCurl
{
    Url2DL            * item2dl ;
    unsigned short    ii ;

    // se il download e' fermo, faccio nulla
    if ( downloadStatus == NO )
    {
        [ self saveListStatus ];
        return ;
    }
    // spazzolo tutti i vari volumi interessati
    for ( ii = 0 ; ii < [ theURLlist count]; ii ++ )
    {
        // recupero l'elemento
        item2dl = [ theURLlist objectAtIndex: ii ];
        // e a seconda del suo stato...
        switch ( [ item2dl dlStatus] ) {
        case    URLSTS_FINISHED :            // gia' scaricato
        case    URLSTS_SUSPENDED :            // sospeso
        case    URLSTS_DWLDERROR :            // in errore
            break ;    // procedo col successivo
        case    URLSTS_DOWNLOADING :        // in corso di scaricamento
            currItem = item2dl ;    // me lo segno
            return ;        // ho finito
          case     URLSTS_WAITING :               // in attesa di essere
scaricato
        case    URLSTS_PARTIAL :            // in parte scaricato
            currItem = item2dl ;    // me lo segno
            // ripulisco i dati
            [ self resetDownloadData: [ item2dl theName ] ] ;
            // faccio partire curl
            theCurlServer = [[CurlServer alloc] initWithCaller: self] ;
            [NSThread detachNewThreadSelector:@selector(getFileWithCurl:)
                toTarget: theCurlServer withObject: item2dl ];
            // salvo i dati
            [ self saveListStatus ];
            return ;
        }
    }
```

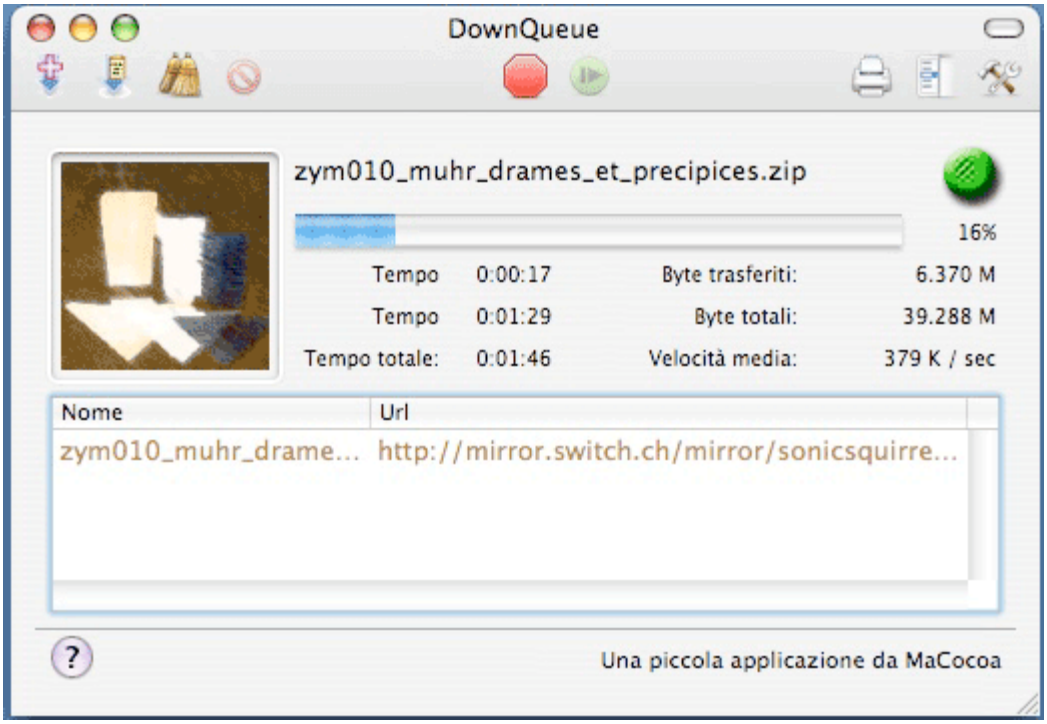
```
}
```

Il metodo in pratica si occupa di rendere valida la variabile d'istanza `currItem`, in modo da sapere quale sia il file in trasferimento corrente (se uno deve essercene; le operazioni potrebbero essere sospese, come si vede dal test sul flag `downloadStatus`). Sono esaminati tutti gli elementi nella lista. Se uno è individuato come in corso di trasferimento, la cosa finisce lì. Se si arriva ad un elemento che sta attendendo il trasferimento, si costruisce un nuovo thread a partire dalla classe `CurlServer`, e si fanno partire le operazioni (descritte sopra).

La comunicazione tra i due thread avviene in pratica attraverso quattro metodi

```
- (void) advanceMsg: (id) statusArg ;
- (void) startingDownload ;
- (void) endingDownload ;
- (void) failedDownload: (NSString *) errMsg ;
```

A suo tempo, avevo chiamato un gruppo di metodi di questo tipo un protocollo. Sempre a suo tempo, non mi ero preoccupato troppo della faccenda, visto che tutto sembrava funzionare correttamente. Inoltre, pur avendolo scritto (nel capitolo 63), non ho fatto troppa attenzione a come due thread diversi si comportassero quando tentavano di aggiornare l'interfaccia grafica in concorrenza.



Ma andiamo per ordine. Illustro l'interfaccia che tiene conto dello stato di avanzamento dei lavori. Ci sono molte informazioni da presentare: il nome del file in corso di trasferimento, o comunque primo nella coda. Una specie di semaforo che indica, col rosso, trasferimento non attivo, col verde, trasferimento in corso, col giallo, trasferimento attivo, ma non ci sono file da trasferire. Durante un trasferimento, poi, sono gestite le seguenti informazioni: il tempo trascorso dalla partenza del trasferimento ed i byte effettivamente trasferiti; se è disponibile poi l'informazione sulla quantità totale di byte da trasferire (anche qui, una informazione che non sempre è disponibile,

penso dipenda da come i server gestiscano la comunicazione), questa quantità totale di byte, assieme ad una stima del tempo necessario a trasferire quelli che restano; una stima del tempo totale di trasferimento; una misura, approssimativa fin che si vuole, ma utile, della velocità di trasferimento. Vediamo come ottenere tutte queste informazioni.

Il punto di partenza è la funzione callback sullo stato di avanzamento. Questa è chiamata periodicamente dalla libreria, ed ha a disposizione il numero di byte trasferiti e quelli totali. Con questi due soli valori (ed un po' di aiuto dal sistema operativo) riesco a produrre tutto il resto.

In primo luogo, quando faccio partire il trasferimento, faccio in modo che `CurlServer` chiami il metodo

```
- (void) startingDownload
{
    // metto da parte il tempo iniziale
    downLoadStartTime = TickCount( );
    [ graphicStatus setImage: [ NSImage imageNamed:@"stsVerde"]];
}
```

Qui metto da parte il tempo corrente del sistema (in `tick`, ovvero sessantesimi di secondo) attravero l'utile (e di Carbon) funzione `TickCount`. Metto anche l'immagine semaforica a verde, indicando trasferimento in corso.

Il grosso del lavoro è svolto dal metodo `advanceMsg`, che ha come argomento un array con i due valori di byte totali e byte finora trasferiti

```
- (void) advanceMsg: (id) statusArg
{
    NSArray    * status = (NSArray*) statusArg ;
    double    sts = [[ status objectAtIndex:0] doubleValue] ;
    double    tot = [[ status objectAtIndex:1] doubleValue] ;
     // tempo trascorso dall'inizio del download, in tick (circa 1/60 di
secondo)
    UInt32    deltatime = TickCount() - downLoadStartTime ;
    // numero di byte finora scaricati
    [ downCurrSize setFloatValue: sts ] ;
    // imposto il tempo trascorso
    [ elaspedTime setFloatValue: (deltatime / 60.0f) ];
    // calcolo la velocita' media
    [ averageSpeed setFloatValue: (sts * 60.0f / deltatime ) ];
    // se ci sono entrambi i dati
    if ( tot && sts)
    {
        // calcolo del tempo residuo
         [ residueTime setFloatValue: ((deltatime / 60.0f)*( tot / sts - 1
)) ];
        // ed il tempo totale
        [ totalTime setFloatValue: ((deltatime / 60.0f)*( tot / sts )) ];
    }
    else
    {
        // manca la dimensione totale, dico che so nulla
        [ residueTime setStringValue: NSLocalizedString(@"resTime Unknown",
@"DQWC: advance info, tempo residuo")];
         [ totalTime setStringValue: NSLocalizedString(@"totTime Unknown",
```

```
@"DQWC: advance info, tempo totale")];
    }
    // se e' nota la dimensione complessiva
    if ( tot )
    {
        // imposto la dimensione complessiva e la barra di avanzamento
        [ downTotSize setFloatValue: tot ] ;
        [ currentStatus setIndeterminate: FALSE ];
        [ currentStatus setDoubleValue: sts/tot ];
        [ downPercent setStringValue: [NSString stringWithFormat: @"%3d%%",
(short)(( 100.0*sts / tot) + 0.5F)]];
    }
    else
    {
        // non so che dire
        [ downTotSize setStringValue: NSLocalizedString(@"totsize Unknown",
@"DQWC: advance info, numero totale byte") ] ;
        [ currentStatus setIndeterminate: TRUE ];
        [ downPercent setStringValue: @"???" ];
    }
}
```

Il metodo è piuttosto lungo e noioso, ma lineare nella sua sua struttura. Anche se teoricamente chiamato periodicamente (una volta al secondo) in base a quanto stabilito da libCurl, la prima operazione consiste nel vedere quanto tempo è trascorso dall'inizio del trasferimento. Una volta convertito in secondi, ho già pronta l'informazione sul tempo trascorso. E dividendo il numero di byte trasferiti con il tempo trascorso, ecco che ho una indicazione della velocità media di trasferimento.

Se poi è nota (diversa da zero) la dimensione totale del file da trasferire, si possono anche fare le stime delle altre grandezze. Con una semplice proporzione ("tempo trascorso" sta a "tempo totale" come "byte trasferiti finora" stanno a "byte totali da trasferire") si ottiene una stima sul tempo totale necessario. Togliendo dalla stima del tempo totale il tempo trascorso, si ha il tempo rimanente.

Con tutto ciò, è possibile anche calcolare la percentuale di avanzamento, ed utilizzarla per animare una simpatica barra di avanzamento.

Quando il trasferimento è terminato, si chiama il metodo seguente, che ripulisce le cose:

```
- (void) endingDownload
{
    // aspetto un tot per mettere a posto la grafica
    // ripulisco i dati
    [ graphicStatus setImage: [ NSImage imageNamed:@"stsGiallo"]];
      [ self resetDownloadData: NSLocalizedString(@"nothing to download",
@"DQWC: advance info, stringa file default") ] ;
    if ( autoCleanQueue )
    {
        if ( [ currItem dlStatus] == URLSTS_FINISHED )
            [ theURLlist removeObject: currItem ];
        [ theQueue reloadData ] ;
    }
    [ self saveListStatus ];
    // nessun elemento corrente
    currItem = nil ;
    // aggiorno un po' tutto
```

```
    [ self performCurl ];    // e riparto con curl
    if ( [ theDrawer state ] == NSDrawerOpenState )
        [ self refreshDrawer ];
}
```

L'immagine del semaforo è rimessa a gialla, si ripuliscono i campi del trasferimento, e si rilancia `performCurl`. In questo modo, se ci sono altri elementi in coda, si passa a quello successivo.

Vengo ora a spiegare il problema della grafica. In un primo momento, questi metodi erano chiamati direttamente dalla classe `CurlServer`. Ad esempio, nella funzione callback `ProgressFuncCallback` avevo scritto

```
    [ [ ohmy theCaller] advanceMsg:[ NSArray arrayWithObjects: [ NSNumber
numberWithDouble: d], [ NSNumber numberWithDouble: t], nil ] ];
```

mentre nel codice visto sopra la chiamata è più complicata:

```
    [ [ ohmy theCaller] performSelectorOnMainThread:@selector(advanceMsg:)
                        withObject:[ NSArray arrayWithObjects: [ NSNumber
numberWithDouble: d], [ NSNumber numberWithDouble: t], nil ]
            waitUntilDone: NO ];
```

Qual è la differenza? Nel primo caso, il metodo, benchè della classe `DownqueuWinCtrl`, è chiamato all'interno del thread associato alla classe `CurlServer`. Questo significa che ci sono due thread separati che intervengono sull'interfaccia utente, o meglio sulla gestione della grafica e del ridisegno delle finestre. Questo, come già accennato in precedenza, non va bene, in quanto le classi dello `Application Kit` non sono rientranti (non possono essere chiamate contemporaneamente da thread differenti). Se le cose non sono troppo complicate, non ci sono problemi evidenti. Tuttavia, non appena si comincia ad avere un po' di finestre, interazioni varie, eccetera, cominciano i problemi, non tanto di funzionalità, quando di cattivi ridisegni dell'interfaccia (i thread sono eseguiti in tempi differenti, talvolta prima uno poi l'altro, e visto che entrambi disegnano qualcosa sullo schermo, a seconda di chi arriva prima, si hanno disegni diversi). La soluzione, semplice a dirsi, è di fare in modo che tutte le operazioni che hanno a che fare con lo schermo e con le finestre siano svolte da un unico thread, quello principale. Questo solleva il problema di come modificare l'aspetto dell'applicazione da parte dei thread secondari (che spesso, come nel mio caso, sono loro che svolgono il lavoro interessante). Allo scopo viene in aiuto proprio il metodo `performSelectorOnMainThread` (nelle varie incarnazioni), il quale aggiunge il metodo indicato dagli argomenti all'interno delle operazioni da effettuare nel main thread. Detto così è piuttosto semplice, ma ho penato parecchio prima di arrivare a questa soluzione.

TOCCHI FINALI (SI FA PER DIRE)

A chiudere l'esposizione dell'interfaccia, noto che ad ogni campo (tempo e dimensioni) che presenta informazioni sul trasferimento in corso ho associato un formatter, in modo da lavorare all'interno della classe non numeri puri, e lasciare appunto ai formatter il compito di presentare in bella vista i dati. Non vale nemmeno la pena di presentare il codice, dato che è molto semplice ed in qualche caso un bel riciclo da formatter utilizzati

in precedenza.

Infine, c'è da aggiustare l'aspetto della toolbar e dei menu a seconda dello stato corrente del trasferimento e della selezione corrente.

È compito di due metodi, simili nello svolgimento ma diverse nelle prescrizioni, che riporto per la sola toolbar.

Ci sono dei protocolli informali che permettono di selezionare lo stato di attivo o disattivo di un elemento di una toolbar, o di una voce di menu: nel caso della toolbar, abbiamo il metodo `validateToolbarItem`

```
- (BOOL)
validateToolbarItem: (NSToolbarItem *) toolbarItem
{
    // il pulsante di stop
if ([[toolbarItem itemIdentifier] isEqual:TBItId_StopAll])
    {
        // riflette lo stato di download
        return ( downloadStatus ) ;
    }
    // in pulsantedi resume
else if ([[toolbarItem itemIdentifier] isEqual:TBItId_Resume])
    {
        // e' il rovescio del precedente
        return ( downloadStatus == NO );
}
    // la cancellazione di un elemento
else if ([[toolbarItem itemIdentifier] isEqual:TBItId_DeleteItem])
    {
        // funziona solo se ci sono elementi selezionati
        return ( ( [ theQueue selectedRow] != -1 ) );
}
    // tutti gli altri sono abilitati
    return ( YES );
}
```

Il metodo restituisce `YES` se il pulsante della toolbar deve essere attivo, `NO` altrimenti. Con questo metodo si dovrebbero intercettare a volo tutte le situazioni che richiedono diverse condigurazioni dei pulsanti, senza dover necessariamente abilitare/disabilitare esplicitamente i pulsanti quando cambiano le condizioni operative.

Come simpatica caratteristica dell'applicazione, ecco un metodo per aprire una pagina web, che ho collegato ad una voce di menu, in modo che dall'intero dell'applicazione si possa accedere al sito di Macocoa:

```
- (IBAction)goToSite:(id)sender
{
      [ [ NSWorkspace sharedWorkspace] openURL: [ NSURL URLWithString:
@"http://www.macocoa.omitech.it"]] ;
}
```

A completamento dell'applicazione, ho realizzato una cosa che solo qualche ottimista inguaribile potrebbe definire `help` (ed in più solo in italiano). Ho scritto qualche pagina in HTML, facendo bene attenzione ad utilizzare quanto Apple consiglia nella realizzazione. La pagina principale dello Help contiene un meta tag del tipo:

```
    <meta name="AppleTitle" content="DownQueueHelp" />
```

Con questo, ho stabilito la presenza di un libro (`book`) di help denominato `DownQueueHelp`. Devo tenere presente questo identificativo, perché, ad esempio, ogni collegamento interno ai file dello Help sono scritti in questa maniera bizzarra:

```
<a href="help:anchor='themainpage' bookID=DownQueueHelp">DownQueue</a>>
```

A parte queste stranezze specifiche del meccanismo di Help della Apple, le pagine sono scritte in un bel codice HTML moderno, ovvero con un foglio di stile in CSS separato dalla pagina html vera e propria. Io ho scritto tutto a mano, ma sospetto che si possa usare facilmente un moderno editor grafico.

Una volta realizzato il contenuto dello Help, bisogna pensare ad associarlo all'applicazione. In primo luogo, occorre compilare l'indice dello help, utilizzando l'apposita applicazione `Help Indexer` fornita coi Developer Tools.

Poi, ho fatto in modo che ogni finestra dell'applicazione (o quasi) presenti nell'angolo in basso a sinistra un pulsante con punto di domanda. Facendo clic su questo pulsante, si chiama un metodo centralizzato (cioé, su `DownQueueWinCtrl`) che apre la pagina appropriata dello Help.

```
- (IBAction)contextHelp:(id)sender
{
                NSString    *locBookName    =    [[NSBundle    mainBundle]
objectForInfoDictionaryKey: @"CFBundleHelpBookName"];
```

```
    switch ( [sender tag] ) {
    case 20:    // main page
            [[NSHelpManager sharedHelpManager] openHelpAnchor:@"mainwindow"
inBook:locBookName];
        break ;
    case 21:    // drawer
             [[NSHelpManager sharedHelpManager] openHelpAnchor:@"thedrawer"
inBook:locBookName];
        break ;
    case 30:    // preference
            [[NSHelpManager sharedHelpManager] openHelpAnchor:@"prefwindow"
inBook:locBookName];
        break ;
    case 40:    // addfile
              [[NSHelpManager sharedHelpManager] openHelpAnchor:@"addfile"
inBook:locBookName];
        break ;
    default :
        break ;
    }
}
```

Tutto ciò realizza una sorta di aiuto contestuale (per lo meno, relativamente alle finestre) certamente molto amichevole.

LOCALIZZAZIONE

Ho cercato di localizzare l'applicazione nelle due lingue, italiano ed inglese, che potrebbero essere utili agli utilizzatori di questa applicazione, ma soprattutto per rinfrescarmi le operazioni da fare.

In primo luogo, all'interno del codice, ogni volta che utilizzo una stringa suscettibile di localizzazione, non lo faccio direttamente, ma uso un costrutto piuttosto brutto a vedersi ma efficace nella sua realizzazione. Ad esempio, nel costruire un dialogo di avvertimento:

```
         [alert addButtonWithTitle: NSLocalizedString(@"Overwrite", @"DQWC:
alert overwrite, overwrite btn")];        // sovrascirvo
        [alert addButtonWithTitle: NSLocalizedString(@"AutoRename", @"DQWC:
alert overwrite, autorename btn")];        // piglio un nome a caso
           [alert addButtonWithTitle: NSLocalizedString(@"Cancel", @"DQWC:
alert overwrite, cancel btn")];            // lascio stare
         [alert setMessageText: NSLocalizedString(@"Overwrite the existing
file?", @"DQWC: alert overwrite, msgTxt")];
         [alert setInformativeText: NSLocalizedString(@"The content of the
previous file is lost.", @"DQWC: alert overwrite, infoTxt")];
```

La funzione `NSLocalizedString` serve appunto a caricare da un file apposito la versione nella lingua corrente dell'espressione indicata nel primo argomento. Il secondo argomento è invece una sorta di memento sulla natura della stringa, utilizzato spesso da chi deve tradurre le stringhe nella versione locale (che non sempre è lo sviluppatore, e che quindi non sa bene a cosa serve quella stringa).

L'utile comando (standard Unix, almeno così mi pare) `genstring` è utilizzabile per estrarre dai file dell'applicazione tutte le occorrenze di `NSLocalizedString` e di riunirle in un apposito file. Ad esempio il comando

```
genstrings -o English.lproj *.m
```

Raccoglie nel file `Localizable.strings` tutte le stringhe estratte dei file `<qualcosa> .m` (quindi, il codice dell'applicazione, se tutti i file .m dell'applicazione si trovano nella directory in cui eseguo il comando). Questo file è costruito in questo modo:

```
...
/* DQWC: alert overwrite, autorename btn */
"AutoRename" = "AutoRename";
...
/* DQWC: alert overwrite, overwrite btn */
"Overwrite" = "Overwrite";
...
```

dove appunto sono riportate (in ordine alfabetico) gli identificatori delle stringhe da tradurre; il secondo argomento di `NSLocalizedString` è messo come commento nella riga precedente.

Adesso, il traduttore deve semplicemente trasformare la parte a destra del segno di uguale nella stringa da visualizzare nella lingua desiderata. Ad esempio, in italiano ho una cosa del tipo:

```
...
/* DQWC: alert overwrite, autorename btn */
"AutoRename" = "Auto-Rinomina";
...
/* DQWC: alert overwrite, overwrite btn */
"Overwrite" = "Sovrascrivi";
...
```

Per quanto riguarda invece finestre e quant'altro, anche qui sono disponibili una serie di strumenti. Il punto di partenza è la versione iniziale (ad esempio, in inglese) del file `nib`. Nella mia applicazione, ce ne sono tre (la principale, l'aggiunta di un file, le preferenze). Di tutte, costruisco la versione localizzata in italiano attraverso le informazioni del nib, all'interno di XCode. Così facendo, sono costruiti tre file nib all'interno della cartella `Italian.lproj`, del tutto uguali alla versione inglese. Adesso, bisogna tradurre tutte le stringhe (ed eventualmente spostare, aggiustare campi, eccetera).

È disponibile un simpatico comando da terminale, `nibtool`, che può essere utilizzato per semplificare il lavoro. Scrivendo (all'intero della cartella Italian.lproj)

```
nibtool -L ../English.lproj/MainMenu.nib >./MainMenu.strings
nibtool -L ../English.lproj/SetDLWindow.nib >./SetDLWindow.strings
nibtool -L ../English.lproj/PrefWin.nib >./PrefWin.strings
```

sono generati tre file che contengono le stringhe presenti all'interno del file nib. Ad esempio, il contenuto assomiglia alle righe seguenti:

```
...
/* NSMenu : <title:action> (oid:341) */
"Action" = "Action";
```

```
/* NSMenuItem : <title:add from clipboard> (oid:223) */
"Add from Clipboard" = "Add from Clipboard";

/* NSMenuItem : <title:add...> (oid:82) */
"Add..." = "Add...";
...
```

Con un editor di testi (XCode stesso va benissimo) si possono modificare le stringhe alla destra del segno di uguale, traducendole nella lingua desiderata:

```
...
/* NSMenu : <title:action> (oid:341) */
"Action" = "Azioni";

/* NSMenuItem : <title:add from clipboard> (oid:223) */
"Add from Clipboard" = "Aggiungi dagli Appunti";

/* NSMenuItem : <title:add...> (oid:82) */
"Add..." = "Aggiungi...";
...
```

A questo punto, si ritorna al comando `nibtool` per effettuare l'operazione inversa

```
nibtool -d MainMenu.strings ../English.lproj/MainMenu.nib -W MainMenu.nib
nibtool   -d   SetDLWindow.strings   ../English.lproj/SetDLWindow.nib   -W
SetDLWindow.nib
nibtool -d PrefWin.strings ../English.lproj/PrefWin.nib -W PrefWin.nib
```

Il comando unisce il file `English.lproj/MainMenu.nib` sostituendo le stringhe che trova in `MainMenu.strings` e costruisce un nuovo file `MainMenu.nib`. Magari qualcosa bisogna aggiustare ancora (perché l'italiano è una lingua che occupa più spazio dell'inglese, occorre verificare tutti i campi di testo, che riescano a contenere la stringa), ma l'operazione è piuttosto semplice ed efficace per avere una versione in lingua dell'interfaccia.