MaCocoa: Indice

MaCocoa: 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.

EcoSito

Troverete questo sito piuttosto scarno (è un eufemismo). Si tratta di un sito ecologico, senza grafica, fronzoli, e abbellimenti vari, teso al risparmio consapevole di banda di trasmissione. Sto mentendo: la mia scrittura è talmente prolissa che la maggior parte sarà da buttare...
Nella sezione Download, tuttavia, troverete dei file da scaricare: servono a facilitare chiunque intenda utilizzare il sito. I file raccolgono, compressi, tutti file del sito, in modo da evitarvi la seccatura di scaricare singolarmente tutti i file che lo compongono. Di più: per chi ha già scaricato una versione precedente, sono comodamente presenti solo gli aggiornamenti.

n.d.r.

Sappiate che né l'autore del sito e titolare di tutti i concetti contenuti nel testo né tantomeno lo scrivente facchino (atresp@libero.it) che si è preoccupato di incartare il tutto nella presente confezione regalo saranno mai disposti ad assumersi una qualsiasi responsabilità derivante da malori, cambi di umore improvviso, malanni, malattie croniche, indisposizioni, varie ed eventuali, causate al vostro amato ***Mac****.*

Siate consapevoli quindi che tutto ciò che farete sarà a vostro chiaro e indiscutibile rischio e pericolo, perciò non vi restano che queste due possibilità:

Declino: buttate 200 pagine fresche di stampa.

Accetto: voltate e assumetevi le ***vostre*** *responsabilità*

Consultate il sito per visionare eventuali aggiornamenti e scaricare i files prodotti nei vari capitoli.

Buona lettura

MaCocoa: Prima di partire

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.
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.

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 riassunto 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 vedete 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.

MaCocoa: Il Sistema Operativo

System Overview

Programmare per Mac OS X richiede almeno una minima conoscenza del sistema operativo. Senza pretendere di essere esaustivo e corretto, rimando i più coraggiosi al documento SystemOverview, che trovate in pdf e semplice html
Ma anche pigliando il centro aiuti, e poi il centro sviluppatori, ci arrivate velocemente.
Come ogni sistema operativo che si rispetti, Mac Os X possiede un modello a cipolla, in cui strati successivi si sovrappongono l'uno sull'altro. Al livello più basso abbiamo il **Kernel** (nucleo). Il kernel è la parte principale di **Darwin**, la parte fondamentale del sistema operativo su cui poggiano tutte le altri parti. Darwin è costituito da **Mach**, **BSD**, dai driver per i vari device hardware presenti, il gestore del file system ed il gestore del networking. Apple ha deciso di rendere il tutto open source, per cui è possibile trovare i sorgenti, modificarli, ricompilarli, installarli modificati al posto di quelli forniti da Apple, e così via.
Non mi discosto troppo dal vero se affermo che il kernel è sostanzialmente la parte fondamentale di un sistema operativo simil-Unix. Non per nulla Unix e il kernel di Mac Os X sono costruiti a partire dalle stesse tecnologie; anzi, Mach e BSD sono alla base di effettivi sistemi Unix.
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.
Sopra ancora, ci sono gli **Application Services**. Qui si trovano tutte le funzioni legate all'interfaccia grafica ed alla gestione degli eventi. Al suo fianco, QuickTime, che fornisce appunto servizi alle applicazioni, anche se ad un livello più elevato degli Application Services.
Siamo finalmente arrivati al livello più alto, formato da cinque ambienti in cui possono essere eseguite le applicazioni: **Classic**, **Carbon**, **Cocoa**, **Java** e **BSD**. Questi *Application Environment* 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 Classic, un terzo Carbon. Io invece ho scelto i mattoncini di tipo Cocoa.
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 Programming Interface), 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 cinque! Ho mentito, sono sole tre. Mentre Carbon, Cocoa e Java sono in grado di costruire applicazioni sostanzialmente simili, quelle con interfaccia simil-Mac (nella nuova versione, Aqua) e quant'altro, gli ambienti Classic e BSD sono adatti solo per particolari applicazioni.

Gli ambienti operativi

L'ambiente BSD

Liquidiamo subito BSD: è l'interfaccia a linea di comando, e che va ad interagire direttamente con il kernel, dove trova appunto gli agganci per eseguire le funzioni di basso livello. Si utilizza l'ambiente BSD per fare applicazioni magari complicate ed interessanti ma di alto livello. Ad esempio, molti porting da ambiente Unix o simile utilizzano in massima parte questo ambiente. L'ambiente applicativo BSD è insomma quello classico di Unix (più o meno, mi pare che non sia Posix, dove Posix è uno standard che definisce le API di un sistema operativo simil-unix).

L'ambiente Classic
Mi sbrigo velocemente anche con Classic. Si tratta di un ambiente all'interno del quale funzionano le vecchie applicazioni Macintosh. Non c'è ragione per cui qualcuno debba sviluppare in ambiente Classic. Fra qualche anno, quando non ci sarà più bisogno dell'ambiente Classic perché tutti i programmi saranno stati aggiornati per Os X, Classic scomparirà (è un po' buffo trattare così sbrigativamente l'ambiente Classic, che nel bene e nel male ci ha accompagnato dal 1984 ad oggi... 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.

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 (c'è un pdf preliminare ed incompleto che documenta tutti questi elementi: sono due volumi da mille pagine ciascuno!).
Non si può pensare di studiarli tutti e poi cominciare a sperimentare. Conviene impadronirsi di qualche elemento, e poi cominciare subito. Quando avremo bisogno di un qualche costrutto particolare, prima di farcelo da soli, converrà perdere 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 ci occorre, ma almeno qualcosa di simile (e che potrà essere modificato alla bisogna).La maggior parte dei nomi (non sono in grado di dire se la totalità, ma finora non ho trovato esempi contrari) 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.

Strati di Cocoa

Cocoa è un ambiente applicativo costituito da tre strati. Il primo strato è **Objective C**, un linguaggio di programmazione. ObjC non è parte di Cocoa, ma Cocoa è basato fortemente su tale linguaggio; gli oggetti di Cocoa sono scritti appunto in ObjC. Il secondo strato è chiamato **Foundation Framework**, ed il terzo strato è l'**Application 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.
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.
Scendendo di un livello di dettaglio, gli elementi si possono raggruppare in diverse famiglie, come le seguenti:

- gestione dei dati, per gestire collezioni di numeri, di stringhe, supporto generico alle strutture dei dati
- testo e stringhe, per la gestione dei testi. Da notare che Cocoa gestisce caratteri in accordo alla specifica Unicode, e quindi è in grado di gestire con facilità alfabeti con latini ed altre complicazioni del genere
- date e ora, per gestire quelle particolari strutture dati legate al tempo
- coordinamento applicazioni, per scambiare messaggi tra applicazioni ed anche all'interno dell'applicazione stessa
- creazione e dismissioni di elementi, gestione della memoria; una cosa importante, per evitare che un programma impazzito causi problemi ad altri programmi (la qualità della gestione della memoria è spesso un indice di un buon sistema operativo)
- distribuzione e persistenza degli elementi, con le funzioni necessarie per la memorizzazione dei dati e la distribuzione degli stessi ad esempio attraverso una rete
- servizi propri del sistema operativo, con la gestione del file system, gestione delle applicazioni, eccetera.

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

MaCocoa: Programmazione Object Oriented

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, dal 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'operazione 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 un insieme di dati (che so, una

lista di indirizzi). Esiste allora la struttura dati generali (una rappresentazione opportuna della lista di indirizzi) e tre moduli funzionali separati, uno per la lettura dei dati ed il riempimento della struttura, uno per effettuare l'ordinamento di questi dati e 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 modifica, 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 sull'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 strategia 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**: deve fornire 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**: deve consentire 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**: consentire 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 è effettivamente realizzata deve poter essere decisa all'ultimo momento utile, in maniera dinamica, perché nel frattempo qualcosa potrebbe essere cambiato (ma di questo concetto, qui mi rendo 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 OO 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 OO 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 OO su uno strato essenzialmente procedurale, e riesce a mescolare malamente le due cose.

MaCocoa: Programmazione Object Oriented

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 nell'ambiente operativo del programma. Sono costanti molti oggetti propri del linguaggio e sempre disponibili. Altrimenti, e 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 OO: 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 OO 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 OO 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 (o 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 OO 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, facciamo 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 OO 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 uguale o più grande rispetto la 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 OO 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 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.

MaCocoa: Objective C

Objective C

Objective C è una estensione del linguaggio C con alcuni costrutti aggiuntivi per poter programmare secondo la metodologia OO. 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 OO "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 odi 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:
`: argomento : argomento : 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 vi 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. 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.
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...

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. 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 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 mangiaodore 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 *teoria 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...

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...

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 realizza i 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 il lavoro non vale la candela. Quindi, mettiamo sempre il nome della superclasse. Se proprio vogliamo avere classi figlie di nessuno, mettiamo almeno `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 noioso per ottenere lo stesso risultato senza la direttiva `#import`, ma visto che c'è questa direttiva, non ne parlo).

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 usa; 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 come 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...

L'ambiente di sviluppo

Project Builder

Il programma che ci interessa si chiama Project Builder. È fornito di serie con Mac OS X, ma bisogna installare i Developers Tools. Non ho trovato alcuna difficoltà a fare tutto ciò: sui cd che costituiscono il prodotto Mac OS X, il primo cd è il sistema operativo vero e proprio, il secondo cd contiene l'ambiente Classic (ovvero, il "vecchio" sistema operativo 9.1 o successivi), mentre il terzo cd contiene appunto i developer tools. Si tratta di installare l'apposito "package" ed il gioco è fatto. L'applicazione PB (come pare si chiami familiarmente Project Builder) si trova poi nella cartella `/Developer/Applications` della partizione che contiene il sistema operativo. Attenzione a non confonderlo con ProjectBuilderWO, che è la versione di PB pensata appositamente per WebObjects, l'ambiente per la realizzazione di siti web evoluti. Per mia comodità, ho trascinato l'icona di PB sul dock, in modo da averlo sempre ad un doppio clic di distanza.
PB è un ambiente integrato per lo sviluppo di applicazioni. Non è una applicazione monolitica con cui fare tutto, ma è semplicemente un centro di controllo ed organizzazione del traffico legato al processo di sviluppo.
Parentesi. 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 verificando 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. Chiusa parentesi, ma la storia continua ugualmente.
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 PB non è altro che una comoda interfaccia grafica per il processo di sviluppo di una applicazione: 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

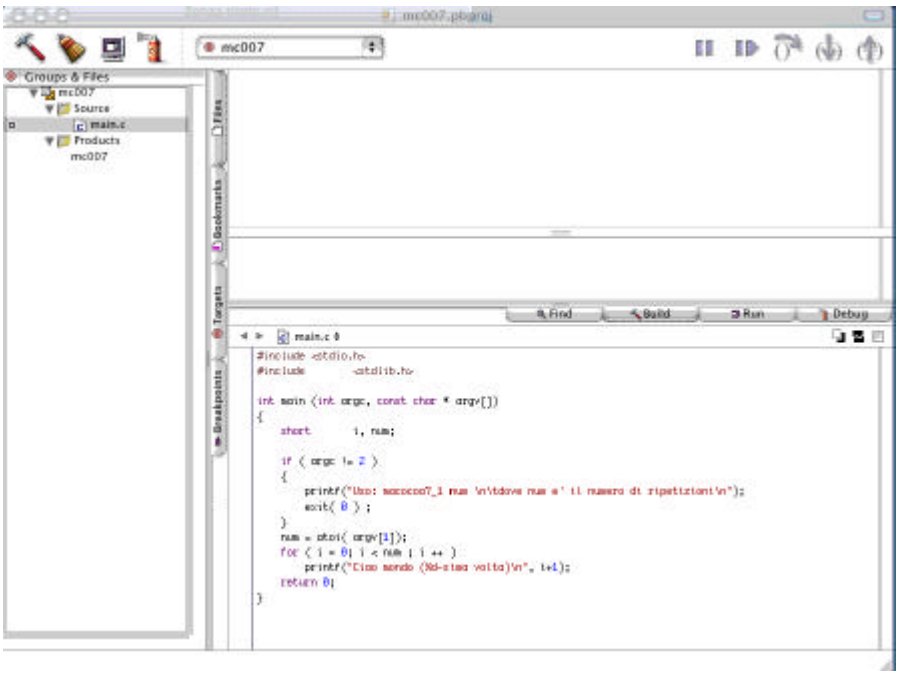
possibilità di costruire un dizionario dei simboli, facilitando la ricerca di informazioni, e per finire supporta il debugging dell'applicazione con qualcosa in più che lanciare in esecuzione il programma di debugging.

La Finestra di PB

Ora che abbiamo un'idea di cosa aspettarci, lanciamo PB 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. Utilizzando il linguaggio OO, un template è una Classe Astratta.

Vediamo più da vicino la finestra del progetto.



E' divisa in tre parti; nella parte sinistra si trova l'elenco di tutte le "cose" (mi veniva da scrivere oggetti, ma adesso questa bella parola generica che era oggetti bisogna riservarla agli oggetti della programmazione OO) che partecipano alla realizzazione del progetto. In particolare, nella vista a file si trova l'elenco dei file, divisi in vari gruppi, rappresentati da cartelle (anche se tali cartelle non necessariamente sono reali). Le altre viste, bookmark al momento ignoro, target dovrebbe essere il risultato del processo di produzione (l'applicazione lanciabile tramite un doppio clic), mentre breakpoint contiene i punti di "rottura" che ci serviranno poi a debuggare il programma.
La parte superiore mostra ancora quattro viste, Find per cercare informazioni all'interno dei documenti di progetto, Build che presumo conterrà i risultati intermedi del processo di costruzione del programma, Run per contenere le informazioni durante l'esecuzione del programma, e Debug per le informazioni necessarie al debug. Infine, la parte inferiore è un editor di testo; niente di speciale, se non che colora ed indenta il testo del programma in accordo al linguaggio (e che cambia contestualmente al tipo di file modificato). Facendo clic su uno dei file nella lista a sinistra, il pannello è riempito con quel file. Se invece faccio doppio clic, il file è aperto in una finestra indipendente.
Rimangono da considerare i controlli sulla parte superiore della finestra: i quattro pulsanti sulla sinistra servono per compilare il programma, reinizializzare il progetto, lanciare in esecuzione il programma o debuggare lo stesso. I pulsanti sulla destra saranno utili quanto si farà il debug del programma. Il menu al centro permette di selezionare il target attivo.

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 lì. 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: faccio clic sull'icona in alto a sinistra rappresentante un bel martello. Quel che ottengo è che la parte superiore della finestra cambia aspetto, si divide in due pannelli; nella parte superiore c'è l'elenco degli errori trovati nel processo di compilazione, nella parte sottostante i dettagli del processo, che vado qui a riportare:

```
/usr/bin/jam -d1
JAMBASE=/Developer/Makefiles/pbx_jamfiles/ProjectBuilderJambase JAMFILE=-
build ACTION=build TARGETNAME=mc007 NATIVE_ARCH=ppc
BUILD_STYLE=Development
CPP_HEADERMAP_FILE=/Users/djzero00/Documents/macProg/devTest/mc007/build/intermedia
tes/mc007.build/Headermaps/mc007.hmap
DSTROOT=/
OBJROOT=/Users/djzero00/Documents/macProg/devTest/mc007/build/intermediates
SRCROOT=/Users/djzero00/Documents/macProg/devTest/mc007
SYMROOT=/Users/djzero00/Documents/macProg/devTest/mc007/build
...updating 8 target(s)...
BuildPhase mc007
Completed phase  for mc007
Mkdir
/Users/djzero00/Documents/macProg/devTest/mc007/build/intermediates/mc007.build/Obj
ects/ppc

CompileC
/Users/djzero00/Documents/macProg/devTest/mc007/build/intermediates/mc007.build/Obj
ects/ppc/main.o

main.c: In function `main':
main.c:13: warning: implicit declaration of function `atoi'
BuildPhase mc007
Completed phase  for mc007
StandaloneExecutable
/Users/djzero00/Documents/macProg/devTest/mc007/build/mc007
/usr/bin/ld: warning -F: directory name
(/Users/djzero00/Documents/macProg/devTest/mc007/build/ProjectHeaders)
does not exist
BuildPhase mc007
Completed phase  for mc007
BuildPhase mc007
Completed phase  for mc007
...updated 8 target(s)...
```

Il succo della faccenda è che il file `main.c` è compilato, e viene riportato un 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 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 più pulitamente, nel pannello superiore. Facendo clic sul warning, la

finestra di editing posiziona il cursore nel punto in cui il warning è stato riconosciuto. 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.
Siamo pronti per lanciare in esecuzione il programma. Faccio clic sul terzo pulsante in alto da sinistra (uno schermo); ancora una volta, la parte superiore della finestra cambia aspetto, è attivato il pannello "Run", e nella finestra 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 la finestra Terminale, ed andare a ragionare con la linea di comando. Bene, cercate Terminale nelle applicazioni, lanciatelo, e portatevi, pieni di buona volontà, nella directory corretta. La mia (il mio utente è djzero00, il vostro non lo so) è la seguente

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

La directory parte dal posto dove avete detto di conservare il progetto (per me, macocoa7_1), e scende ulteriormente di un livello, dentro la cartella build, dove sono appunto conservati i risultati della compilazione.
Con il comando ls -l (o, molto più semplicemente l), avrete qualcosa del tipo:

```
total 8
drwxr-xr-x  4 djzero00  staff   92 Nov  3 19:44 build
-rw-rw-r--  1 djzero00  staff  367 Nov  3 19:43 main.c
drwxrwxr-x  4 djzero00  staff   92 Nov  3 19:42 mc007.pbproj
```

ciò che ci interessa è il secondo, l'eseguibile che ci interessa. Per lanciare in esecuzione il programma, si tratta di fare semplicemente macocoa7_1. Lo faccio, e non funziona... (salta fuori `mc007: Command not found.`) Ah, per forza, il path non è corretto.

Scrivo

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

```
echo ${PATH}
/Users/djzero00/bin/powerpc-apple-macos:
/Users/djzero00/bin:
/usr/local/bin:
/usr/bin:
/bin:
/usr/local/sbin:
/usr/sbin:
/sbin
```

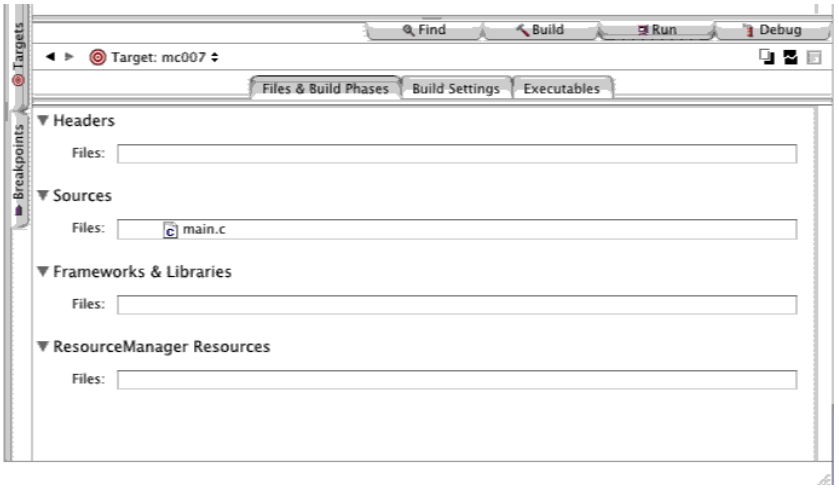
La directory in cui si trova l'eseguibile non compare nell'elenco, quindi, niente esecuzione. Sono quindi costretto a specificare il percorso completo, come ho fatto sopra, ma è un po' noioso. Uso allora la seguente scorciatoia, sempre disponibile in un ambiente Unix: esiste la possibilità di indicare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

```
./mc007 3
```

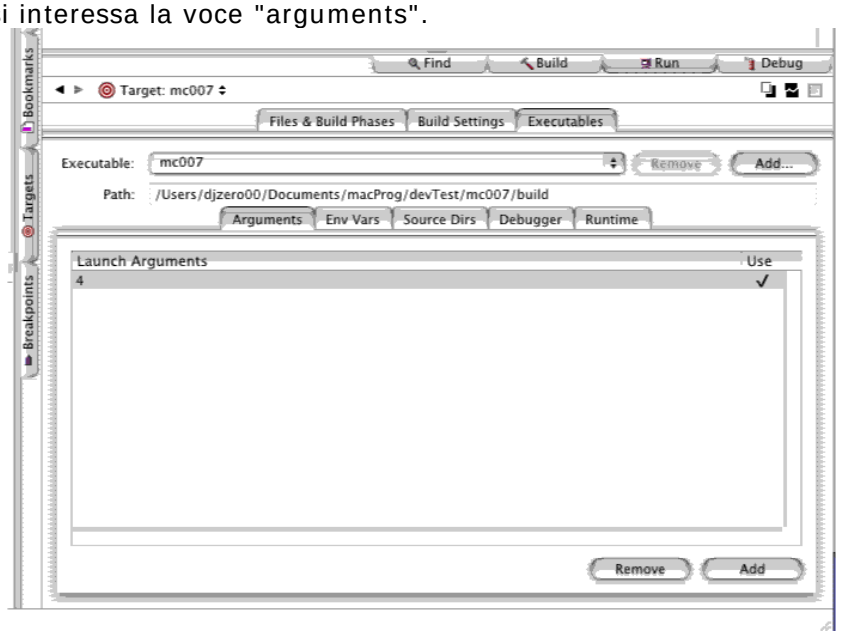
e tutto torna a funzionare. Per completezza, ecco cosa salta fuori:

```
Ciao mondo (1-sima volta)
Ciao mondo (2-sima volta)
Ciao mondo (3-sima volta)
```

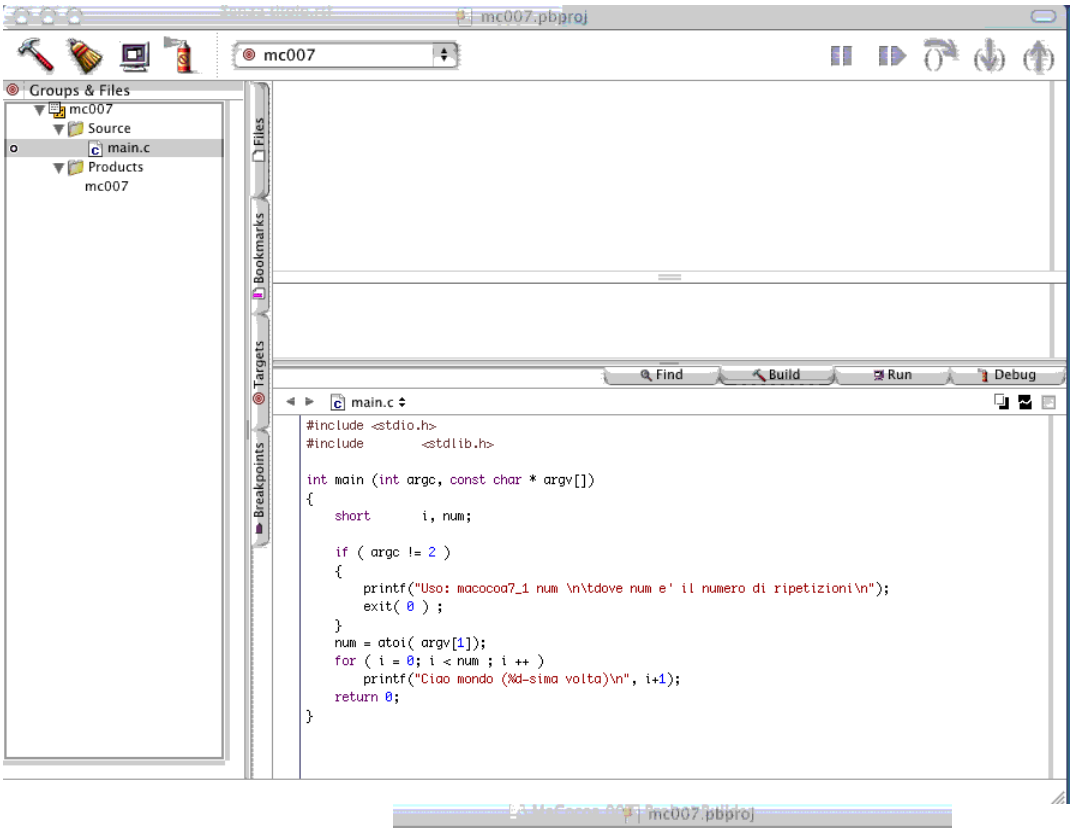
Potete provare con altri numeri, ma non otterrete nulla di più eccitante. Torniamo allora dentro PB. Esiste un metodo per passare argomenti al programma senza dover passare attraverso il terminale: si tratta di andare a selezionare uno dei pannelli sulla sinistra, target. Si ottiene così un pannello diviso in due parti: nella parte superiore, quella che ci interessa, è indicato l'elenco dei prodotti del nostro processo di sviluppo del programma. Al momento c'è solo la dicitura macocoa7_1 con il quale siamo partiti. Selezionando, il pannello di editing della finestra di progetto mostra un pannello preferenze con diverse possibilità.

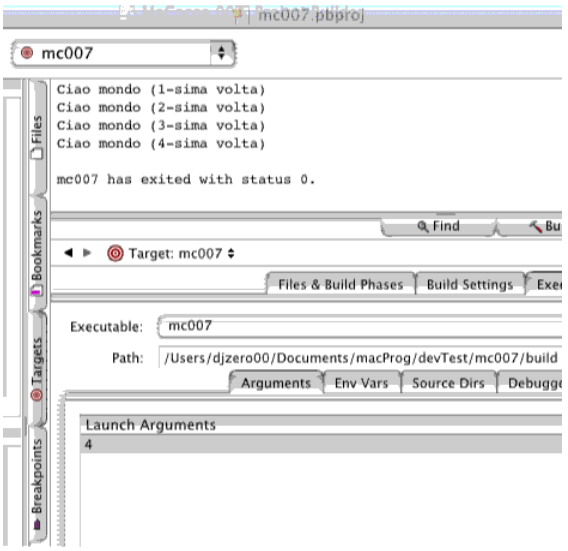


Facendo clic su "Executables" salta fuori un pannello all'interno del quale c'è un altro pannello preferenze, dove si interessa la voce "arguments".



Qui è possibile aggiungere argomenti di lancio al programma, come abbiamo fatto da linea di comando. Inserisco 4 con il pulsante "Add". Torno allora ad eseguire il programma (clic sull'icona a forma di schermo). Il pannello "Run" 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.

Un programma in Objective C

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 un 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 PB 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.
PB 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 (Aiuto Apple, e cercate appunto `NSLog`) e trovo una cosa piuttosto interessante.

La funzione NSLog

`NSLog` è una funzione (dicono una macro, ma al momento ci interessa poco) 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, dovrebbe funzionare più o meno come la funzione di libreria standard del C `printf`, tanto cara ed utile a tutti i programmatori.
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. Vado sul manuale (manu Aiuto, e poi cerco `NSString`... ci

siamo capiti...) e trovo 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 @"". Qui non ho trovato molte informazioni. A quanto pare, con questo operatore è possibile definire al volo oggetti di tipo NSString; non so 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:

```
Oct 29 22:05:38 mc008[584] ______1234567890_1234567890123456_
Oct 29 22:05:38 mc008[584] aloha_  10.34000_          -76546_salve a tutti_
mc008 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).

Ancora NSLog

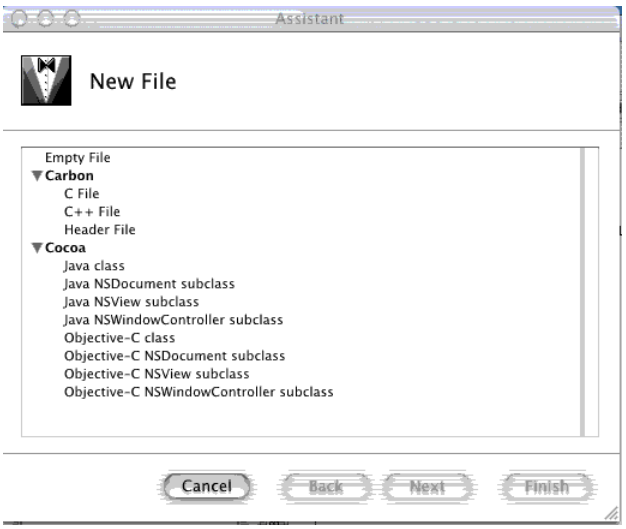
Quando uno non sa qualcosa, cerca in Internet. Trovo il sito www.cocoadevcentral.com, dove si trovano maggiori informazioni sull'argomento. Riprendo da lì velocemente:
Come per la funzione printf dello standard C, nella stringa di formato si possono inserire delle sequenze di controllo che cominciano col carattere %, seguite da un carattere che indica il tipo di valore da scrivere. Il tipo è uno dei seguenti:

```
%          Type          Result
--------------------------------
c          char          singolo carattere
i,d        int           numero in rappresentazione decimale
o          int           numero in rappresentazione ottale
x,X        int           numero in rappresentazione esadecimale
u          int           intero senza segno
s          char *        classica stringa C terminata da \0
f          double/float  numero floating point
e,E        double/float  numero floating point in notazione ¬ scientifica
@          object        valore di un oggetto
%          -             il carattere %
```

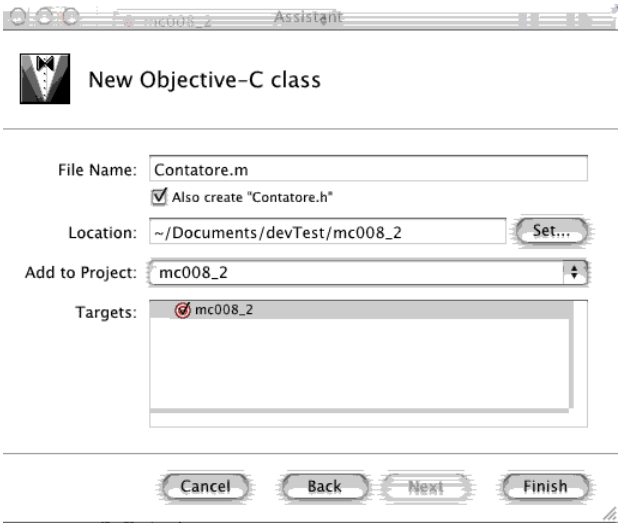
Tra il carattere % l'indicazione del tipo di valore può essere presente una indicazione di formato, come già del resto mostrato nell'esempio precedente. Ad esempio con %10.5f si indica un numero floating point rappresentato con dieci cifre totali, di cui cinque decimali.
C'è una cosa da notare, e che 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, 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 decrementare il contatore. Facciamo partire il PB 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.



PB ha infatti 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 all'interno di PB. 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...

Finalmente Cocoa

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 Project Builder PB.
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

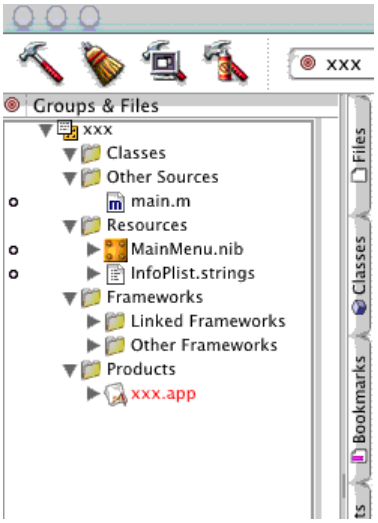
Abbiamo alcuni concetti da esplorare prima di avventurarci in pratica. Il primo concetto è il file **NIB** (ignoro il significato dell'acronimo). Un file nib è associato ad una finestra, o meglio, è un archivio di oggetti generati attraverso IB. Ovvero, IB salva tutto ciò che è in grado di manipolare 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

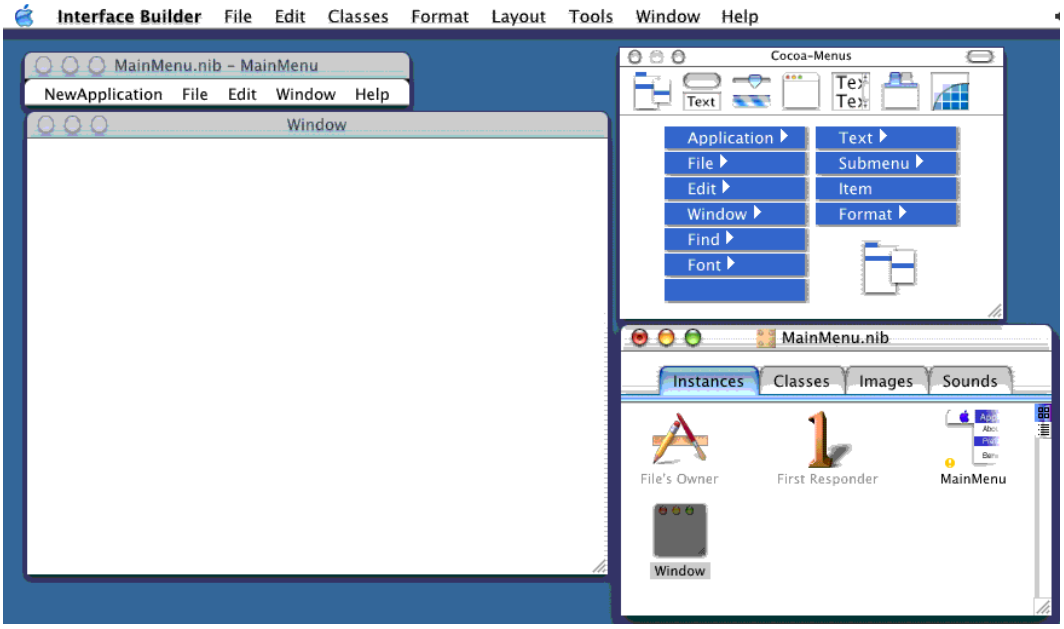
A questo punto, parto con la nostra applicazione. Apro PB e specifico un nuovo progetto, questa volta una `Cocoa Application`. PB, di suo, ci mette un po' di cartelle e qualche file predefinito.



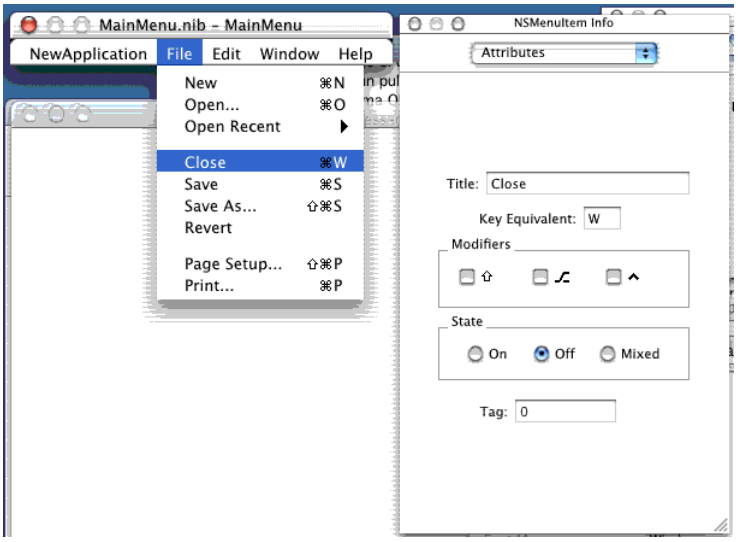
Ho una cartella "`Classes`" dove sono (saranno) inserite le Classi da me definite e che costituiscono l'applicazione. Nella cartella "`Other Sources`" c'è il file `main.c`. A vederlo, non è che ci sia molto da modificare, per il momento (è una riga sola):

```
#import

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.



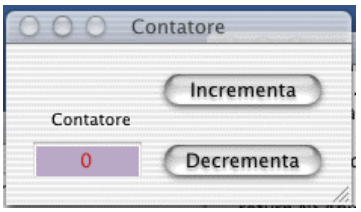
La prima finestra, larga e stretta, denominata "`MainMenu.nib - MainMenu`", è un po' speciale perché contiene i menu che fanno parte dell'applicazione. Con un po' di pazienza e un uso dosato dei clic del mouse, è possibile costruire il menu, voce per voce, completo di scorciatoie da tastiera. Allo scopo, è molto utile aprire una ulteriore finestra, dal menu Tools, la voce "`Show Info`".



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'n'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:

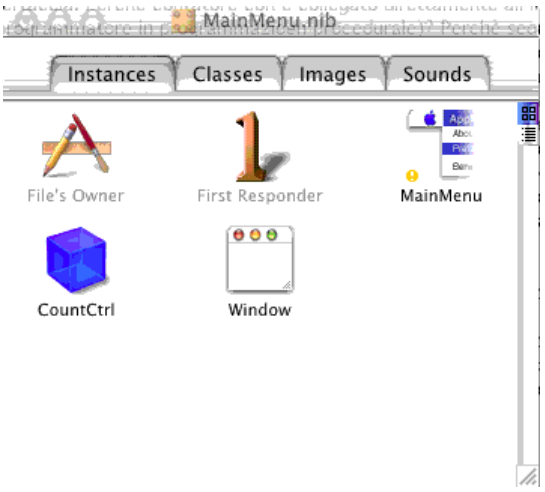
- muovendo gli elementi qui e lì all'interno della finestra, ogni tanto compaiono delle linee blu di guida per il posizionamento; il posizionamento stesso poi è "costretto" all'interno di una griglia nascosta. Ora, Apple ha reso disponibile le linee guida di progettazione dell'interfaccia, un tomo corposo dove sono elencate tutte le regole da seguire per avere una interfaccia in puro stile Aqua (trovate il tomo a questo path: `/Developer/Documentation/Essentials/AquaHIGuidelines/index.html`). 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 opera 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

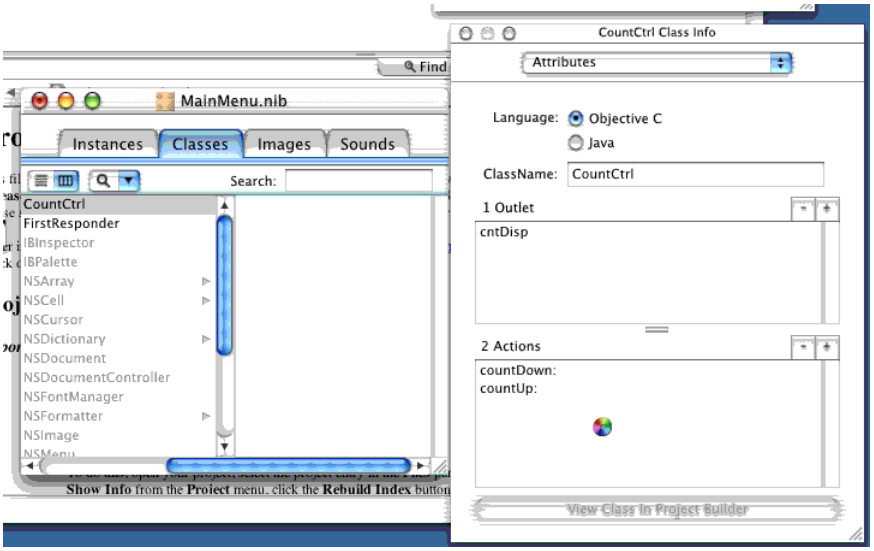
Ad un certo punto della storia avevo introdotto il modello MVC (Model-View-Controller) come un utile paradigma per la realizzazione di applicazioni object oriented, particolarmente utile nei casi in cui si ha a che fare con una interfaccia grafica. È giunto il momento di applicare questo paradigma. Avendo 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 un'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 invii 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`.

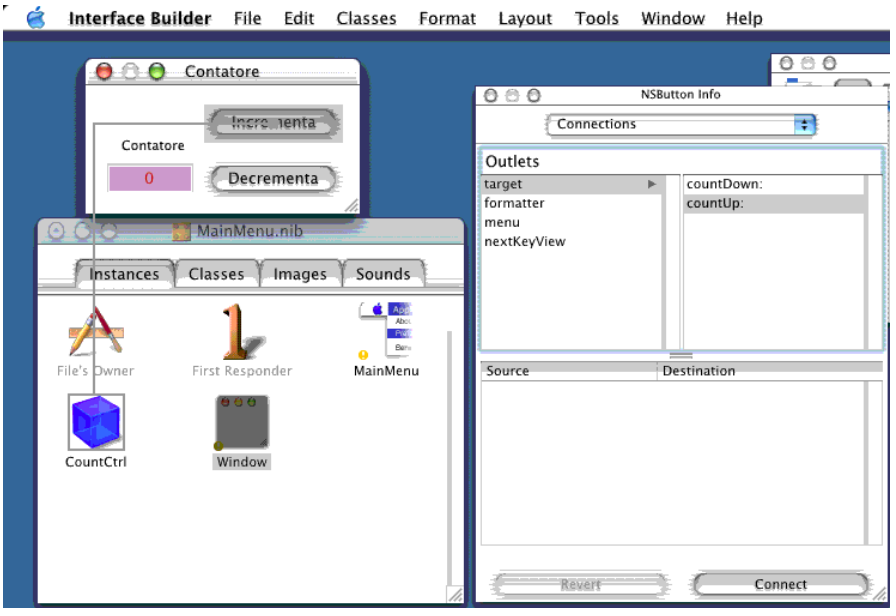


Questa è l'istanza della classe (classe ed oggetto hanno lo stesso nome... non è una bella cosa, potrebbe ingenerare confusione, ma poiché c'è una sola istanza della classe, lo sopporto). Ma io torno indietro alla vista per Classi, perché devo ancora perfezionare la dichiarazione della classe `CountCtrl`.
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

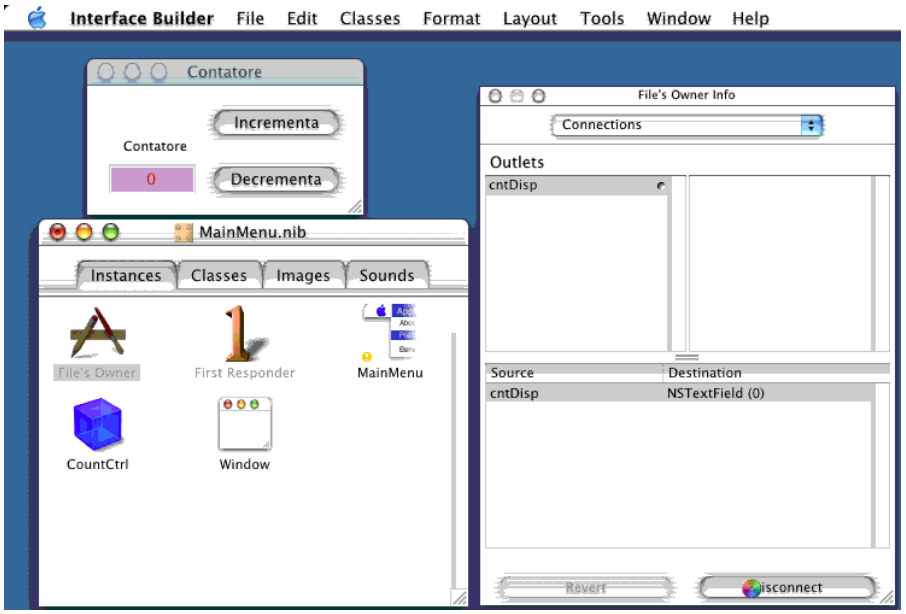
`countDown:` ed una variabile d'istanza `cntDisp`. IB chiama queste due cose rispettivamente una Action e un Outlet.
Parentesi terminologica. Non è che Action sia un altro nome per Metodo e Outlet un altro nome per variabile d'istanza. Una Action è un metodo utilizzato per comunicazioni tra oggetti facenti 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 delle `Info`, sul pannello `Attributes` (sto lavorando con IB 10.1... la cosa potrebbe essere diversa con versioni precedenti). 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 di `Info` 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 `Info` 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 PB 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 Files`". 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
```

cioè, 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 in esame. 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 a 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 n'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 idea talmente buona 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...

Attività di contorno

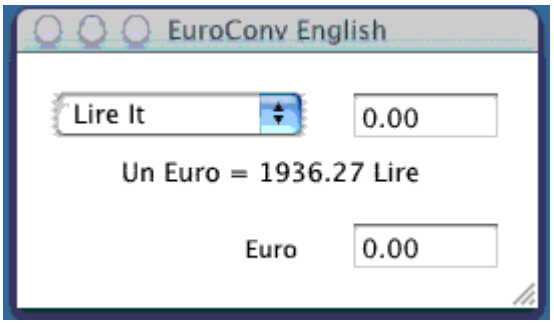
Usare IB e PB

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.
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

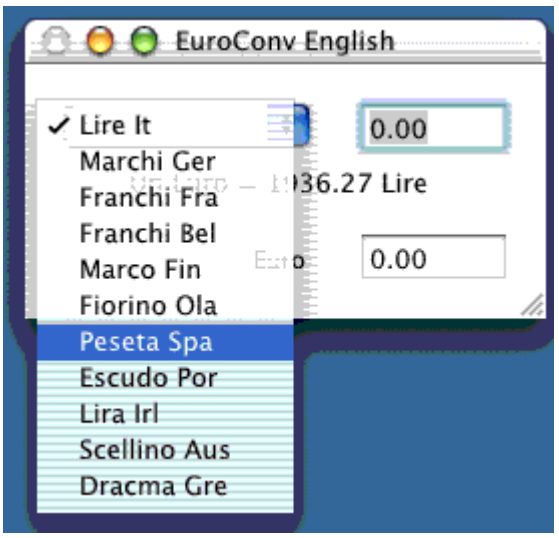
L'applicazione **EuroConv** presenta una interfaccia molto semplice: ci sono due campi di testo per contenere l'uno l'importo in Euro e l'altro l'importo in una delle valute dell'area Euro. Un menu poop-up permette di selezionare la valuta che si intende utilizzare (lire, marchi, franchi, eccetera),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



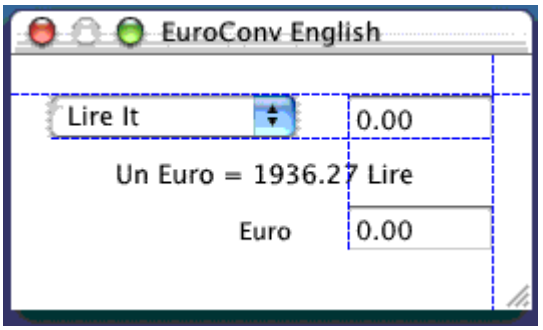
L'idea è che la semplice scrittura di un valore in uno dei due campi scateni la conversione (in un senso o nell'altro) dell'importo. D'altra parte, il cambio di valuta mantiene fisso il valore in Euro e converte il campo della valuta locale. In questo modo, se voglio sapere a quanti franchi francesi 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 PB, 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 nella puntata 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.

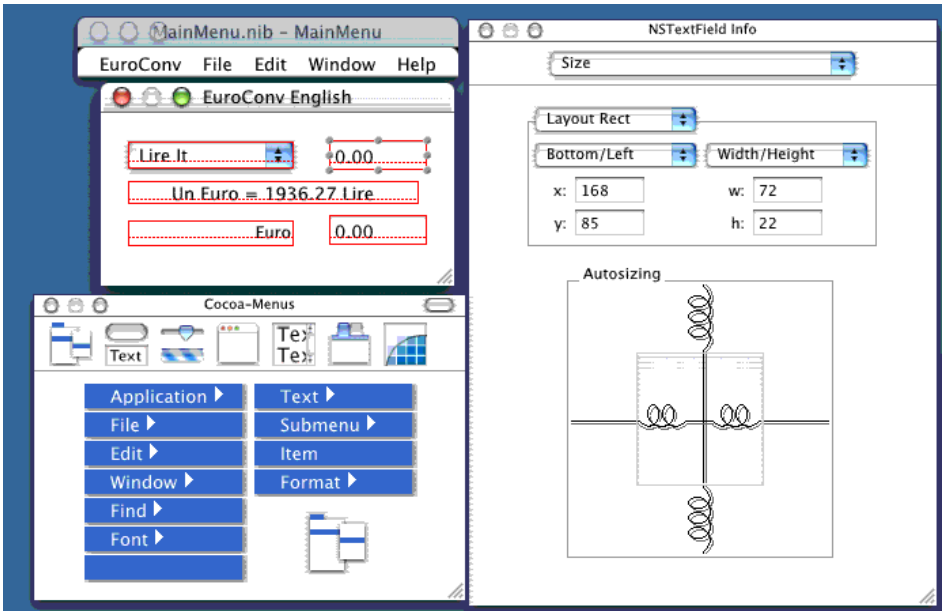


Dopo di che, allineo il menu pop-up in modo che, di altezza uguale, corrisponda orizzontalmente. Il campo della valuta in Euro lo piazzo sotto il campo testo della valuta locale, e poi aggiungo due testi non editabili per contenere informazioni (in uno ci ho scritto semplicemente "`Euro`" e non lo cambierò 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 le 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.

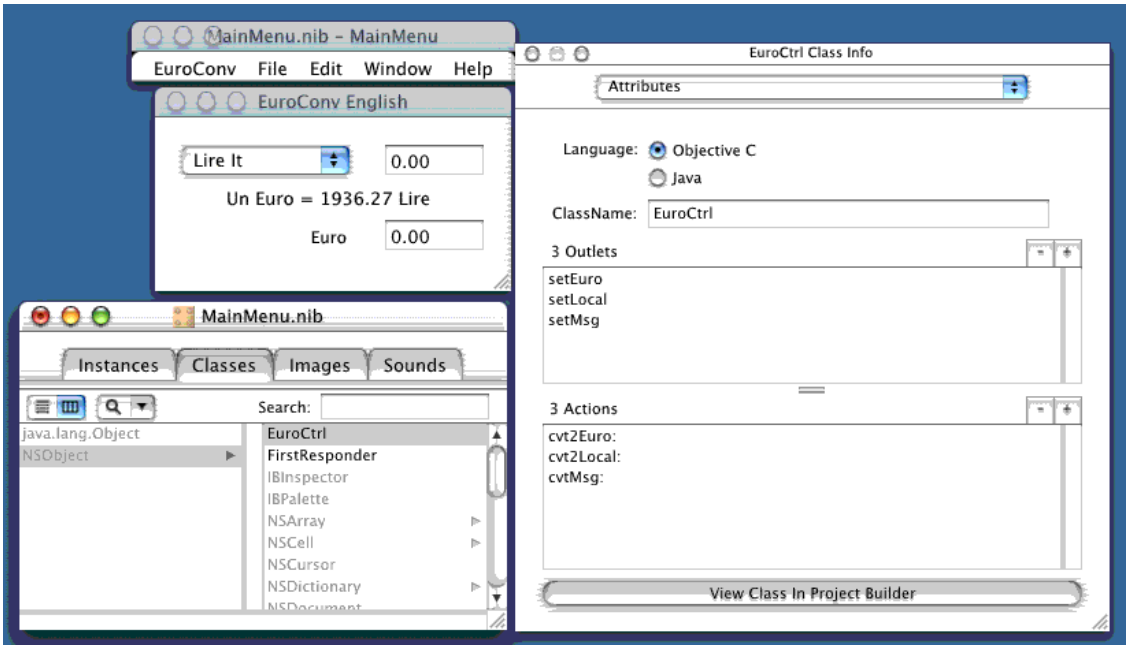
Per fare ciò, ho a disposizione il pannello "Size" della finestra di informazioni (che si ottiene facendo "Show info" dal menu "Tools" di IB). In questo pannello, oltre a decidere numericamente le dimensioni, ho un disegno (il riquadro "Autosizing") che determina il comportamento dell'oggetto selezionato 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.
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 sulle Info...). 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 EuroCtrl. Questa classe avrà un'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 le 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.

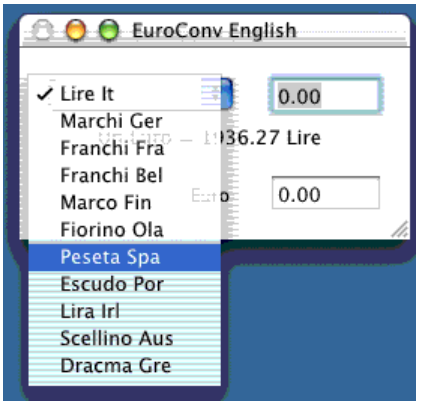
Un po' di codice

Siamo adesso in PB. 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!

La puntata 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 nella puntata 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:  ]
```

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:

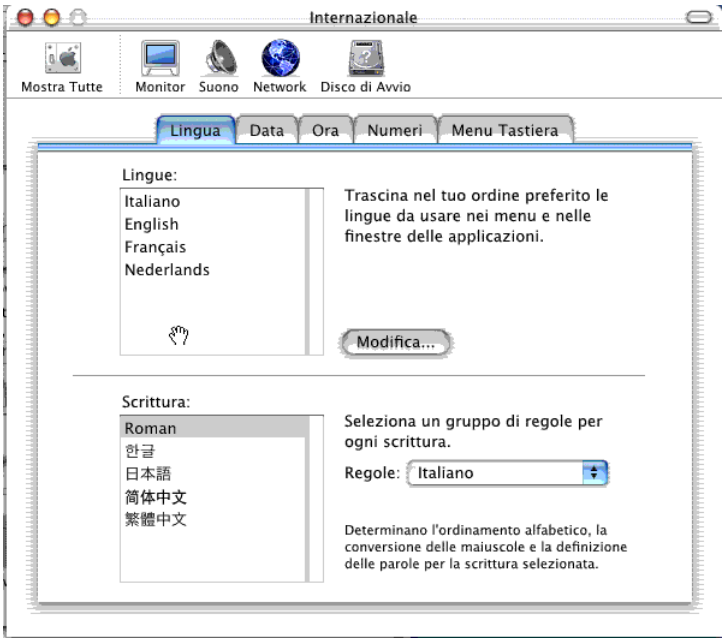
```
- (IBAction)cvtMsg:(id)sender
{
        currSel = [ sender indexOfSelectedItem ];
        [ setLocal setFloatValue: [ setEuro floatValue ] * cvtRate[ currSel ] ] ;
}
```

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 nella prossima puntata, a cui pieroangelicamente vi rimando.

Localizzazione, icone ed altre storie

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 questa puntata.
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.

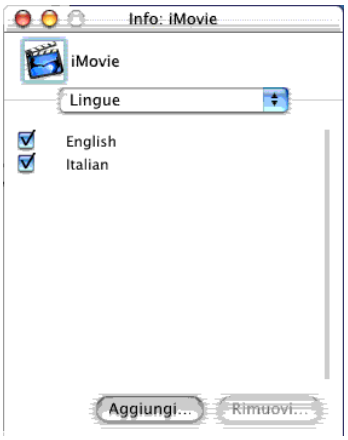


Dal punto di vista del programmatore, l'adeguamento alla lingua (operazione nota come **localizzazione** o **internazionalizzazione**) genera tipicamente un sovraccarico di lavoro piuttosto noioso e non focalizzato sull'applicazione vera e propria. Apple, nel documento System Overview, opera una sottile 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 pazienza da certosino. Con Mac Os X le cose sono molto diverse: esiste un'unica applicazione che contiene al suo interno tutte le risorse per la localizzazione nelle varie lingue. Questo significa che lo stesso file (che so, iMovie.app) 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 iMovie 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...).

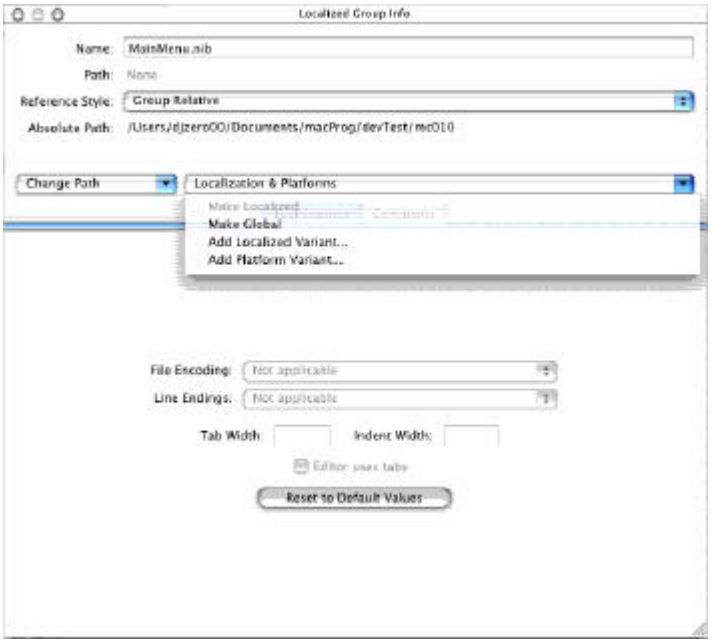
```
[localhost:/Applications/iMovie.app/Contents] djzero00% l
total 32
-rw-rw-r--    1 root  admin  3060 Sep  8 00:18 Info.plist
drwxrwxr-x    4 root  admin   264 Nov 10 15:13 MacOSClassic
-rw-rw-r--    1 root  admin     8 Nov  6 19:39 PkgInfo
drwxrwxr-x   21 root  admin   670 Nov 10 15:13 Plug-ins
drwxrwxr-x   18 root  admin   568 Nov  6 19:39 Resources
-rw-rw-r--    1 root  admin   514 Sep  8 03:53 version.plist
[localhost:iMovie.app/Contents/resources] djzero00% l
total 7160
drwxrwxr-x    6 root  admin   264 Nov  6 19:39 English.lpro
jdrwxrwxr-x   9 root  admin   264 Nov  6 19:39 Interface Files
drwxrwxr-x    6 root  admin   264 Nov  6 19:39 Italian.lproj
drwxrwxr-x    20 root  admin  636 Nov  6 19:39 Sound Effects
-rw-rw-r--     1 root  admin  2662995 Aug 15 21:00 Splash.mov
-rw-rw-r--     1 root  admin    35067 Nov  6 19:39 app.icns
<< altri file che non ci interessano>>
```

Il contenuto della cartella `/Application/iMovie.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 iMovie fosse localizzato anche

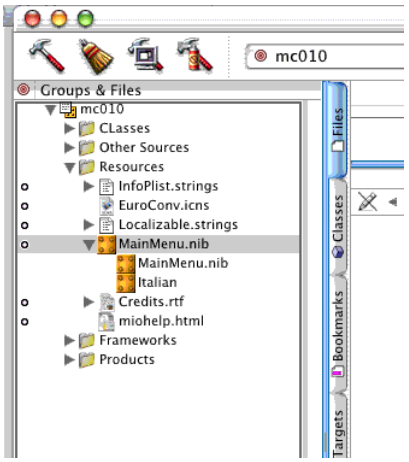
per la lingua tedesca, ci sarebbe stata una cartella "`Deutsch.lproj`", e via di questo passo per le altre lingue.

Supporto all'internazionalizzazione

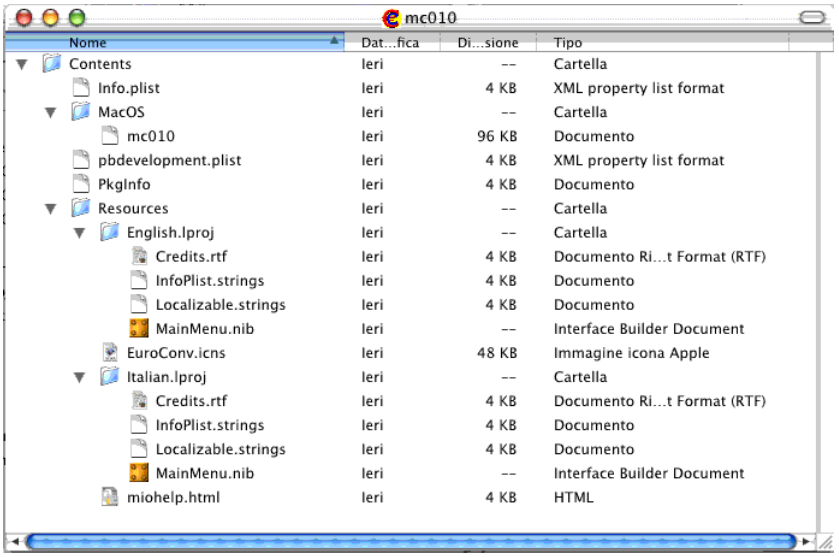
Per realizzare una struttura similare dell'applicazione EuroConv, torno nel PB e seleziono le risorse che voglio far comparire in versione localizzata. Al momento, ho solo il file nib che contiene finestra e menu. Seleziono il file e scelgo di vedere le informazioni relative (menu '`Project`', voce '`Show Info`'). Nella finestra che si apre seleziono da uno dei menu popup la possibilità di inserire una variante locale.



Seleziono '`Add Localized Variant`' ed al dialogo 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 adeguarli 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 è molto più semplice, perché si può fare direttamente dal Finder (perché non si potesse fare la stessa cosa con iMovie mi resta misterioso... dev'essere una complicata questione di diritti che non ho voglia di indagare). Dicevo, 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 la puntata 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 la stringa corretta in dipendenza della 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"
```

ed 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 ;
```

e 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:

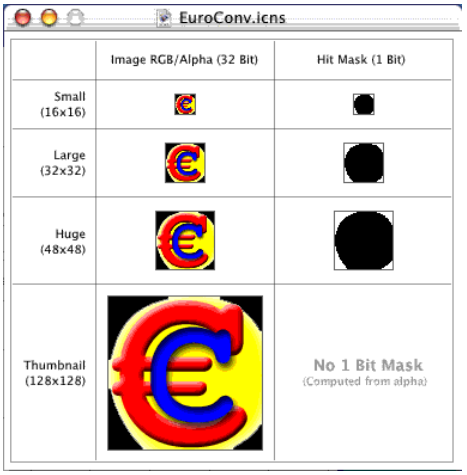
```
NSLocalizedString( cvtMsg[ currSel ], @"cheneso" )
```

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.

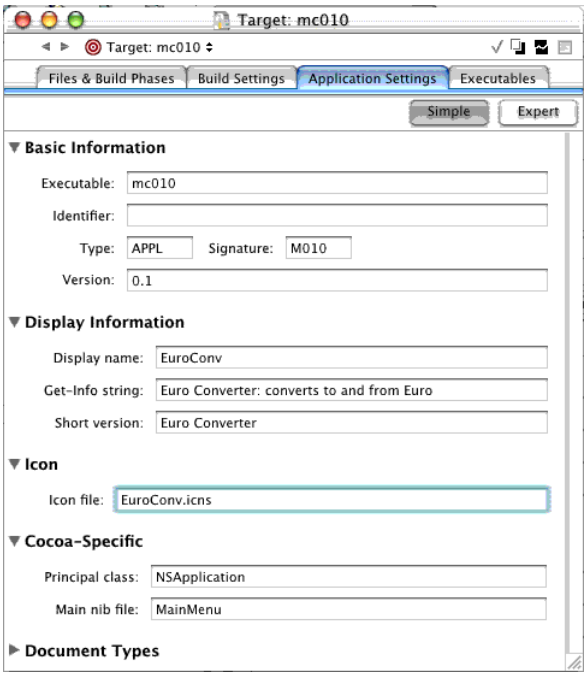
Rimane da chiarire l'uso del commento all'interno della macro `NSLocalizedString`. In effetti, non ha qui alcuna funzione. È però utilizzata come supporto all'internazionalizzazione. Accade che il programmatore non conosca tutte le lingue del mondo, ma solo la sua. Allora, quando ha bisogno di una stringa 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.

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 con Photoshop. Poi apro l'applicazione IconComposer nei Developer Tools; con la tecnica del drag'n'drop trascino il file (ancora nel formato photoshop!) 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 PB e lo aggiungo al progetto, nella sezione risorse. Il passo successivo è associare questo file alle icone del programma; basta andare nel pannello 'Application Settings' del Target sotto esame e scrivere il nome del file nell'apposito spazio.

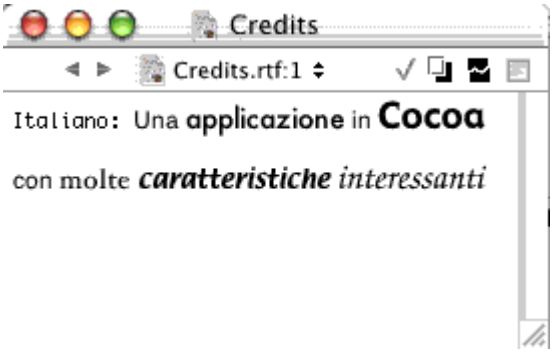


Già che ci sono ho riempito un altro paio di campi, utili per cambiare il nome del file eseguibile (altrimenti PB usa di default il nome del progetto), informazioni da mostrare nella finestra del Finder, eccetera.

About

Per chiudere, personalizzo la finestra di About dell'applicazione. In primo luogo utilizzo il file InfoPlist.string, costruito da PB 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 PB 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 credo che si debba lavorare un bel po'. Non è questo il momento. Per ora mi accontento di aver costruito una applicazione funzionante e abbastanza completa. Fra un po', potrei anche cominciare a spacciarne in giro come freeware...

Questioni di memoria

Puntatori annacquati

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.

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 connette 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).

Una tabella di file

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.

L'interfaccia

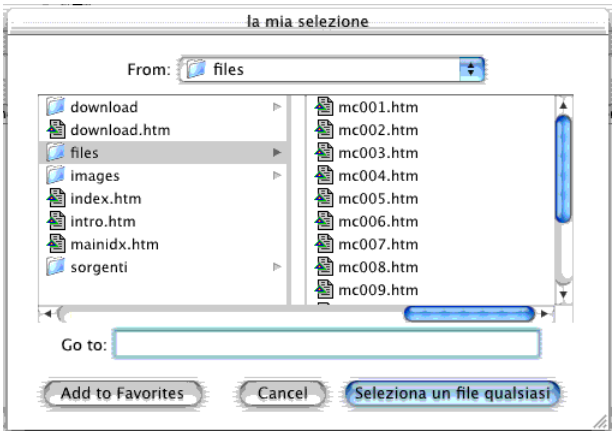
Velocemente costruisco una interfaccia per fare quanto detto sopra. Sarà scarna e per nulla bella, ma al momento non ci interessa.

Mi occorre un pulsante per far partire le attività, un campo dove mettere il percorso completo del file (mi 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).

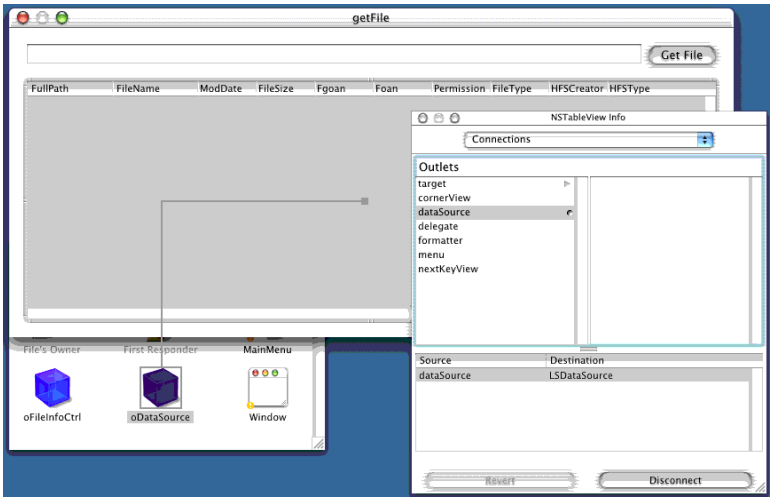
```
- (id) initWithPath: (NSString*) aFile
{
        NSFileManager *manager = [NSFileManager defaultManager];
        NSDictionary *fattrs = [manager fileAttributesAtPath: aFile traverseLink:NO];
        NSNumber *num ;
        [ super init ];
        [ self setFileFullPath: aFile ];
        [ self setFileName: [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 da un file, e le varie colonne contengono i vari campi di informazioni 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; mostro le due soluzioni possibili e già discusse:

```
- (id ) init
{
        self = [ super init ] ;
//      [self setListaFile: [ NSMutableArray array ]];                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 ];
else if ( [ colId isEqual: @"nome seconda colonna")
                valore = [ fInfo ];
```

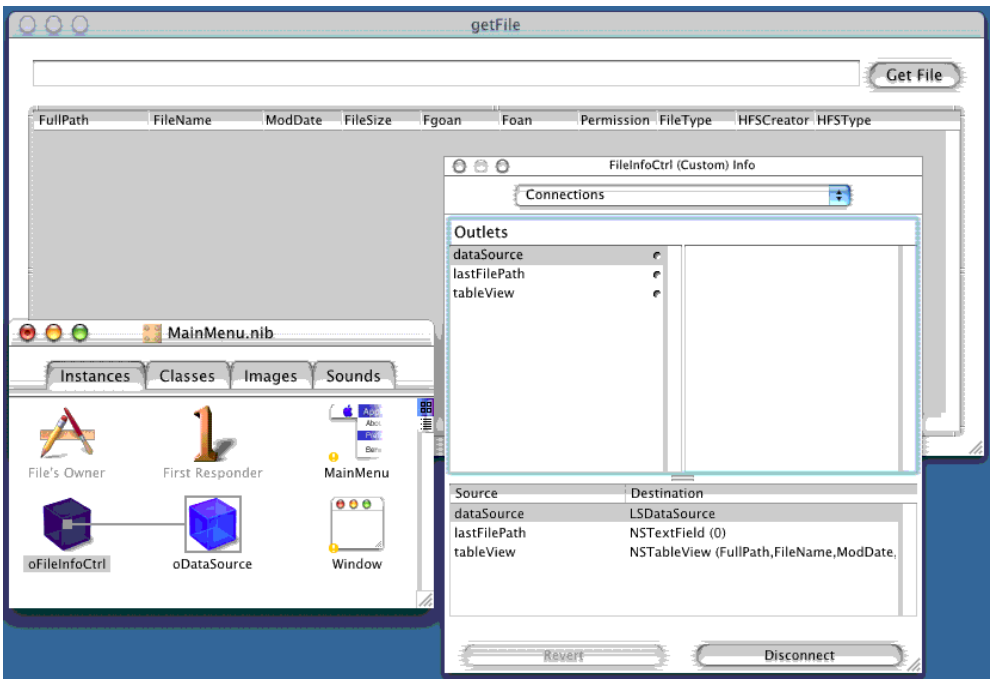
e via così per tutte le possibili colonne. Invece, c'è un trucco. All'interno di un oggetto posso recuperare il valore di una variabile d'istanza se ne conosco il nome (non è certo il metodo più efficiente per farlo, però è mooolto comodo). Ciò si realizza con il metodo valueForKey:.
Con questo trucco,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

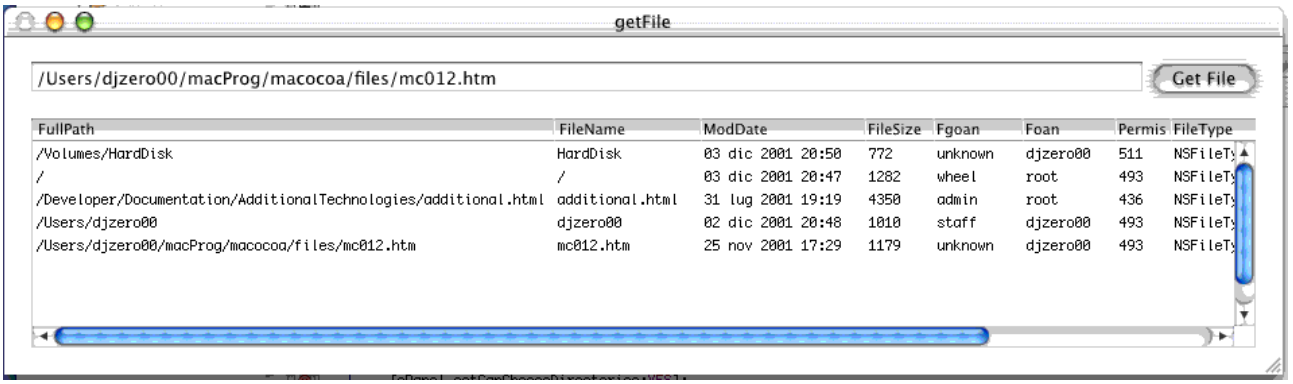
Nel caso ci sia una selezione effettiva (l'utente ha fatto clic sul pulsante di OK), recupero il nome del file, costruisco un nuovo oggetto fInfo con tutte le informazioni relative al file selezionato ed aggiungo tale oggetto alla struttura dati della classe LSDataSource invocando il metodo addFileEntry: 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 all `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...).
Ma questo sarà argomento di qualche altro capitolo.



I nudi fatti:

- Il file FileInfoCtrl.h
- Il file FileInfoCtrl.m
- Il file LSDataSource.h
- Il file LSDataSource.m
- Il file LSFileInfo.h
- Il file LSFileInfo.m
- L'inutile file main.m

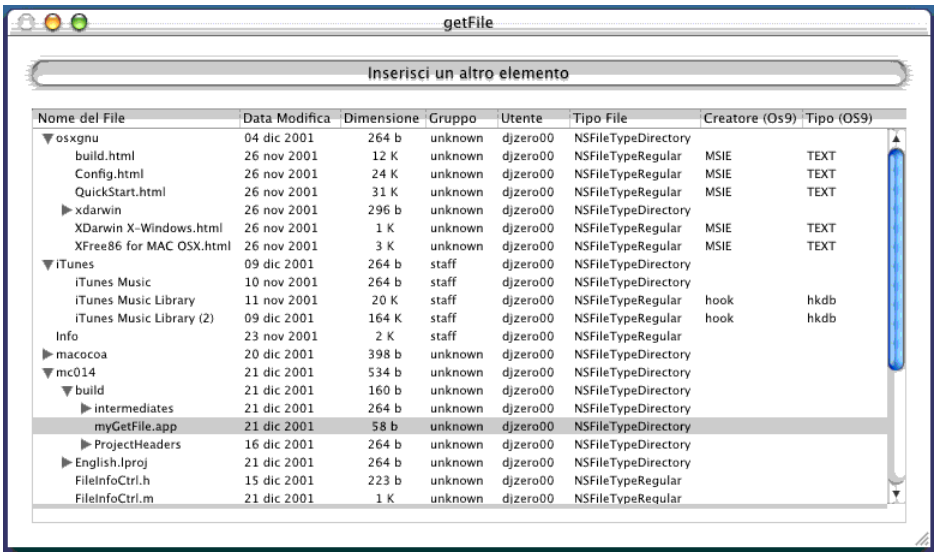
L'inizio del Catalogo

Espansione di file

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.

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.



Nome del File	Data Modifica	Dimensione	Gruppo	Utente	Tipo File	Creatore (Os9)	Tipo (OS9)
▼osxgnu	04 dic 2001	264 b	unknown	djzero00	NSFileTypeDirectory		
build.html	26 nov 2001	12 K	unknown	djzero00	NSFileTypeRegular	MSIE	TEXT
Config.html	26 nov 2001	24 K	unknown	djzero00	NSFileTypeRegular	MSIE	TEXT
QuickStart.html	26 nov 2001	31 K	unknown	djzero00	NSFileTypeRegular	MSIE	TEXT
▶xdarwin	26 nov 2001	296 b	unknown	djzero00	NSFileTypeDirectory		
XDarwin X-Windows.html	26 nov 2001	1 K	unknown	djzero00	NSFileTypeRegular	MSIE	TEXT
XFree86 for MAC OSX.html	26 nov 2001	3 K	unknown	djzero00	NSFileTypeRegular	MSIE	TEXT
▼iTunes	09 dic 2001	264 b	staff	djzero00	NSFileTypeDirectory		
iTunes Music	10 nov 2001	264 b	staff	djzero00	NSFileTypeDirectory		
iTunes Music Library	11 nov 2001	20 K	staff	djzero00	NSFileTypeRegular	hook	hkdb
iTunes Music Library (2)	09 dic 2001	164 K	staff	djzero00	NSFileTypeRegular	hook	hkdb
Info	23 nov 2001	2 K	staff	djzero00	NSFileTypeRegular		
▶macocoa	20 dic 2001	398 b	unknown	djzero00	NSFileTypeDirectory		
▼mc014	21 dic 2001	534 b	unknown	djzero00	NSFileTypeDirectory		
▼build	21 dic 2001	160 b	unknown	djzero00	NSFileTypeDirectory		
▶intermediates	21 dic 2001	264 b	unknown	djzero00	NSFileTypeDirectory		
myGetFile.app	21 dic 2001	58 b	unknown	djzero00	NSFileTypeDirectory		
▶ProjectHeaders	16 dic 2001	264 b	unknown	djzero00	NSFileTypeDirectory		
▶English.lproj	21 dic 2001	264 b	unknown	djzero00	NSFileTypeDirectory		
FileInfoCtrl.h	15 dic 2001	223 b	unknown	djzero00	NSFileTypeRegular		
FileInfoCtrl.m	21 dic 2001	1 K	unknown	djzero00	NSFileTypeRegular		

Ricorsione sulle directory

Torno adesso in PB 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 programmazione O-O, 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 directoryContents AtPath: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 realizzati 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, 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 quanto mostrato nella 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, debbono 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, attribuendo 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.

Formattatori

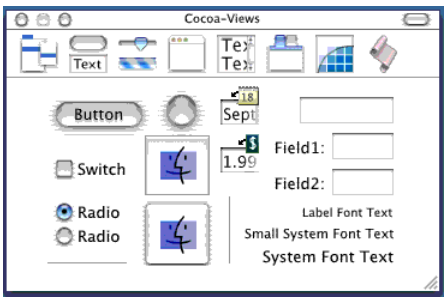
Celle e Formattatori

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, eseguendo attraverso di essi 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, adatte a rappresentare diverse unità di informazioni; ad esempio `NSTableView` contiene un elemento cella per ogni colonna e ogni cella può essere configurata in modo differente dalle altre. In questo modo 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 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'n'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 intelliggibile 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
withDefaultAttri¬
butes:(NSDictionary *)attributes
{
                return ( nil);
}
@end
```

Il secondo ed il terzo metodo, come si può vedere, non 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 OO, 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 long¬
Value]: 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. Un'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 che, 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 delle 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 rappresentato 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! Eh 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 si sottopongono loro 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 directoryContentstPath: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...

Codifica ed Archiviazione

Introduzione

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.

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 una linea 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, si 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 inseriti 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];
```

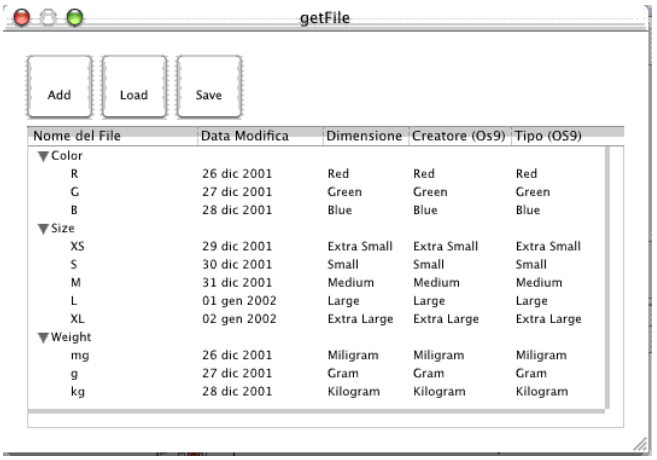
e per recuperarlo dal file basta assegnare un oggetto root:

```
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 dal 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).
I nudi fatti:

- Il file FileInfoCtrl.h
- Il file FileInfoCtrl.m
- Il file FileStruct.h
- Il file FileStruct.m
- Il file LSDataSource.h
- Il file LSDataSource.m
- Il file LSFileInfo.h
- Il file LSFileInfo.m
- Il file LSFormatter.h
- Il file LSFormatter.m

Documenti

Introduzione

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`.

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 PB), 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 dell'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 la 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 delegata, 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 menù "`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 PB), 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 facciano 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 quindi essere in grado di svolgere sono essenzialmente: 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 e 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 ed è qui dove si traccia lo stato del documento (se è stato modificato o meno).

NSWindowController

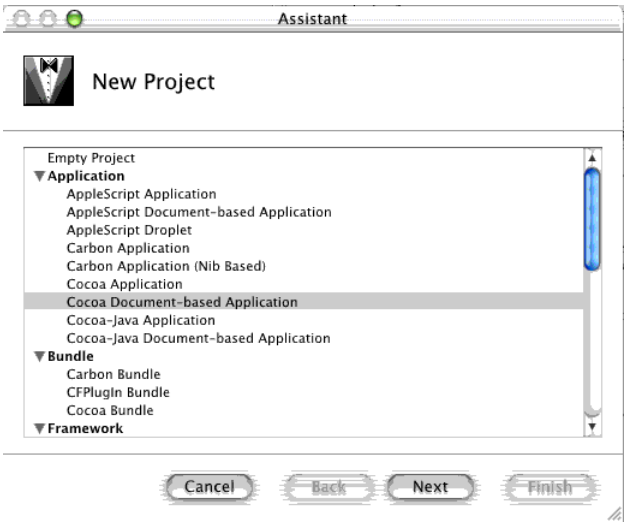
Un oggetto `NSWindowController` gestisce una finestra associata ad un documento. Un documento può avere diverse finestre associate, ciascuna con il suo `NSWindowController`. Questi oggetti non sono molto interessanti, nel senso che il comportamento di default fornisce a `NSDocument` sufficienti servizi 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.

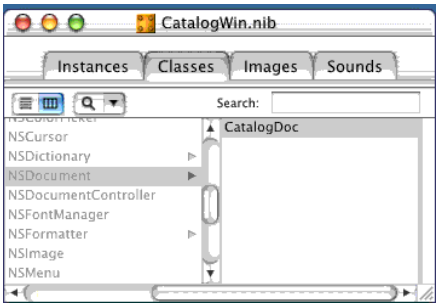
Sembra facile. Parto.

Comincio

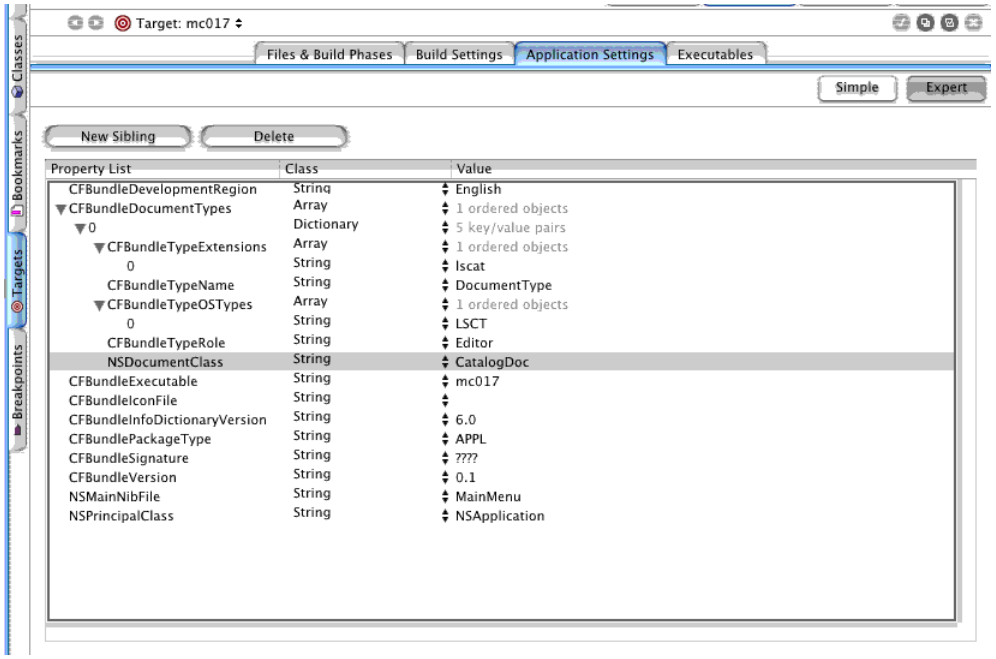
Faccio un nuovo progetto, dal pregnante nome `mc017`, e dico di farlo "`document-based`".



PB 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`). Trovo (si fa per dire: senza il libro non l'avrei mai trovato) questa cosa in PB, la finestra delle opzioni del `Target`, pannello "`Application Setting`".

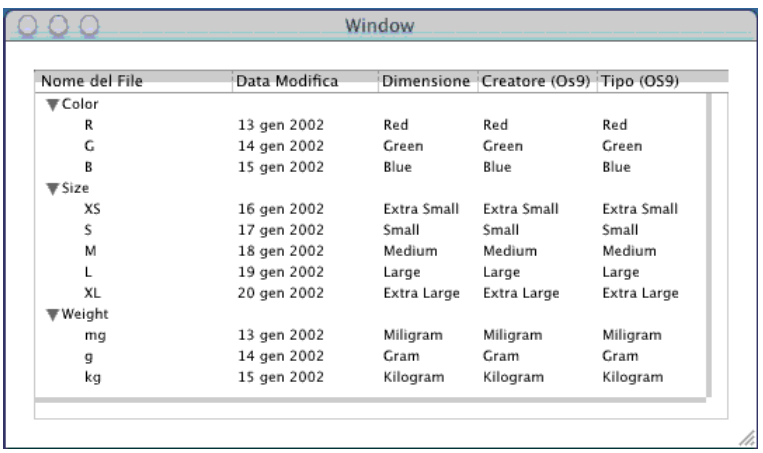


C'è un pulsante "`Expert`", lo premo; viene fuori una lista di voci. Seleziono di seguito `CFBundleDocumentTypes` e poi 0 (zero!), c'è una voce `NSDocumentClass`, la cambio, ancora una volta, in `CatalogDoc`. E questa è finita.

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).

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).



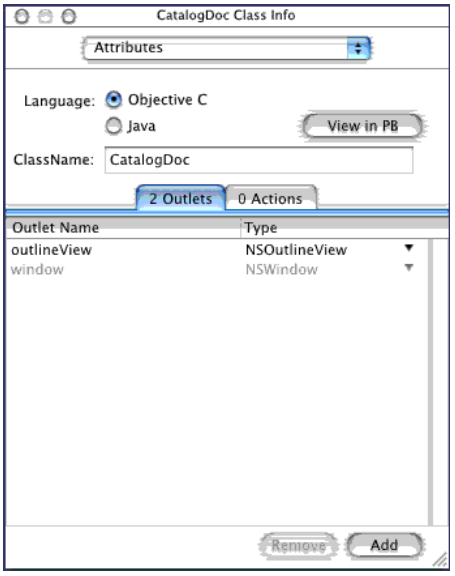
Già che ci sono, apro anche il nib principale con il menu, ed aggiungo un nuovo menu. Ci metto due voci, "Add..." e "Delete", dove la prima voce intende rimpiazzare uno dei pulsanti che avevo nelle vecchie realizzazioni dentro la finestra. I pulsanti Load e Save sono invece sostituiti dalle voci di menu 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 l'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 PB.



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 l'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, quand'è che i dati vengono inseriti all'interno della finestra, ovvero, in altre parole, quando è chiamato 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:

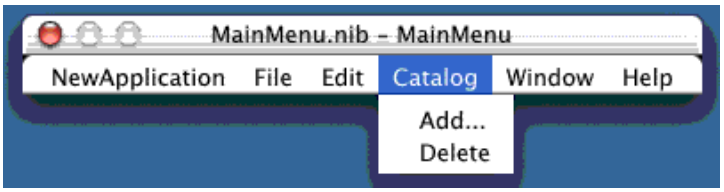
```
- (void) loadCatalogWithData: (NSData *) data
{
        if ( data )
        {
                NSMutableArray * catData = [NSUnarchiver unarchiveObjectWithData: data ];
                // assegno il tutto alla sorgente di dati
                [ dataSource setStartPoint: catData ] ;
                // 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.

Menu e Palette

Introduzione

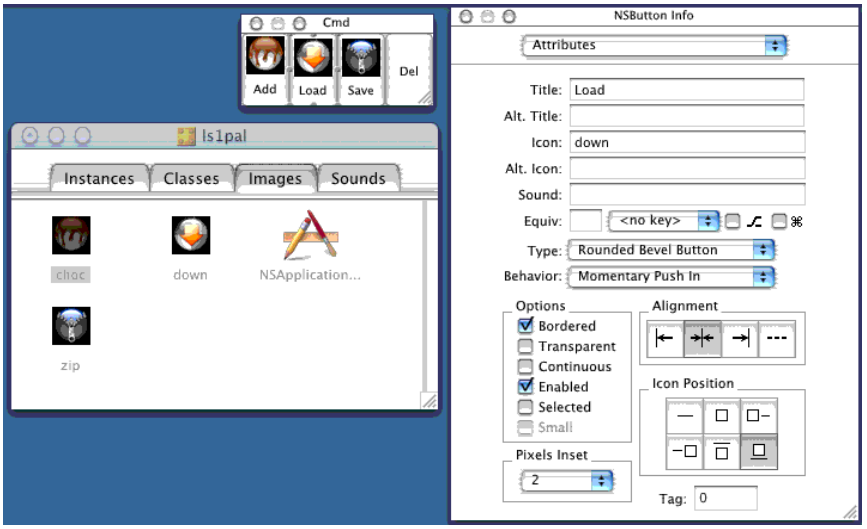
In questo capitolo aggiungo due voci di menu e una palette di comandi all'applicazione finora costruita. Il menu l'ho già introdotto il capitolo precedente, con due voci all'interno di un menu specifico; intendo con questo sostituire i pulsanti che, ancora un capitolo precedente, utilizzavo per aggiungere 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.

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").

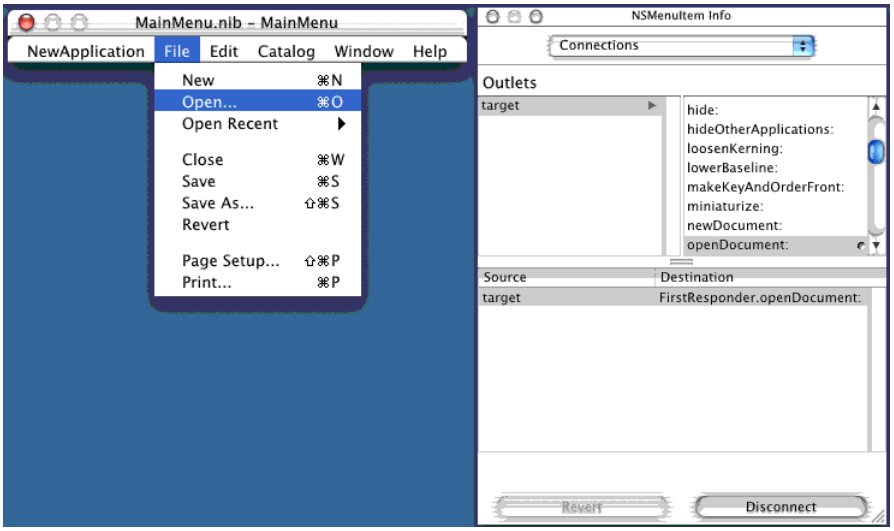


Visto che ci sono, scopro come sia molto facile aggiungere immagini e suoni ai pulsanti.
Per prima cosa, occorre importare le immagini che si intendono usare all'interno del progetto PB. Io ho recuperato dei disegni non so più da dove in formato pict.
Questi disegni sono poi disponibili anche in IB una 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`".

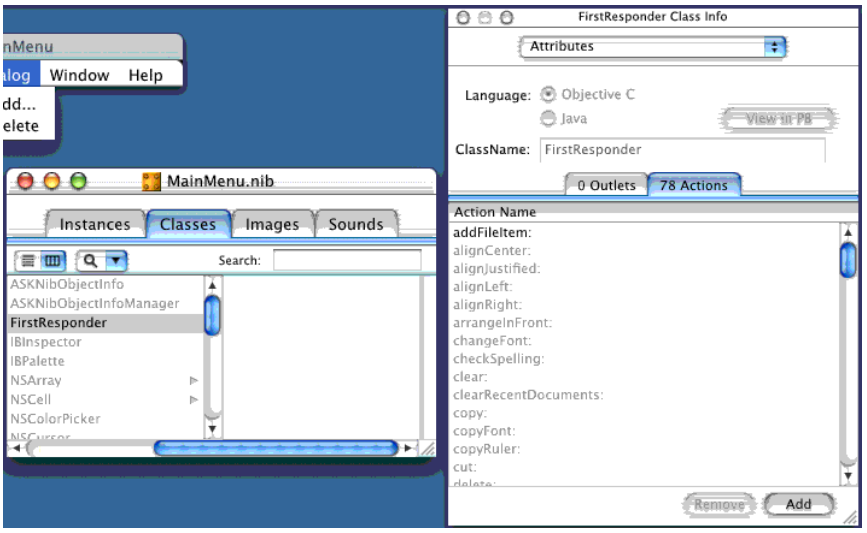
Collegamenti

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.

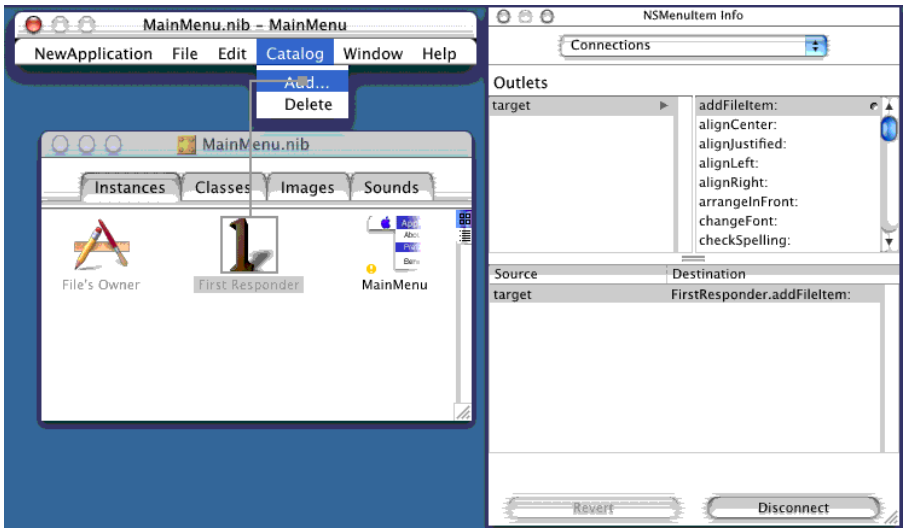


Ad esempio, la voce "`New`" ha già realizzato, nel paradigma target/action, il collegamento tra la selezione del menu e l'azione "`NewDocument`". Sì, ma a chi è diretto il messaggio? Al "`First Responder`", l'oggetto misterioso di questo paragrafo.
La piglio da lontano: una applicazione, una volta che 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 PB, e a mano aggiungo i 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:

```
intDeleteItemFromHierarchy ( 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] , ofItem ) == 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 collegare i 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 un 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

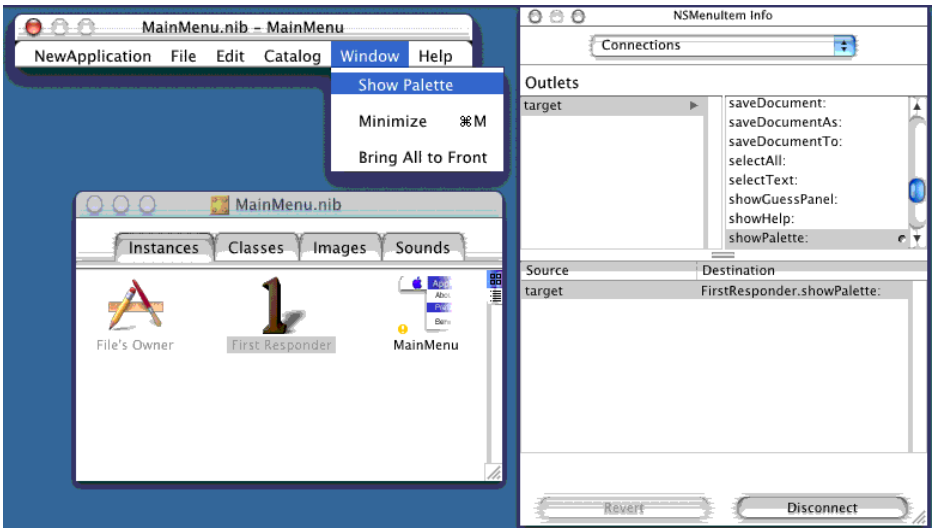
```
+ (id) sharedPalette
{
        static palette * _sharedPalette = nil ;
```

```
        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 PB 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-classedi `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.

Sesso Droga e Drag'n'Drop

Introduzione

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 interne trascinandole da una finestra del Finder all'interno di una finestra di Catalogo.

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 conseguente; 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. Quando 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 wantstypes = [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 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 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 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 istruzioni. 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 conteneva 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...

I nudi fatti:

- Il file CatalogDoc.h
- Il file CatalogDoc.m

Ricominciamo

Introduzione

Ricomincio a lavorare con Cocoa riprendendo in mano le vecchie cose, operando qualche modifica, ed accorgendomi che ci sono alcune cose nuove.

Rileggendo i file

Sono con la versione 2.0 di Project Builder, il sistema è arrivato alla versione 10.1.5, ormai dovrei avere tutta la documentazione completa ed in linea. Anche 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

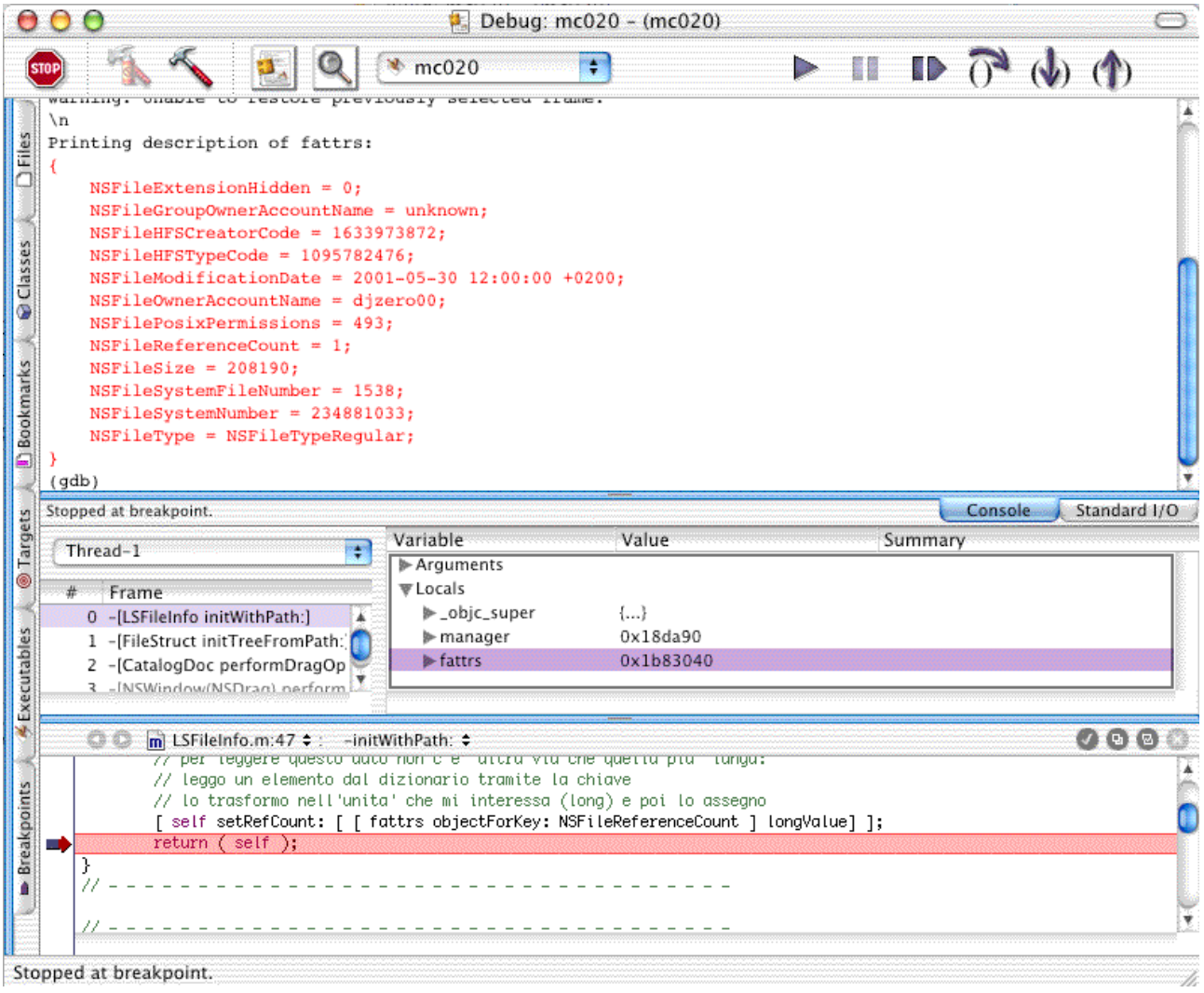
I primi quattro attributi sono facili da capire: la dimensione del file stesso, quando è stato modificato l'ultima volta, gli attributi relativi ai diritti di proprietà dell'utente e del gruppo cui appartiene l'utente. Anche gli ultimi due attributi sono chiari: si tratta dei vecchi codici utilizzati 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 dovrebbero 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, am eno che non si voglia catalogare il disco su cui si trova il sistema operativo, si troveraano solamente file normali e directory, in quanto tutti i file di sistema sono abilmente mascherati dal Finder, che non li visualizza.

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 abiatuati. Normalmente mi aspetto un valore di 1 (il file stesso), a mano che si tratti di file speciali di sistema.
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 "`Apple DVD player 2.7`"):

```
Printing description of fattrs:
```

```
{
    NSFileExtensionHidden = 0;
    NSFileGroupOwnerAccountName = unknown;
    NSFileHFSCreatorCode = 1633973872;
    NSFileHFSTypeCode = 1095782476;
    NSFileModificationDate = 2001-05-30 12:00:00 +0200;
    NSFileOwnerAccountName = djzero00;
    NSFilePosixPermissions = 493;
    NSFileReferenceCount = 1;
    NSFileSize = 208190;
    NSFileSystemFileNumber = 1538;
    NSFileSystemNumber = 234881033;
    NSFileType = NSFileTypeRegular;
}
```

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).

Fatti e formati nuovi

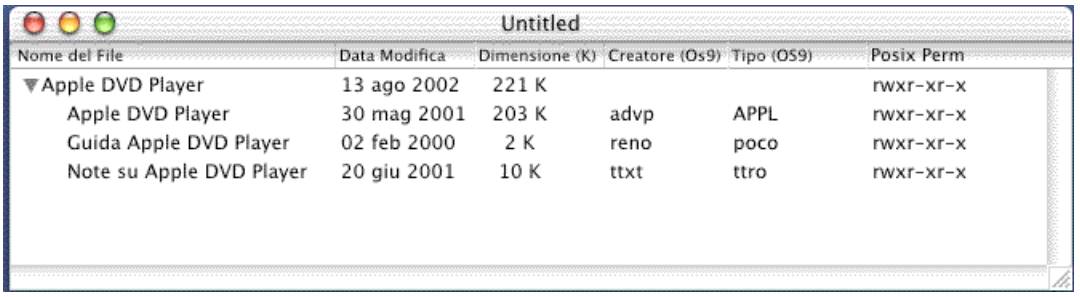
Una cosa interessante è invece il fatto che sono comparsi (o non li avevo visti prima) dei metodi di comodo per leggere elementi dal dizionario che contiene gli attributi del file file. Alcuni di questi metodi non sono nemmeno documentati, ma presenti nei file di interfaccia `NSFileManager.h`. Con tutto ciò, e tenedo 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):

```
-rwxr-xr-x  1 djzero00  unknown  208190 May 30  2001 Apple DVD Player
-rwxr-xr-x  1 djzero00  unknown    1536 Feb  2  2000 Guida Apple DVD Player
-rwxr-xr-x  1 djzero00  unknown   10606 Jun 20  2001 Note su Apple DVD Player
```

Qui si dice che il file "Apple DVD Player" può essere letto ('r'), scritto ('w') ed eseguito ('x') dal possessore `djzero00` (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 ] );
}
```

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 con tutti.
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....

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 a causa della ricorsività che ho realizzato nell'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 librerie 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 (che 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, a 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

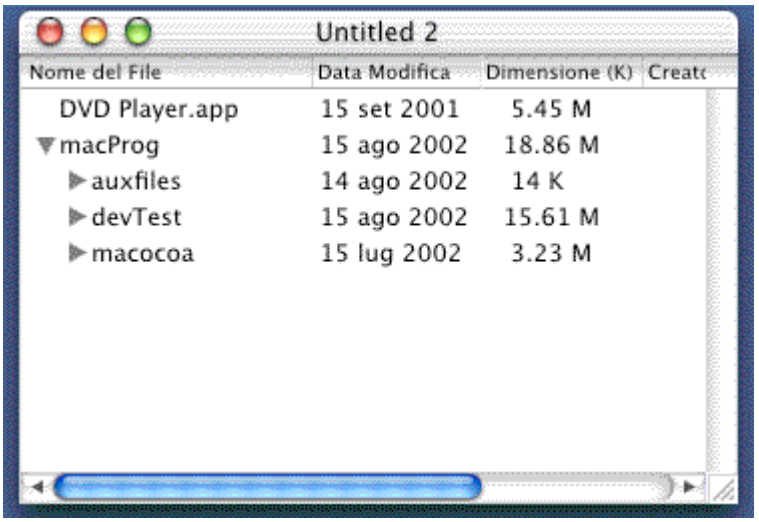
```
- (int)
    outlineView:            (NSOutlineView *)outlineView
     numberOfChildrenOfItem:    (id)item
 {
    // se l'item e' nil, stiamo parlando della radice
    if (item == nil)
        // dico quindi che ci sono tanti elementi quanti presenti nel vettore
        return( [ [self startPoint] 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, 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 la prossima puntata, cui vi rimando anche per tutti i file ed il progetto.

Preferirei di No

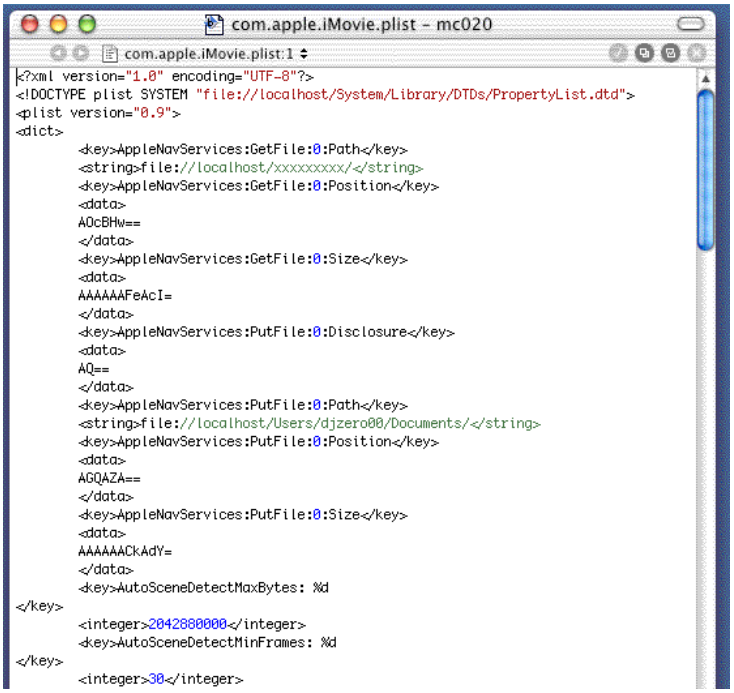
Documentazione ed esempi di Apple

Introduzione

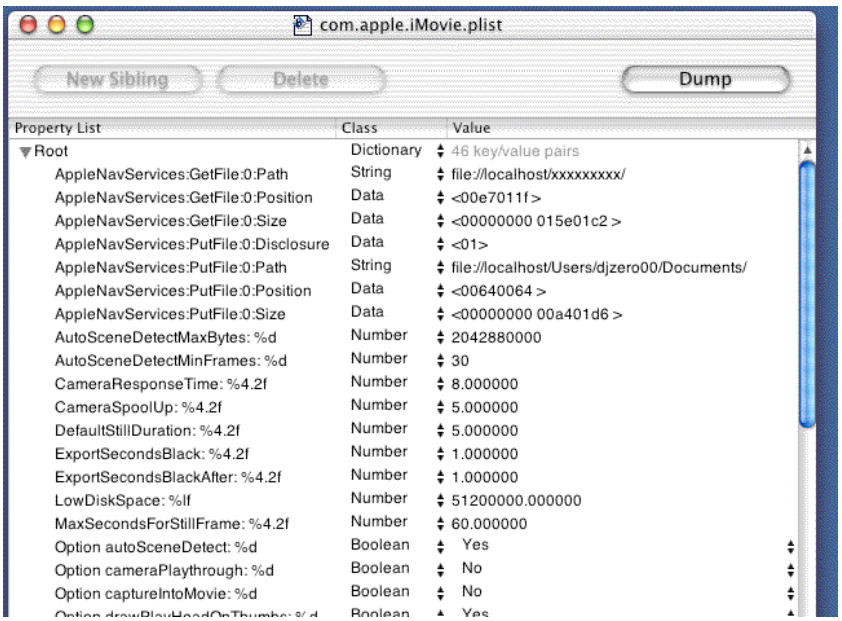
Scopo della puntata è di costruire un meccanismo (finestra e messaggistica) per l'impostazione di un sistema di preferences.

Le Preferenze

Il problema lasciato in sospeso la puntata precedente era di poter determinare il comportamento della `NSOutlineView` quando si trattava 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 Mac OS X, sono memorizzate in un file di banale testo. I documenti che raccolgono le preferences, 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.
Aprendo a caso un file di preferences (IMovie, 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.imovie.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 (anzi, meglio, nei "`Programming Topics`", l'argomento "`User Defaults`", esiste una lunga spiegazione di quali sono i valori di default, come sono cercati nei vari database, eccetera. In realtà, le cose sono molto più semplici. Basta una istruzione:

```
NSUserDefaults      *defaults = [NSUserDefaults standardUserDefaults];
```

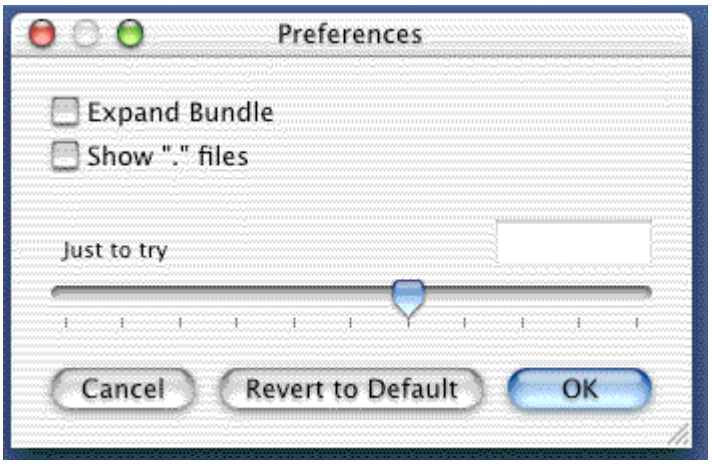
Con questa, costruisco un oggetto della classe `NSUserDefaults` che gestisce il meccanismo. Da questo, attraverso metodi appositi, sono in grado di leggere, scrivere e gestire in generale le varie preferenze presenti. Ora, una preferenza può essere un oggetto di una di queste classi: `NSData`, `NSString`, `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 (sono degli oggetti della classe dizionari). 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.

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, niente
    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;
}
```

dove 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 dizionario
        [locDict setObject: [defaultValues() objectForKey:key] forKey:key] ;
    }
}

void
readIntDefault ( 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 numberWithInt:[locDef integerForKey:key]] forKey:key] ;
    }
    else
    {
        // recupero dal dizionario def-code il valore e lo metto del dizionario
        [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 qualcosa 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 *)
```

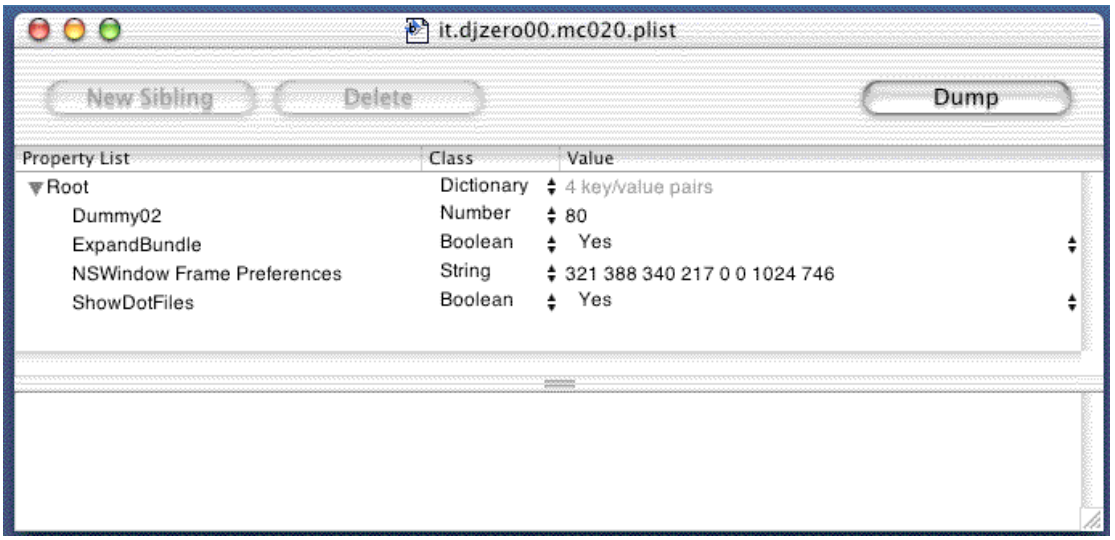
```
getPreferences
{
    return defCurrValues;
}
```

In questo modo, per accedere ad un valore di preferences, 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.



Fuori i file

I file qui collegati 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).

Informazioni e Notai

Copio, semplifico e stravolgo un esempio su "Learning Cocoa"

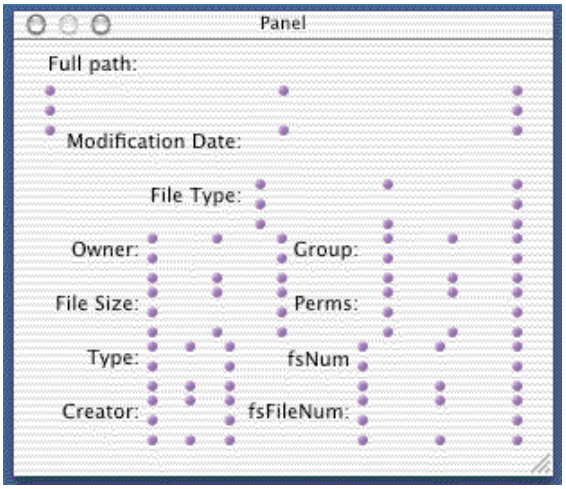
Introduzione

Scopo del capitolo è di realizzare una finestra di informazioni, ovvero una finestra all'interno della quale sono presentate tutte le informazioni relative ad un file.

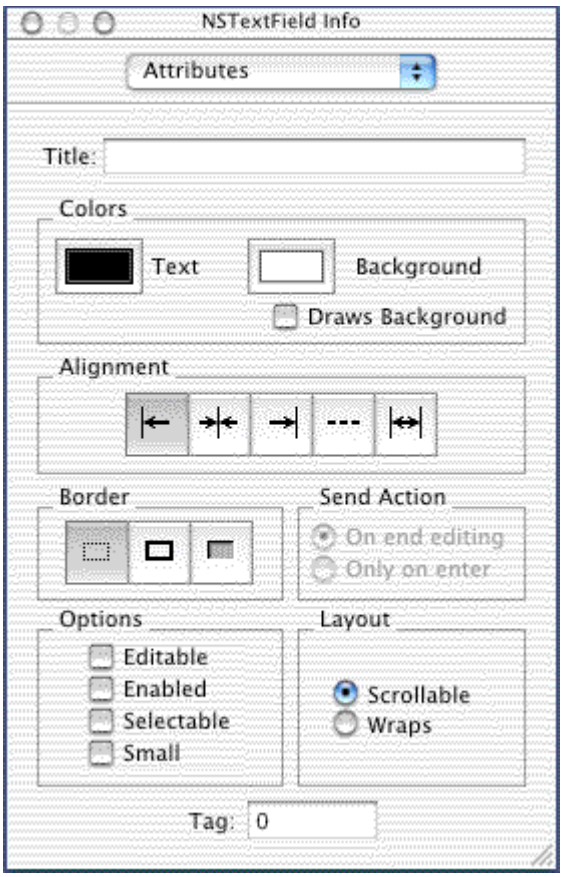
Finestra di informazioni

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.

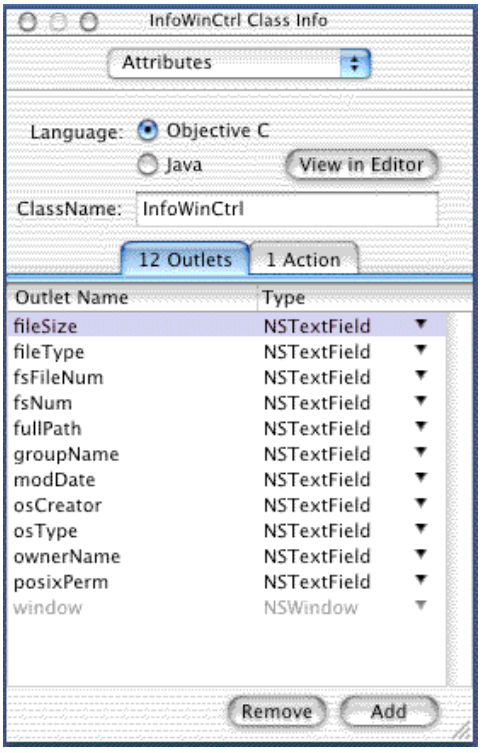
Comincio quindi a costruire la finestra di informazioni; apro IB e costruisco il nib (a proposito, ho scoperto che significa `Nextstep Interface Builder`, ovvio no?) nominato `infoPanel`.



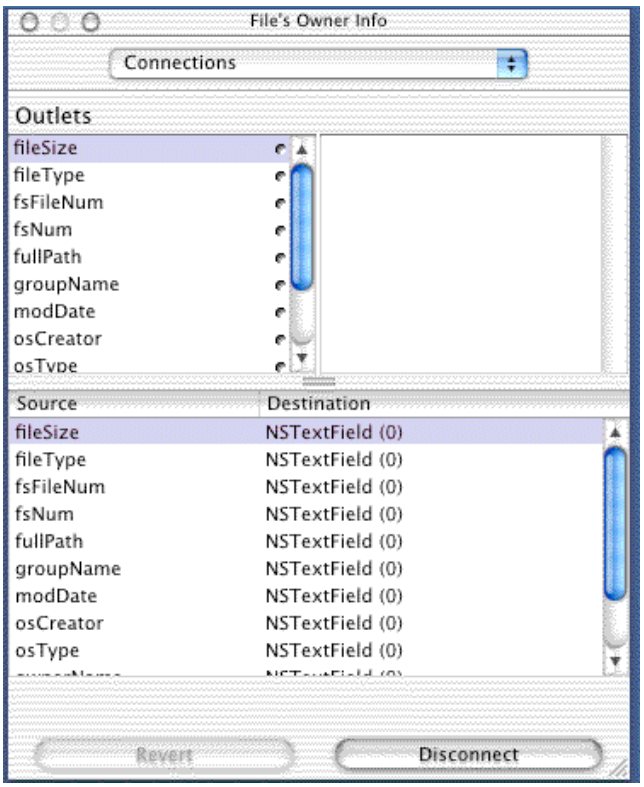
Nella figura non si vedono esplicitamente i campi di testo, ma solo dei puntini che li contornano: ho deciso di rendere il loro background trasparente (non si tratta di campi che si possono editare, dopotutto).



Ho poi definito una sottoclasse di `NSWindowController`, InfoWinCtrl, che non ho istanziato; piuttosto, ho cambiato la classe di `"File's Owner"` proprio a `InfoWinCtrl` (ricordo che il `"File's Owner"` è l'oggetto proprietario della finestra, e al momento dell'esecuzione sarà proprio un oggetto della classe `InfoWinCtrl`).
Ho collegato tramite una lunga sequenza di outlet



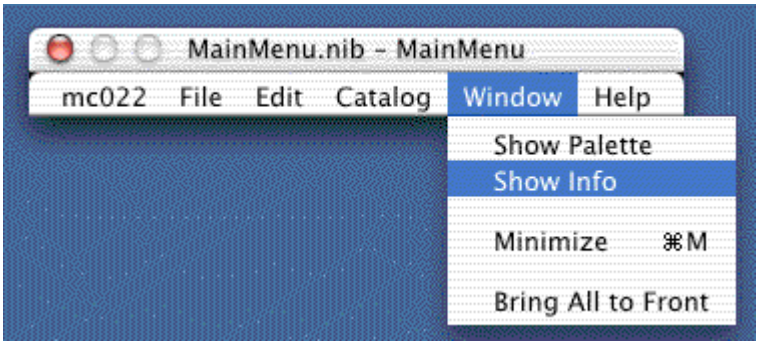
il `File's Owner` ai vari campi



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 PB 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 file 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.

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.

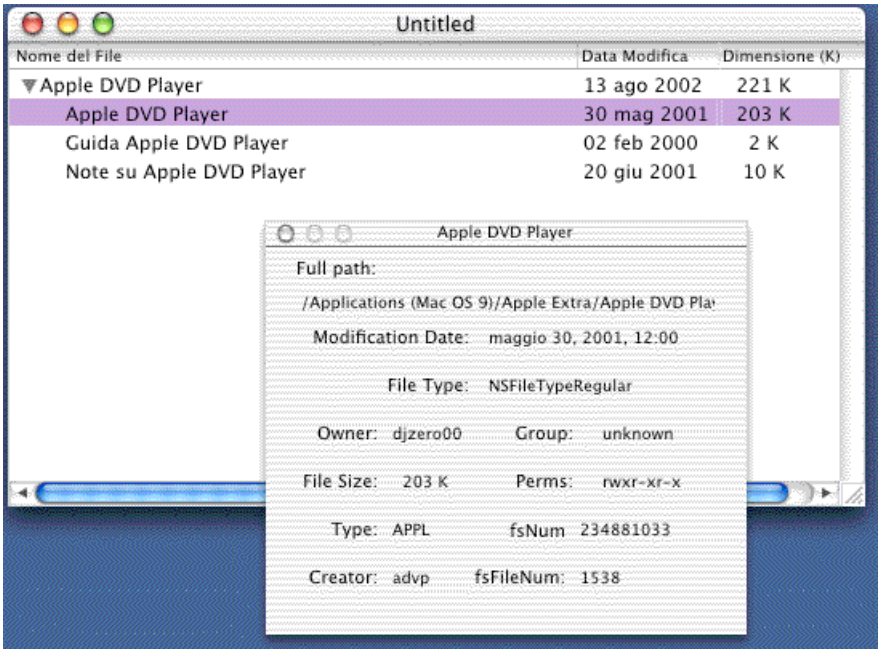
Due minuti dopo: troppo bello per essere vero. Se il file non è in linea, non solo l'icona non viene

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: [ locIt¬
emfileFullPath ]  ] )
    [[ 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.

Preferenze Rivisitate

Documentazione Apple e un po' di sudore

Introduzione

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.

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
}
```

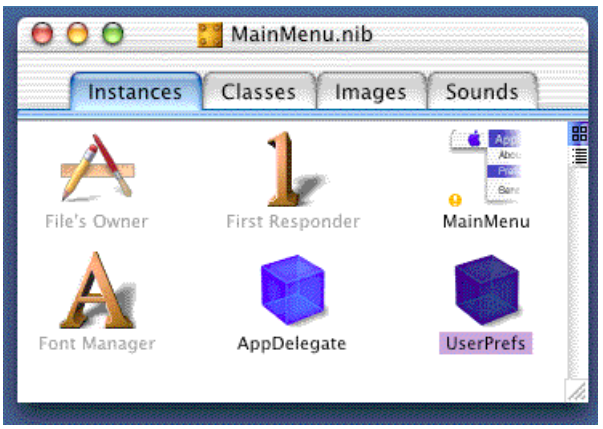
In primo luogo, l'inizializzazione ha luogo solo una volta, confrontando il valore della variabile `locSharedPref` . Il dizionario delle Preferenze costruito dal codice questa volta preleva i valori da un file (che poi si troverà all'interno del bundle dell'applicazione), per una più facile manutenzione (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.



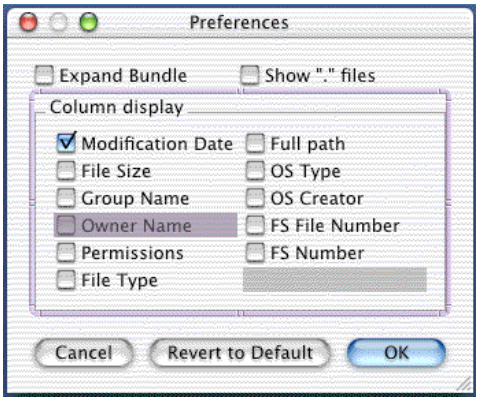
In questo modo, al caricamento del menu principale, è costruita anche l'istanza della classe `UserPrefs`.

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 PB, 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). Il fatto è che permettono di rendere il codice seguente 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 indica partono da Zero (convenzione C) piuttosto che da Uno (convenziona "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 );

    // 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 ];
}
```

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

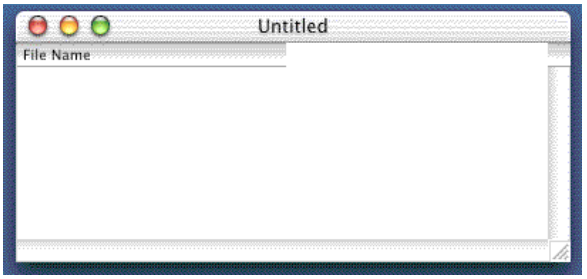
Ora che all'interno delle Preferenze c'è scritto quali colonne vogliamo siano visualizzate all'interno della finestra, occorre scrivere un po' di codice per adeguare la `NSOutlineView` a quanto richiesto. Allo scopo, ho scritto il seguente metodo:

```
- (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 per il momento 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];
}
```

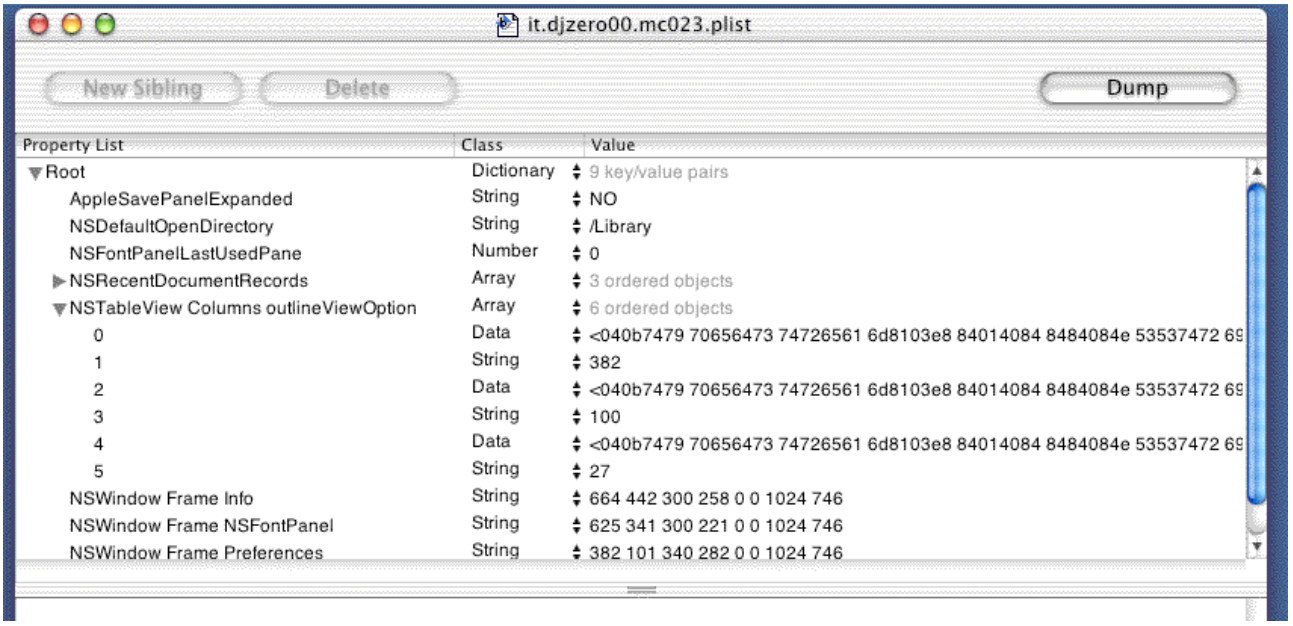
A parte la serie di scritture dei vari campi delle Preferenze, l'ultima istruzione invia appunto una notifica dal nome `PREF_UPDATE_NOTIFICATION` alla redazione; da notare come non si preoccupi per nulla del fatto che ci sia qualcuno ad ascoltare.

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.


it.djzero00.mc023.plist
New Sibling
Delete
Dump
Property List
Class
Value
Root
Dictionary
9 key/value pairs
AppleSavePanelExpanded
String
NO
NSDefaultOpenDirectory
String
/Library
NSFontPanelLastUsedPane
Number
0
NSRecentDocumentRecords
Array
3 ordered objects
NSTableView Columns outlineViewOption
Array
6 ordered objects
0
Data
<040b7479 70656473 74726561 6d8103e8 84014084 8484084e 53537472 69
1
String
382
2
Data
<040b7479 70656473 74726561 6d8103e8 84014084 8484084e 53537472 69
3
String
100
4
Data
<040b7479 70656473 74726561 6d8103e8 84014084 8484084e 53537472 69
5
String
27
NSWindow Frame Info
String
664 442 300 258 0 0 1024 746
NSWindow Frame NSFontPanel
String
625 341 300 221 0 0 1024 746
NSWindow Frame Preferences
String
382 101 340 282 0 0 1024 746

A Proposito

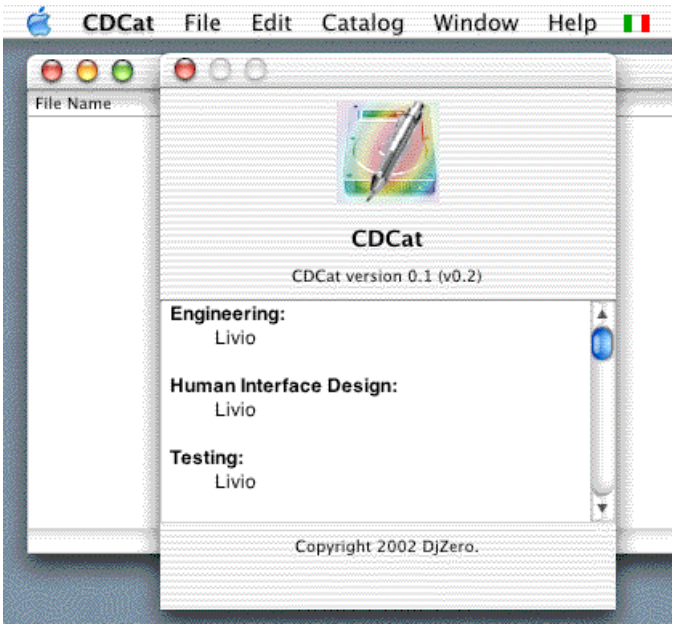
Faccio mio un tutorial su CocoaDevCentral.

Introduzione

Questo capitolo introduce molti nuovi concetti, forse frettolosamente, ma con lo scopo di arrivare ad una finestra About personalizzata ed animata.

A proposito

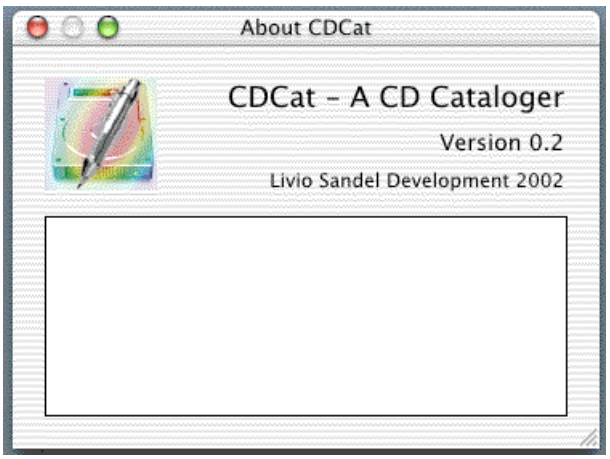
Quando si utilizza Cocoa, PB e IB per la costruzione di una applicazione, si ottengono molte caratteristiche gratuite (copia, incolla, menu, eccetera). Tra le altre, si ottiene gratis anche la finestra "About", in cui sono presentate le informazioni relative al programma.



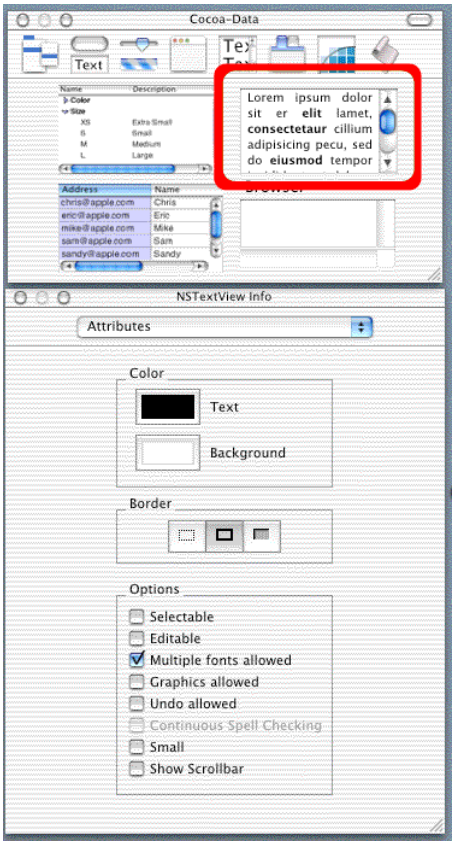
In maniera automatica Cocoa fornisce la finestra, che viene riempita dall'icona del programma (a proposito, ho cambiato l'icona del programma con quella porcheria che potete vedere in figura... se qualche anima pia mi fornisce gratuitamente una icona esteticamente più pregevole, è benvenuto) e dalle 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

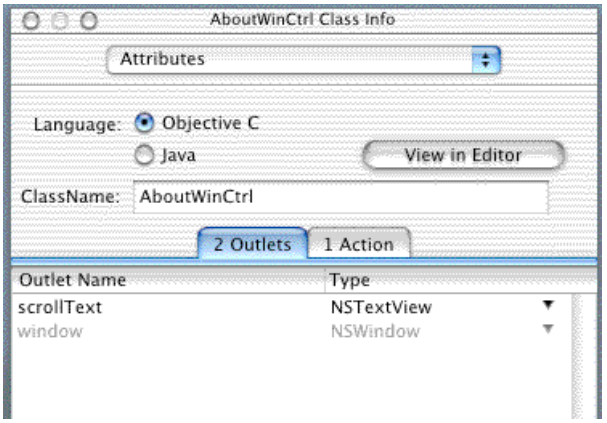
Vado in IB e costruisco la nuova finestra di `About`.



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 visualizzazione 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 ;
```

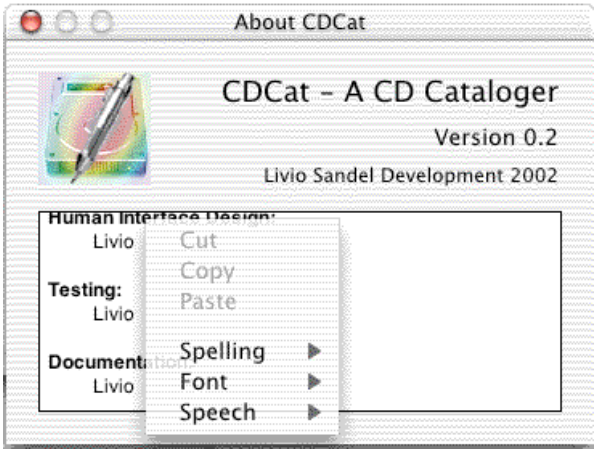
Qui di seguito carico in memoria il contenuto del file `Credits.rtf`; prima costruisco il path completo tenendo conto delle possibili localizzazioni (e quindi ricorro ai bundle), poi metto il contenuto del file all'interno di una `NSAttributedString`, piuttosto che in una normale `NSString`. Infatti 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 definizioni 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 )];
```

Questo 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`.

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 ogetto `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;
    }
```

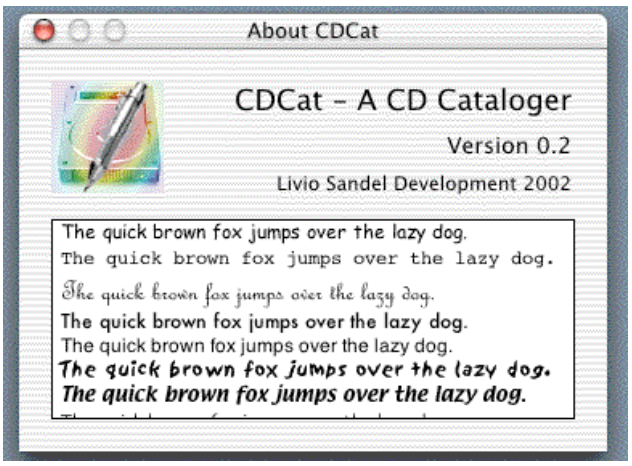
Questo è il pezzo di codice in cui si entra quando si deve cominciare a scrollare la finestra (perché è appena scaduto il tempo di attesa, oppure perché si è arrivati alla fine del giro precedente). Imposto un paio di variabili, e poi dico che la posizione di scroll è Zero. Questa posizione va data come 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.



I nudi fatti (quelli modificati):

- Il file AboutWinCtrl.h
- Il file AboutWinCtrl.m
- Il file AppDelegate.h
- Il file AppDelegate.m
- Il file CatalogDoc.h
- Il file CatalogDoc.m
- Il file PrefWinCtrl.h
- Il file PrefWinCtrl.m

AIUTO!

Documentazione Apple e CocoaDevCentral.

Introduzione

Questo breve capitolo dice come aggiungere Help all'applicazione.

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`

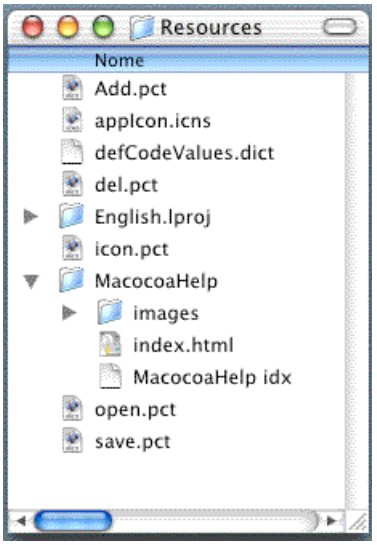
e che invece io, trovandosi nella parte relativa a Carbon, ho inizialmente colpevolmente trascurato.
Di particolare interesse la sezione "`Designing a Help Book`", dalla quale stralcio velocemente alcuni punti.
Lo Help è tipicamente formato da una pagina di titolo e da una pagina principale, dalla quale 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.

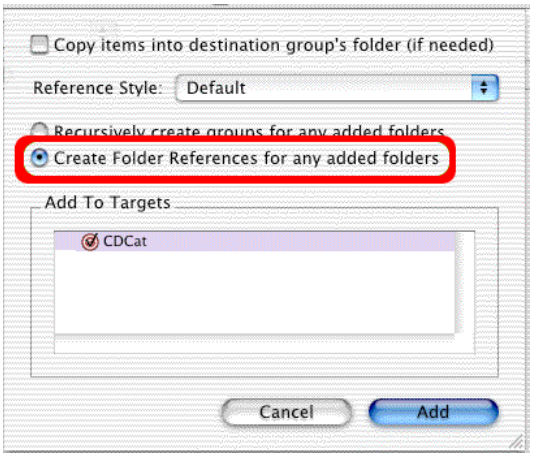


Se proprio si vuole essere professionali, occorre indicizzare lo Help, in modo che quando l'utente dall'interno dello Help Viewer esegue delle ricerche, queste possano produrre risultati anche guardando questi nuovi file di Help.

L'operazione è oltre tutto molto semplice: basta droppare l'intera cartella sull'applicazione "`Apple Help Indexing Tool`" che si trova in "`/Developer/Documentation/Apple Help`" (anche se un link è all'interno della cartella "`/Developer/Application`". Questa operazione produce un file (nel mio caso, "`MacocoaHelp idx`") già nel posto corretto.

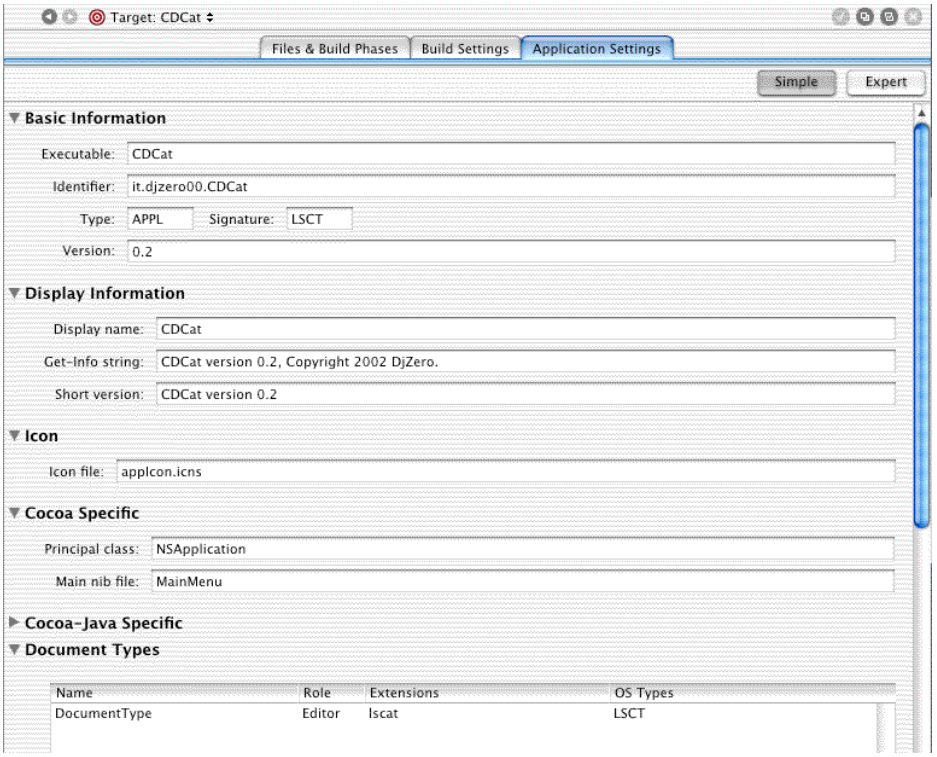
Registrazione dell'Help

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,

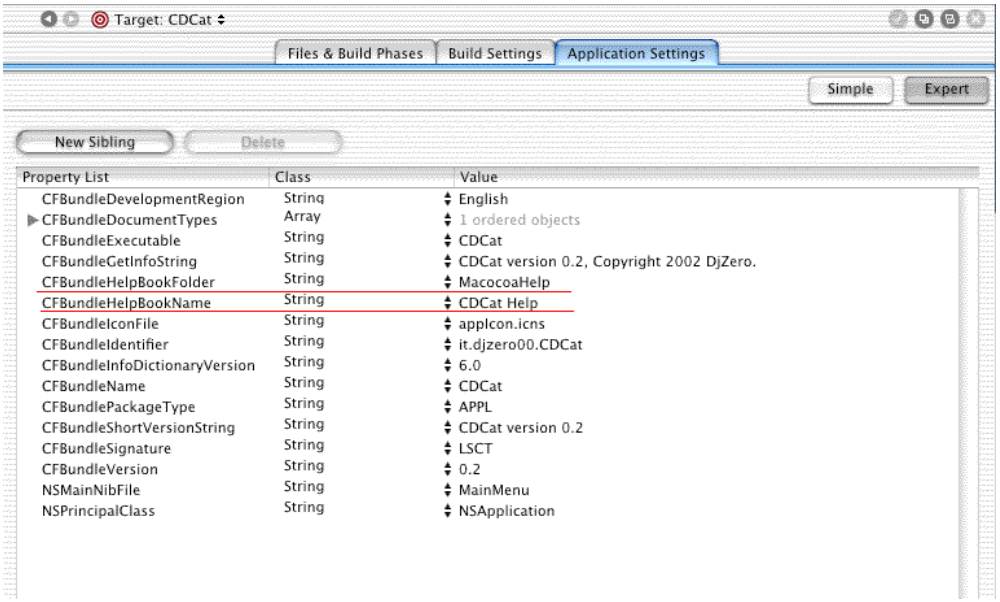


bisogna selezionare il pulsante "`Create Folder References...`"; nel mio caso, era automaticamente selezionato l'altro pulsante (che aggiunge i file al progetto,ma non nel posto giusto), cosa che proibiva il corretto funzionamento del meccanismo.
Scopo ultimo di questa operazione è di fare in modo che la cartella e l'intero suo contenuto siano copiate all'interno della cartella "`Resources`" (vedi figura sopra) 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. Normalmente, per impostare valori su questa property list si usa il tab "`Target`" della finestra principale del progetto, si seleziona il target e poi il tab "`Application Settings`" della finestra che compare.



Tuttavia, le informazioni da aggiungere qui non sono previste: occorre passare alla modalità "`Expert`", facendo clic sul pulsante apposito. Ciò che compare è una finestra 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).

Fritto Misto

Tanto lavoro di debugger; un esempio di Apple.

Introduzione

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.

Due Errori

Nel codice finora sviluppato, ci sono due errori; meglio, comportamenti non conformi quando si lavora in condizioni eccezionali. 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 puntata 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 ] ) ;
}
```

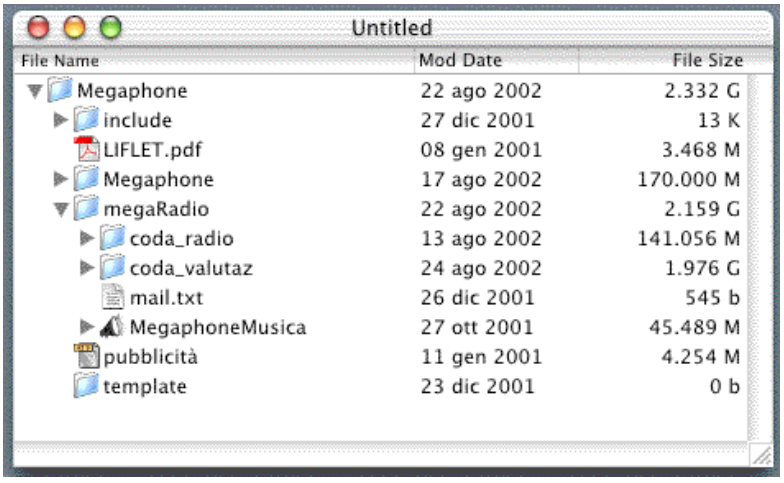
Senza saper né leggere né scrivere (ma soprattutto programmare), ho deciso di non perdere troppo tempo sull'argomento e di inserire il caso specifico della dimensione del file per... per costruire un oggetto di tipo `NSNumber`. Infatti, il metodo serve per restituire il valore di una casella della `NSOutlineView`, 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.
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 PB. Ebbene, la cosa è già fatta! Nell'esempio, il programmatore ha costruito una classe proprio a questo scopo. Copio pedissequamente i file, non il codice, 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

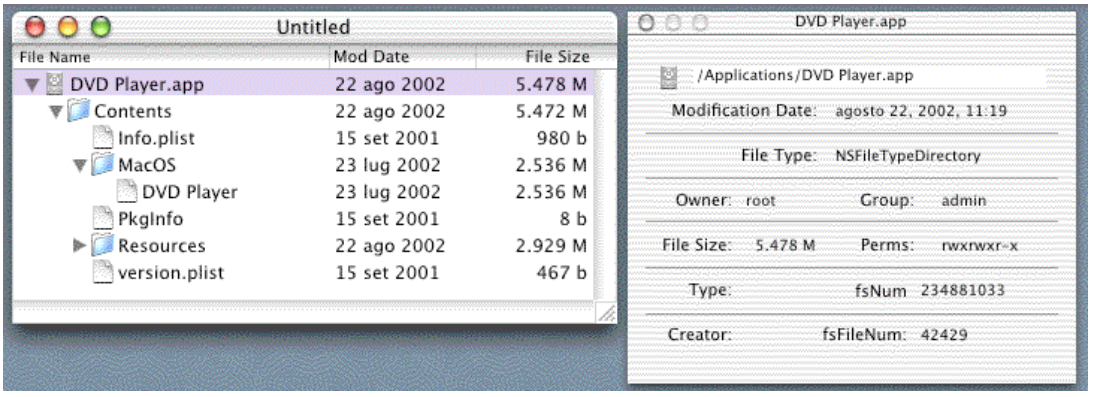
qualcuno ha provveduto ad inserire all'interno dell'oggetto come variabile d'istanza.
Per effettuare questo scambio di celle, occorre scrivere qualche riga di codice all'interno del metodo `windowControllerDidLoadNib` della classe `CatalogDoc`, lì dove si predispone il funzionamento della `NSOutlineView` (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.

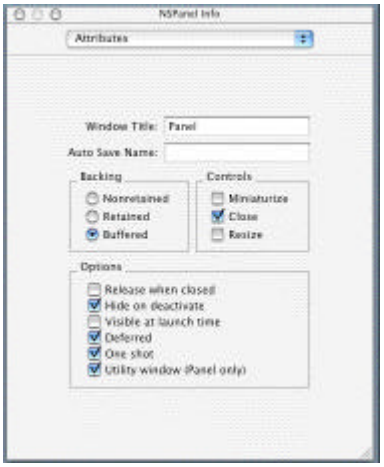
Intercettando la richiesta relativa al nome del file, inserisco in quel momento l'immagine come variabile d'istanza:

```
- (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.
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 gestioen 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 dicsco, 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:`.

Solo Volumi, per favore

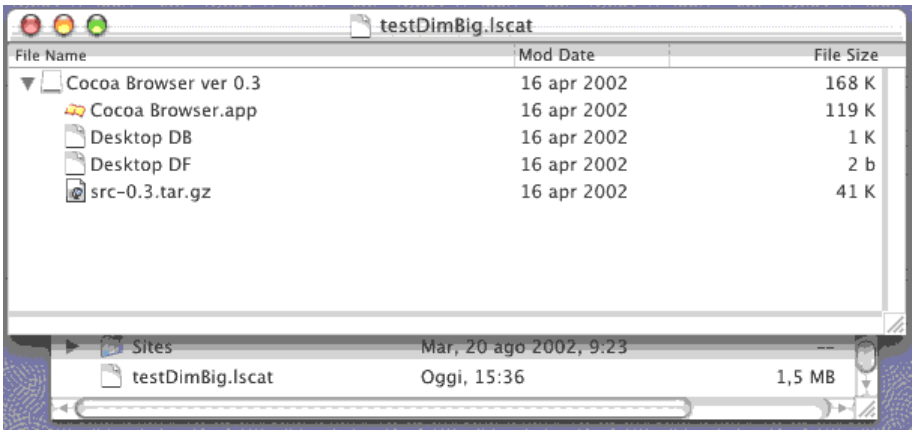
Documentazione Apple, pazienza e fantasia.

Introduzione

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.

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 .img di un programmino che ho scaricato da internet...), mi risulta un file di 1.5 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 (72K vs 1.5 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.

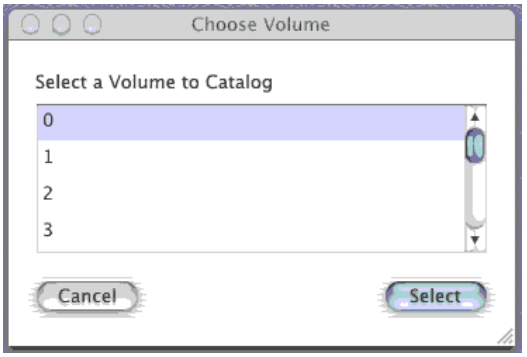
Solo volumi

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 PB 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 "nil". 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. Questo 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 ]  ];
}
```

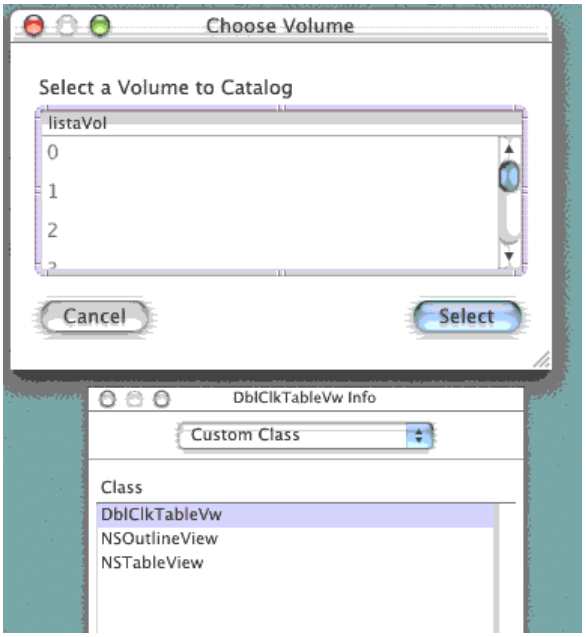
Il metodo `cancelButton:` è del tutto simile; l'unica differenza è l'argomento `returnCode`, pari a -1. L'idea infatti è che la sheet comunichi alla finestra la riga selezionata dall'utente come l'argomento `returnCode`, che corrisponde ad un elemento nel vettore variabile d'istanza. La finestra di catalogo 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 PB.
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)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 dragati 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...

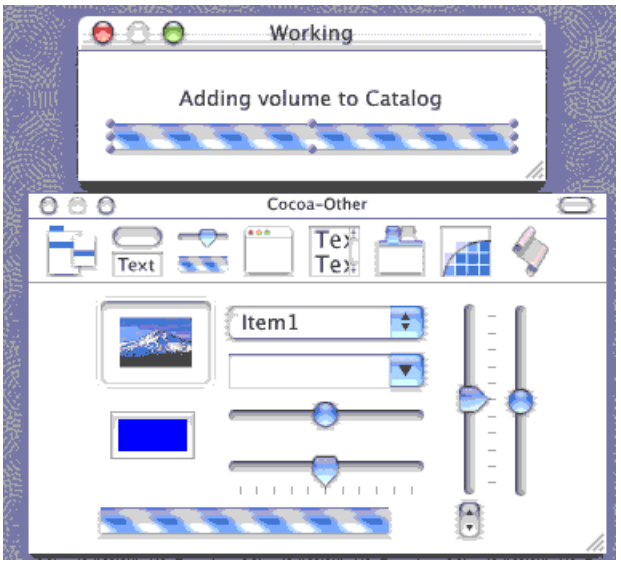
```
- (BOOL)
performDragOperation:(id)sender
{
    NSPasteboard     * dragPasteboard;
    NSArray        * pList ;

    // 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 ];
}
```

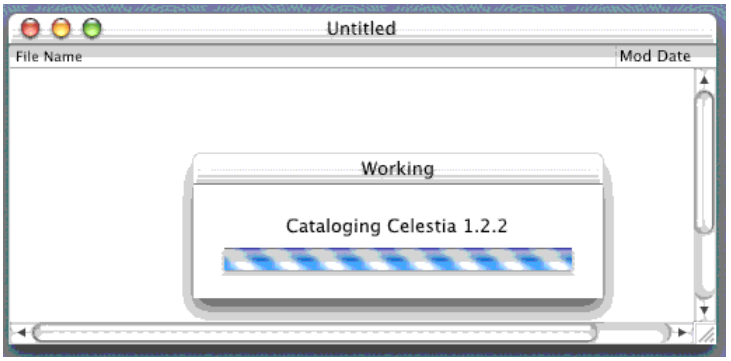
I primi due metodi si appoggiano sui metodi seguenti (non posso raggiungere gli outlet dentro i metodi di classe...):

```
- (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.

Barra degli Strumenti

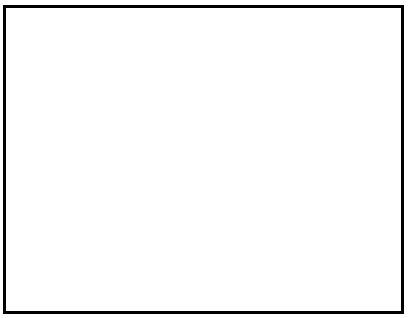
Introduzione

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"`)

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 PB.
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:

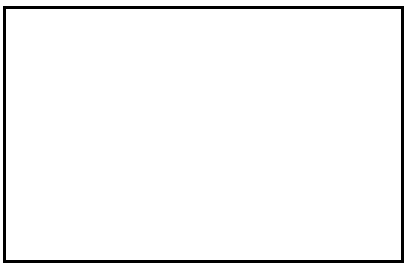
```
#define         CatDocToolbarId         @"tbName_CatDoc"
- (void)
addToolbarToWin: (NSWindow *) currWin
{
        // costruisco la nuova istanza della toolbar
        // la faccio autorelease perche' sara' poi ritenuta dalla finestra
        NSToolbar *toolbar = [[[NSToolbar alloc] initWithIdentifier:
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:... .

Ordine, ordine

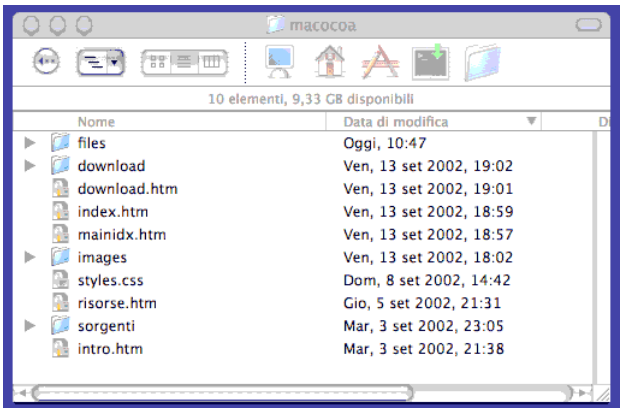
Introduzione

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

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

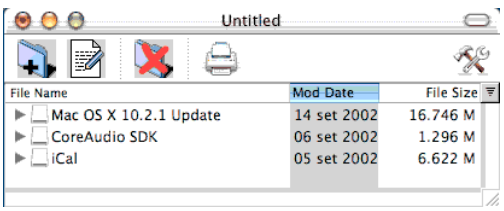
Una volta che ho scritto le istruzioni per l'ordinamento degli elementi da visualizzare, occorre informare la visualizzazione di come procedere all'ordinamento. In altre parole, devo trovare il posto più adatto per assegnare un valore alla variabile d'istanza `orderColumn` della classe sorgente di dati.
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:

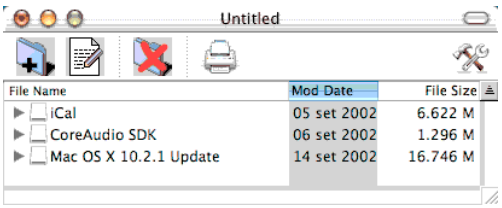
```
- (BOOL)
outlineView:(NSOutlineView *)outlineView
        shouldSelectTableColumn:(NSTableColumn *)tableColumn
{
        // assegno la nuova chiave di ordinamento al dataSource
        [[ self dataSource] setOrderColumn: [ tableColumn identifier]];
        // rinfresco la finestra con i nuovi dati
        [ [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`.

Un ordine migliore

Introduzione

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

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:

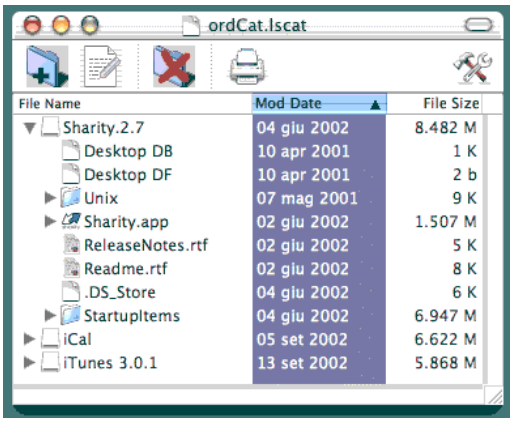
```
- (id)outlineView:
(NSOutlineView *) outlineView
        child:
                (int) index    ofItem:
                (id) item {    NSArray * newArray ;
        NSMutableArray * tmpArray ;
        if (item == nil)
        {
                // ordino gli elementi secondo criterio
                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...

Ricerca

Introduzione

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.

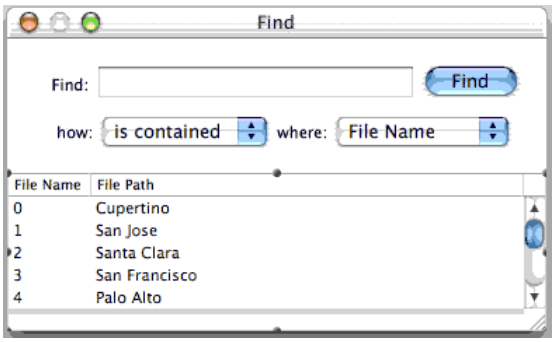
Sorgenti: nessuna

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.

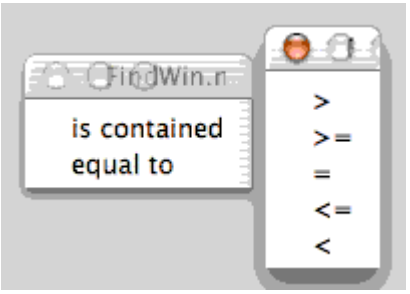
La finestra

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.

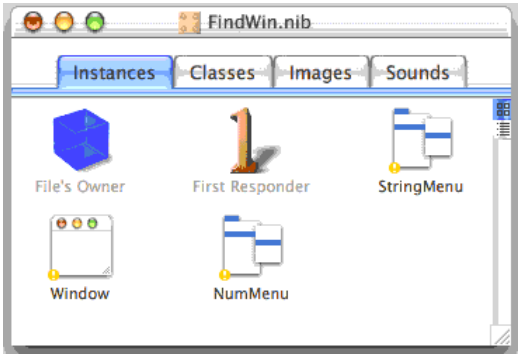


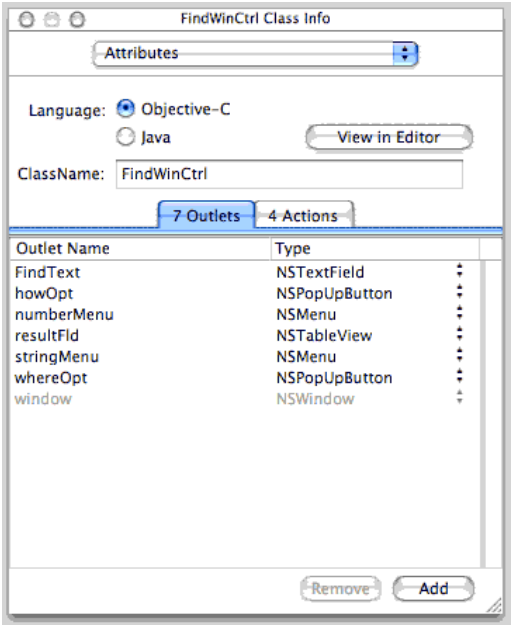
Già così, comincio ad avere dei problemi con l'interfaccia. Il primo problema è il comportamento dei vari elementi al ridimensionamento, che non mi piace per nulla. Allora, ho racchiuso il menu

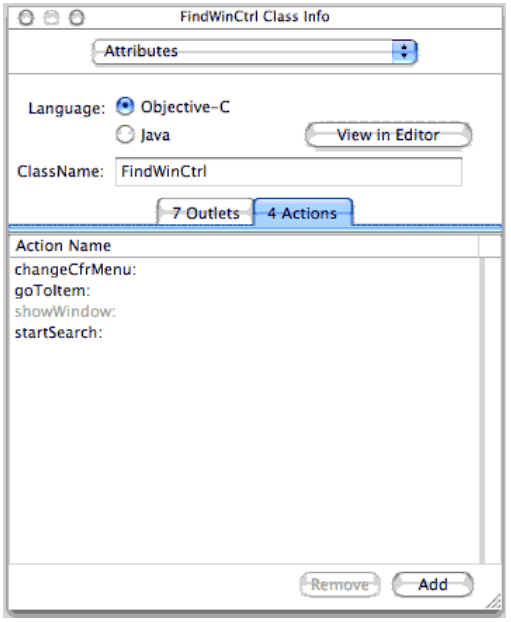
pop-up e la sua stringa di riferimento all'interno di una `NSBox`, completamente trasparente e senza bordi (praticamente invisibile). In questo modo, le specifiche di dimensionamento dinamico generano un comportamento che mi piace di più.
Il secondo problema è come associare il giusto menu di confronto; 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.

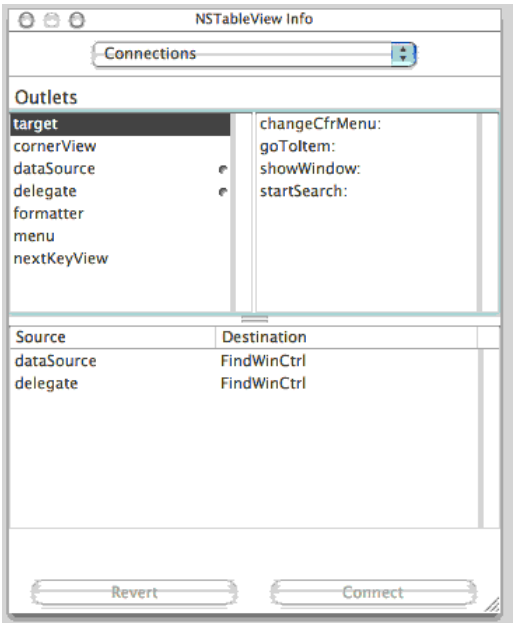


C'è poi il problema di capire quale voce di menu è selezionata correntemente. Scelgo il metodo del "`tag`"; ad ogni voce di menu assegno in IB un identificatore numerico nel campo "`Tag`" disponibile nella finestra di informazioni relativa alla voce. Poi, da programma, recupero il tag usando l'apposito 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 PB 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:

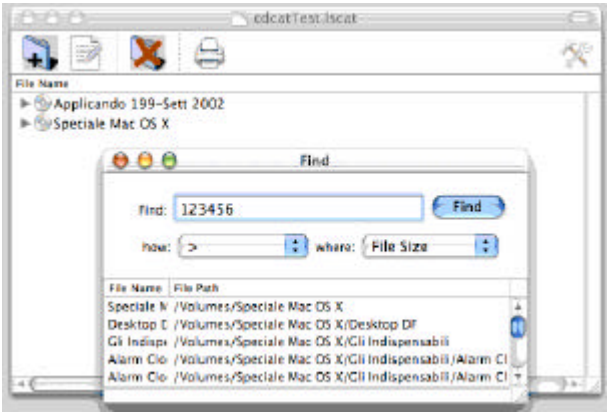
```
BOOL
checkSearch( LSFileInfo * item, id searchItem, NSString * colId, int opt )
{
        // colonne che contengono stringhe
        NSArray *id_string = [NSArray arrayWithObjects:
                COLID_FILENAME, COLID_FULLPATH, COLID_GROUPNAME,
                COLID_OWNERNAME, COLID_FILETYPE, nil ];
        // 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:

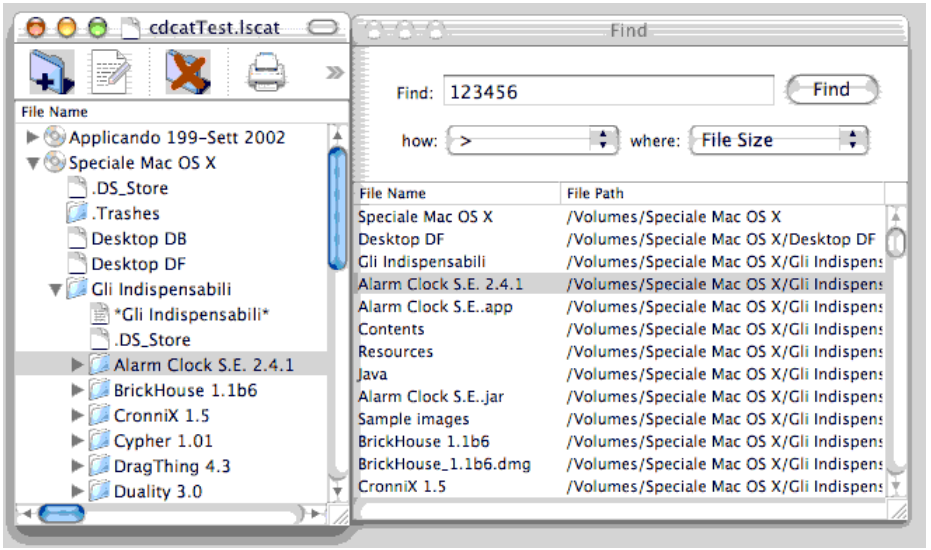
```
- (void)
expandDStoItem: (FileStruct*) expItem
        inOutlineView: (NSOutlineView *) outView
{
        // recupero il path completo dell'elemento obiettivo
        NSString        * fPath = [expItem fileFullPath] ;
        // lo spezzo ordinamente nei vari elementi
        NSEnumerator    * pathComps = [[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.