Marco Coïsson

Introduzione a Cocoa

Coïsson Editore

Introduzione a Cocoa

Piano dell'opera

Benvenuti alla terza e ultima puntata di questa avvincente saga. Dopo esserci occupati del linguaggio C e del linguaggio Objective-C, ora usiamo le conoscenze accumulate per affrontare non già un linguaggio di programmazione, ma un "ambiente" di sviluppo, o *framework*, Cocoa per l'appunto. L'argomento è potenzialmente immenso. Se avrete voglia di andarvi a cercare un po' di bibliografia, o anche solo di sfogliare quella presentata nell'apposita sezione di questo manuale, vi renderete conto che i testi che cercano di coprire in modo esaustivo i vari aspetti di Cocoa riempiono molte centinaia di pagine. Qui, naturalmente, non possiamo pensare di affrontare uno sforzo così grande. In fondo, questa è solo un'introduzione. Pertanto qui non copriremo tutto lo scibile esistente su Cocoa, ma cercheremo piuttosto di dare le basi, quelle conoscenze fondamentali per poter affrontare da soli lo sviluppo di semplici programmi Cocoa e lo studio, all'occorrenza, della documentazione fornita da Apple sulle classi implementate da Cocoa o della letteratura (in inglese) esistente sull'argomento.

Come da tradizione, i primi due capitoli sono più "filosofici" e inquadrano l'argomento, spiegando che cosa sia Cocoa, che cosa può fare e non può fare, come si può usare Cocoa e perché usare Cocoa e non un altro ambiente di sviluppo. A partire dal capitolo 3, invece, affronteremo i rudimenti di Cocoa, un pezzo per volta, usando Objective-C come linguaggio per interfacciarci con Cocoa. Avremo bisogno di XCode e di InterfaceBuilder per sviluppare i nostri programmi, pertanto, se ancora non li avete installati, fatelo subito!

Dopo i numerosi esempi di cui era costellata la prima puntata di questa trilogia, e i molto meno numerosi esempi che caratterizzavano la seconda puntata, qui torniamo all'antico proponendo una vasta collezione di programmi che verranno presentati nella loro interezza e commentati. Tutti gli esempi che forniremo saranno dei dimostratori, nel senso che non faranno nulla di veramente utile se non mostrare, spero con un certo dettaglio, le classi fondamentali (tra cui quelle per implementare l'interfaccia utente Aqua di MacOS X) di Cocoa. Malgrado la loro apparente "inutilità", gli esempi saranno comunque programmi completi e funzionanti e non semplici porzioni di codice, così da essere il più possibile uno strumento utile per il lettore che col tempo si troverà a dover affrontare, da solo, la realizzazione da zero del proprio primo programma che faccia uso di Cocoa.

Tutti gli esempi riportati sono stati realizzati con XCode 1.5 e MacOS X 10.3.5.

Che cosa faremo

In questo manuale studieremo i concetti di base di programmazione in Objective-C usando l'ambiente Cocoa fornito da Apple per lo sviluppo di applicazioni native per MacOS X. Gli argomenti saranno affrontati partendo dalle basi, senza dare per scontata nessuna conoscenza su Cocoa. Saranno invece richieste una conoscenza del C e dell'Objective-C, che è come dire che i primi due volumi di questa trilogia sono un prerequisito per la lettura di quest'ultima puntata. Alla fine del manuale avremo coperto tutti gli aspetti essenziali della creazione di un'applicazione Objective-C/Cocoa e le classi principali di Cocoa.

Che cosa non faremo

Non studieremo tutte le cassi che Cocoa ci mette a disposizione, né commenteremo in dettaglio tutti i metodi delle classi che studieremo, limitandoci a descrivere quelli che ci torneranno utili. Il lettore troverà, auspicabilmente, in questo manuale l'aiuto necessario per affrontare da solo lo studio dei metodi e delle classi che qui non sono stati trattati, servendosi dell'ottima documentazione che Apple fornisce con XCode nonché *on-line*, o di un buon libro di testo sull'argomento.

1. Che cos'è Cocoa

Librerie e Contesti

Parlando del C nella prima puntata di questa trilogia, abbiamo speso qualche buona parola in favore delle *librerie*: avevamo detto che esse sono una collezione di *funzioni* ben documentate che possono essere usate dall'interno dei nostri programmi per eseguire determinati compiti. Questo è molto comodo, perché anche se il programma che stiamo sviluppando fa qualche cosa di assolutamente innovativo e rivoluzionario, sicuramente la maggior parte dei suoi compiti, dal più insignificante al più complesso, non saranno in realtà niente di nuovo: infatti esso dovrà leggere e scrivere dati sul disco, accettare l'input da tastiera e dal mouse da parte dell'utente, scrivere dei messaggi a schermo, disegnare finestre, pulsanti e menu, emettere suoni e chissà quant'altro. Tutti questi aspetti di dettaglio del programma sono fondamentali, perché senza di essi il nostro programma non esisterebbe, eppure non dobbiamo preoccuparci troppo di loro, perché possiamo contare su un certo numero di librerie che ci nascondono la maggior parte del "lavoro sporco" che sta dietro ad ognuna di queste operazioni, permettendoci di compierle in una o due righe di codice. È infatti grazie a questa semplicità che noi possiamo combinare in maniera produttiva e originale questi "mattoncini" di cui è costituito il nostro programma, fino a fargli fare qualche cosa di mai visto prima, o comunque di utile.

Potete pensare a Cocoa come se fosse una libreria. Ma in realtà non è esatto. Apple chiama Cocoa un *framework* mentre una libreria, in inglese, si chiama *library* (che poi vuol dire *biblioteca* e non libreria, e messa così ha anche molto più senso, perché una library è una collezione di funzioni consultabili e utilizzabili, così come una biblioteca è una collezione di libri consultabili). Sì, ma che cos'è un framework? In italiano potete chiamarlo *struttura*, ma io, almeno per quanto riguarda la programmazione in Cocoa, preferisco chiamarlo *contesto* (anche se poi continuerò a chiamarlo framework).

Facciamo conto che stiate scrivendo un libro di cucina. Ci state mettendo tutta la vostra creatività e la vostra abilità tra i fornelli. Questo libro di cucina vi renderà famosi e farete un sacco di soldi. Ma come faranno i vostri lettori ad emulare le vostre gesta culinarie? Come potranno, seguendo le istruzioni riportate nelle vostre ricette, ripdodurre quelle delizie? È semplice, serviranno loro due requisiti: il primo è la conoscenza della tecnica di cucina, il secondo è avere una cucina attrezzata e una dispensa fornita. A quel punto avranno tutto ciò che serve loro per eseguire il programma (seguire la ricetta). Che cosa c'entra la cucina attrezzata in tutto questo? È il framework. Per eseguire il programma (cucinare le uova al tegamino) vi servono le uova (ovviamente), il tegamino, il fornello, ecc… Non importa se il tegamino o il fornello del lettore del vostro ricettario sono diversi dal vostro, perché il *contesto* è lo stesso, ovvero quello di una cucina.

Con Cocoa è la stessa cosa. Il programma che devo sviluppare avrà bisogno di disegnare una finestra sullo schermo e metterci dento dei pulsanti e dei campi di testo coi quali l'utente interagirà. Il mio programma non si curerà minimamente dei dettagli di tutto ciò. Semplicemente dirà a Cocoa che ha bisogno di una finestra, che dentro ci deve mettere pulsanti e campi di testo, e per favore gli faccia sapere quando l'utente clicca da qualche parte che sia utile (ad esempio su un pulsante). Quando l'utente eseguirà il programma, il contesto sarà lo stesso, perché tutte le installazioni di MacOS X includono Cocoa, e il programma potrà essere eseguito allegramente senza il minimo problema.

Ecco che cos'è Cocoa. È più di una libreria, perché non si limita a fornire delle funzionalità precotte ma scollegate l'una dall'altra che possiamo usare semplicemente con poche righe di codice. È un contesto, perché è qualche cosa di molto più ampio. Restando nell'analogia precedente, il tegamino è una libreria, ma l'intera cucina attrezzata è Cocoa.

Vi è venuta fame? Fatevi uno spuntino, poi tornate qui e vi dico finalmente che cos'è Cocoa.

Ora sappiamo perfettamente che cos'è un framework, ma ancora non sappiamo *a che cosa serve* Cocoa. È presto detto: a dotare un'applicazione sviluppata per MacOS X di tutte quelle meraviglie che ci offre l'interfaccia utente (Aqua), come finestre, menu, pulsanti, icone, campi di testo, pannelli, barre di scorrimento, barre di progressione, mouse, scorciatoie da

tastiera e via discorrendo. Per quanto innovativo o rivoluzionario possa essere il vostro programma, sicuramente avrà bisogno di almeno alcuni di questi elementi di interfaccia. Per evitare che dobbiate reinventarveli voi ogni volta che volete sviluppare un nuovo programma, e per far sì che ci sia sempre un'interfaccia grafica coerente che non spiazzi l'utente, Cocoa vi mette a disposizione tutto questo (e molto altro ancora) affinché voi lo possiate usare liberamente e con la maggiore semplicità possibile. Inoltre, vi offrirà avanzati meccanismi di gestione dei file, delle preferenze, delle localizzazioni (traduzioni delle applicazioni in altre lingue), dei timer, delle notifiche, delle stringhe di testo, delle array, della comunicazione tra più processi o task, ecc… Insomma, una vasta collezione di funzionalità non banali ma fondamentali che sono lì per voi, affinché, con uno sforzo minimo, possiate usarle senza pensarti troppo, lasciandovi la libertà e il tempo di concentrarvi effettivamente su ciò che vi interessa, ovvero su quanto di specifico farà il vostro programma.

Linguaggi per Cocoa

Come dicevamo, Cocoa è un framework, quindi in quanto tale non dipende strettamente dal linguaggio che scegliete di utilizzare per programmare. In linea di principio, quindi, potete programmare su MacOS X usando il framework Cocoa con qualunque linguaggio vi venga in mente. In effetti le cose non stanno esattamente così. Esiste un linguaggio privilegiato, quello con cui interagire con Cocoa è più facile, non fosse per altro che Cocoa stesso è scritto in quel linguaggio ed è pensato appositamente per essere usato con esso. Se durante la lettura della seconda puntata di questa trilogia non stavate dormendo, avrete capito che si tratta dell'Objective-C.

Ad oggi, Objective-C è il linguaggio di programmazione che garantisce la massima fruibilità e con la massima semplicità di tutte le meraviglie che ci sono messe a disposizione da Cocoa. Se volete sviluppare un'applicazione per MacOS X che faccia uso di Cocoa, prendete in seria considerazione la possibilità di usare Objective-C.

Naturalmente, esso non è l'unico linguaggio. Esiste infatti la possibilità di creare dei *bridge*, dei ponti, tra linguaggi di programmazione non direttamente supportati da Cocoa e Cocoa stesso. È quanto ha fatto Apple per Java, ad esempio, che può efficacemente essere usato per interagire con la maggior parte delle funzionalità di Cocoa. È quanto fanno altri bridge che, con limitazioni più o meno forti, consentono l'uso di Cocoa dall'interno di programmi scritti in AppleScript, Perl, C, C++, BASIC di vario tipo ecc…

Quindi, anche se non amate Objective-C, questo volume potrebbe comunque esservi utile, perché quello che impareremo qui, anche se sarà applicato usando Objective-C, sarà trasportabile anche al vostro linguaggio preferito, con la sintassi che vi verrà proposta dal bridge che dovrete utilizzare.

Siete caldi? Avete voglia di scrivere il vostro primo programma? Allora fatevi una doccia fredda e datevi una calmata. C'è ancora un breve capitoletto introduttivo, prima di buttarci a pesce nella programmazione sfrenata.

2. Perché Cocoa

MacOS X e Cocoa

Piaccia o no, Apple ha scelto di inserire Cocoa tra i framework inclusi con MacOS X. Ce ne sono tanti, di framework, ma Cocoa copre un ruolo speciale, perché infatti è uno dei due che si occupa di fornire alle applicazioni sviluppate per MacOS X il supporto per l'interfaccia grafica, la memoria protetta, il meccanismo degli eventi, l'input/output da tastiera, mouse e file, e un sacco di altre comodità (per chi fosse curioso, l'altro framework preposto a questo è Carbon)[1]. Se decidete di sviluppare applicazioni native per MacOS X, Cocoa sarà probabilmente la scelta raccomandata. Apple stessa suggerisce agli sviluppatori di usare Cocoa ovunque sia possibile, riservando l'uso del framework Carbon al "trasferimento" delle vecchie applicazioni per MacOS 9 ad un ambiente comunque nativo per MacOS X e allo sviluppo di alcune applicazioni o funzionalità molto specifiche che (ancora) non sono supportate da Cocoa. In buona sostanza: se dovete sviluppare una nuova applicazione da zero, e se la vostra applicazione non deve fare cose troppo strane (non stiamo per ora a sottilizzare su quali siano queste cose strane), Apple vi consiglia caldamente di usare Cocoa. Non sarò Apple, ma ve lo consiglio anch'io. Naturalmente, potete usare il linguaggio che volete (purché abbia un bridge per Cocoa), noi useremo l'Objective-C.

Per la sua stessa natura, Cocoa è strettamente vincolato a MacOS X. In altre parole, non troverete Cocoa su un sistema operativo diverso da MacOS X. Darwin (il "cuore" OpenSource di MacOS X), Windows, FreeBSD, le varie incarnazioni di Linux, VMS, qualunque altro sistema operativo vi venga in mente *non* dispongono di Cocoa. Quando decidete di usare Cocoa, fate una scelta radicale: sapete fin da principio che l'applicazione che sviluppate potrà funzionare*solo* su MacOS X; portarla su un altro sistema operativo vorrà dire mettere pesantemente mano al codice almeno per tutti quegli aspetti che riguardano l'interfaccia utente ed altre funzionalità che potreste aver sfruttato.

Di contro, Cocoa vi garantisce che la vostra applicazione sviluppata oggi per MacOS X "Panther" funzionerà anche domani con la prossima versione di MacOS X, e anche tra 10 anni quando il sistema si sarà evoluto in maniera tale da essere quasi irriconoscibile. Inoltre, la vostra applicazione beneficerà automaticamente di tutte le migliorie che Apple avrà nel frattempo apportato al sistema operativo e, di riflesso, a Cocoa stesso. Infatti, se la vostra applicazione usa Cocoa per eseguire un certo compito, e se domani Apple, con un aggiornamento di sistema, migliora sensibilmente le prestazioni e le funzionalità di quel compito, la vostra applicazione ne beneficerà automaticamente senza bisogno né di essere modificata né di essere anche solo ricompilata.

Cocoa e altri ambienti

Naturalmente Cocoa non è l'unico ambiente disponibile. Carbon è un altro, come dicevamo, fornito dalla stessa Apple. Anche con Carbon potete realizzare applicazioni dotate di interfaccia grafica, ma in maniera molto più complicata rispetto all'uso di Cocoa. Se siete nuovi della programmazione, non complicatevi inutilmente la vita con Carbon e pensate a Cocoa, piuttosto. Anche le applicazioni scritte usando il framework Carbon sono vincolate inscindibilmente a MacOS X, ma altri ambienti vi offrono alternative più "trasportabili". Java, ad esempio, è dotato di librerie che consentono di sviluppare interfacce grafiche "interpiattaforma", ovvero che vanno bene per una varietà di sistemi operativi diversi. Simili servizi sono offerti da RealBASIC e forse da altri ambienti ancora. In questi casi i programmi che sviluppate possono essere portati con poche modifiche o addirittura nessuna anche su altri sistemi operativi. Lo svantaggio però è che difficilmente fruirete automaticamente delle migliorie che Apple potrebbe apportare al suo sistema operativo, e sicuramente dovrete fare i conti con un'interfaccia grafica mai perfettamente identica ad Aqua, il che farà sì che molti utenti

[1] Vi ho mentito, c'è un terzo framework che si occupa di queste cose (ma non di fornire un'interfaccia grafica), ed è CoreFoundation, ma il suo uso è molto specialistico (sviluppo di driver per periferiche, di kernel extensions, di applicazioni da eseguire in *single user mode* o in fase di login ecc…).

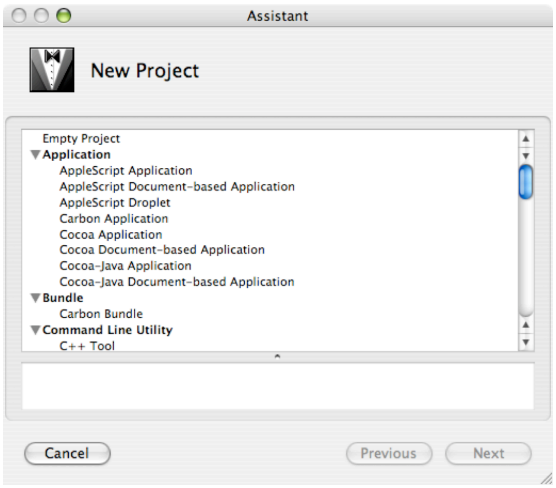
guarderanno con diffidenza e sospetto la vostra applicazione e potrebbero essere tentati di non usarla.

Per finire, alcuni ambienti di sviluppo concorrenti di XCode, come Metrowerks CodeWarrior, offrono soluzioni altamente professionali per lo sviluppo di programmi per MacOS X, facendo uso di specifici programmi (editor di interfaccia, editor di codice, compilatore, linker ecc…) ma offrendo simultaneamente la possibilità di usare in maniera completa ed efficace i framework di Apple, come, indovinate un po', Cocoa. Se siete degli appassionati ma non traete profitto dalla vostra attività di programmazione, XCode e gli strumenti gratuiti messi a disposizione da Apple saranno più che sufficienti. Se siete dei programmatori o sviluppatori professionisti e ritenete che un ambiente di programmazione come CodeWarrior sia più consono alle vostre esigenze, chiedetevi perché state leggendo questo manualetto e non un serio testo di programmazione per Cocoa.

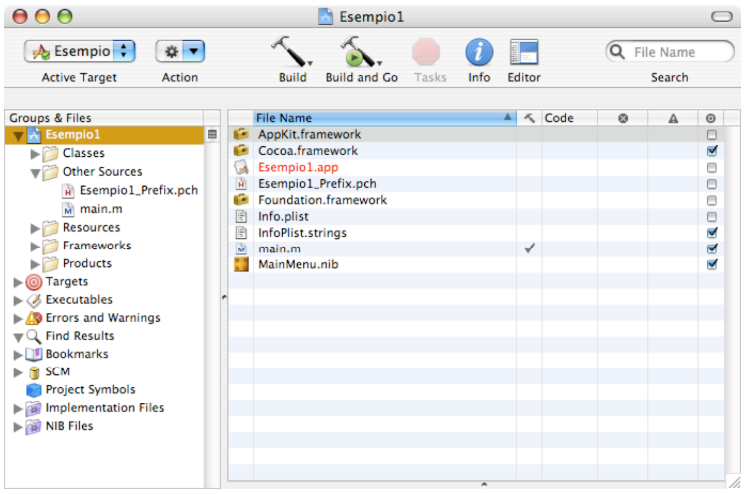
3. Campi di testo e pulsanti

Outlet e Azioni

È giunto il momento di conoscere per la prima volta quello che diventerà presto il nostro migliore amico: XCode (con suo fratello InterfaceBuilder). Si tratta di programmi messi a disposizione da Apple con i suoi DeveloperTools; sono programmi gratuiti e di qualità certamente professionale, anche se esistono ambienti di sviluppo ancora migliori (come probabilmente Metrowerks CodeWarrior).

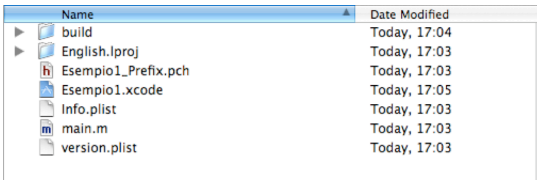


L'uso di XCode e di InterfaceBuilder richiede un minimo di assuefazione: bisogna entrare nello spirito del gioco. Se non ci si riesce, la costruzione di un programma Objective-C che faccia uso di Cocoa resterà sempre un grande mistero. Invece non c'è nulla di terribilmente complicato, ma non bisogna farsi spaventare. Per cui, almeno adesso che siamo all'inizio, faremo tutto un passettino per volta. Iniziamo pertanto con l'aprire XCode, che se avete fatto un'installazione standard dei DeveloperTools sta in /Developer/Applications.
Come prima cosa, XCode vi chiede quale tipo di progetto vogliate creare (si veda la figura di fianco).
Della varietà di opzioni che ci viene offerta, sceglieremo *Cocoa Application*. Non ci addentreremo qui nelle varie altre possibilità, limitandoci ad osservare che la categoria *Application* ci permette comunque di creare applicazioni AppleScript, Carbon, Cocoa e Java. Della differenza tra *Cocoa Application* e *Cocoa Document-based Application* non ne discuteremo qui, purtroppo. La scelta di creare una *Cocoa Application* fa sì che XCode generi automaticamente tutto quello che serve alla nostra applicazione per funzionare, tranne, naturalmente, il codice specifico che toccherà a noi scrivere.

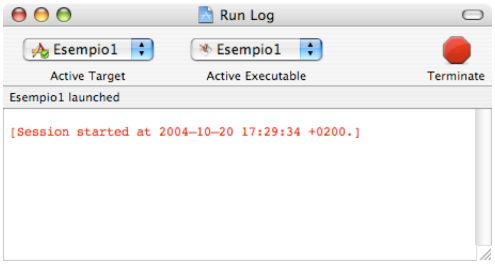


Selezioniamo quindi *Cocoa Application* e clicchiamo su *Next*. Ci verrà chiesto di dare un nome al progetto (chiamiamolo *Esempio1*, senza spazi): una cartella col nome del progetto verrà automaticamente creata sul disco al percorso indicato (potete modificarlo, se volete). Il risultato dell'operazione che avete appena fatto è mostrato qui di fianco. La vostra finestra del progetto è organizzata in una colonna di sinistra che mantiene, in una struttura ad albero, tutto ciò che compete al progetto (file sorgenti, risorse, eseguibili ecc.), mentre nel pannello di destra compaiono dei dettagli relativi all'oggetto che avete selezionato. La struttura ad albero sulla sinistra è almeno in parte modificabile, nel senso che ai livelli interni a quelli del progetto (*Esempio1*, nel nostro caso) potete aggiungere file da raggruppare come più vi piace, creando nuove cartelle. Questa struttura infatti è soltanto una comodità per lo sviluppatore che può così organizzare i propri file, ma non si riflette affatto nella struttura che i file hanno sul disco. Infatti, la cartella *Esempio1* che XCode ha appena creato per noi ha un aspetto ben diverso (si

veda l'immagine qui di fianco). Possiamo distinguere, oltre al file del progetto vero e proprio (*Esempio1.xcode*), il file di codice creato automaticamente da XCode (*main.m*), qualche file ausiliario di cui ci occuperemo in seguito, e due cartelle. La prima, *build*, conterrà i *prodotti finali*, ovvero l'applicazione eseguibile vera e propria (quando l'avremo compilata), insieme con i vari file temporanei di cui XCode avrà bisogno durante la compilazione. L'altra, invece, contiene le *risorse* dell'applicazione (ci torneremo tra un po'), specificamente quelle relative alla localizzazione inglese. È un fatto che XCode parte dal presupposto che, quando sviluppate un'applicazione, vogliate innanzitutto crearla in inglese. Il processo di localizzazione è in realtà semplice, per cui questa assunzione un po' autoritaria non è in realtà troppo male. Inizialmente, per evitare inutili complicazioni, svilupperemo le nostre applicazioni in italiano usando comunque le risorse contenute in *English.lproj*. Questo perché tanto i nostri esempi non saranno certo applicazioni di livello tale da meritare la distribuzione. Quando, più avanti, parleremo di localizzazione, la cosa più semplice da fare sarà creare la prima bozza dell'applicazione in inglese, per poi localizzarla in tutte le lingue previste (ad esempio l'italiano), e poi proseguire con lo sviluppo in parallelo per tutte le lingue.



La cosa commovente di tutto ciò che abbiamo fatto fino ad ora è che l'applicazione *Esempio1* è già funzionante. Provate a raggiungere il menu *Build* di XCode e selezionate il comando *Build and Run* (la scorciatoia è mela-R, conviene impararla perché la useremo spesso). Partirà la fase di compilazione, e dopo qualche secondo la vostra nuova applicazione comparirà in tutto il suo splendore: come vedete è dotata di una finestra (vuota), di una barra dei menu (con tanto di menu *Servizi* attivo!), la finestra può essere spostata e minimizzata e ridimensionata, e i principali comandi da menu sono già funzionanti. Insomma, considerato che non abbiamo ancora scritto una sola riga di codice non è poi male.

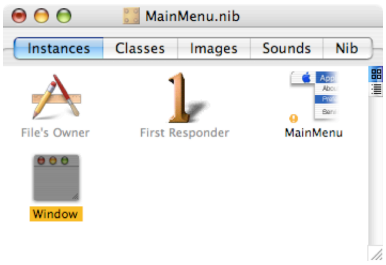


Se ci avete fatto caso, ha fatto la sua comparsa sul monitor del Mac anche un'altra finestrella, quella mostrata qui di fianco. Essa sarà una delle nostre più grandi amiche, grazie a lei impareremo un sacco di cose! Sì, ma che cos'è? È una finestra di *log*, ovvero una finestra in cui XCode riporta tutte le informazioni essenziali riguardanti ciò che sta facendo o ha fatto la nostra applicazione. Ad esempio ci sta dicendo che l'applicazione è stata avviata ad una certa data e ad una certa ora. Quando selezionerete il comando *Quit* dal menu *Esempio1* della nostra applicazione, la finestra di log vi segnalerà che l'applicazione ha terminato la sua esecuzione. Di già che ci siamo, notate il segnale rosso di stop identificato come *Terminate*: ecco, tutte le volte che per un qualunque motivo l'applicazione che state sviluppando si inchioda e non risponde più ai comandi, potete "ucciderla" o "terminarla" cliccando su quell'icona.

Visto che siamo qui che non facciamo altro, fatevi un giro sul Finder, nella cartella del nostro esempio. Aprite la cartella *build*. Avete notato? È comparsa l'icona di un'applicazione generica, stranamente denominata *Esempio1*. Che sarà mai? Sì, avete indovinato. È l'applicazione che avete appena creato, la vostra prima applicazione Cocoa!

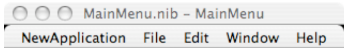
Così com'è, *Esempio1* non fa molto. Dobbiamo far sì che faccia qualche cosa di più interessante. Per fortuna non è difficile. Iniziamo a decidere che cosa farà il nostro programma. Ad esempio potremmo realizzare una sofisticatissima calcolatrice che esegue solo addizioni: l'utente scrive in un campo di testo il primo addendo, in un altro campo di testo il secondo addendo, poi premendo un pulsante il risultato viene scritto in un terzo campo di testo (non editabile). Che questa applicazione non serva assolutamente a nulla è chiaro, e che sia pure progettata in un modo un po' insolito per essere una calcolatrice è evidente, ma qui lo scopo è didattico. Se impariamo a fare questa semplice operazione, abbiamo imparato concettualmente come funziona *qualunque applicazione Objective-C/Cocoa*, e scusate se è poco.

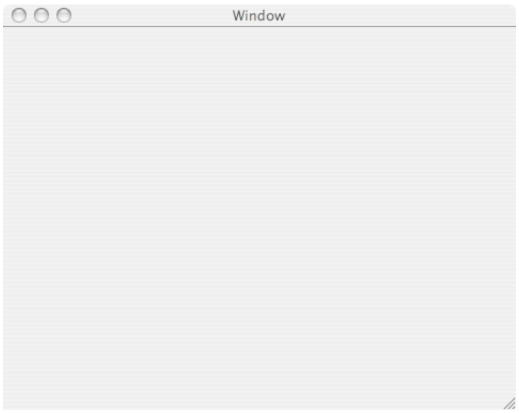
Nella finestra del progetto individuiamo allora la cartella denominata *Resources* e apriamola cliccando sul triangolino che sta lì di fianco. Facciamo due click su *MainMenu.nib*, e



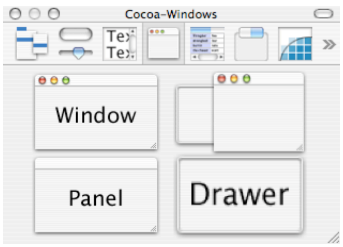
osserviamo che si apre *InterfaceBuilder*. L'integrazione tra XCode e InterfaceBuilder è totale, ed è una gran comodità che sia così. Se XCode serve per scrivere il codice vero e proprio di cui è costituita la nostra applicazione, InterfaceBuilder serve per realizzarne l'aspetto grafico, l'*interfaccia utente*. In InterfaceBuilder vi ritroverete con quattro finestre aperte. La più importante è quella che si chiama proprio *MainMenu.nib*, e che vedete qui di fianco. Essa contiene, organizzandole in pannelli, tutte le risorse della vostra applicazione (immagini, suoni, finestre, menu ecc.), ma anche qualche cosa in più, ovvero le *classi di controllo*. Che saranno mai? Momento: è ancora troppo presto per parlarne, ma non dovrete pazientare ancora per molto. Dalla finestra *MainMenu.nib* potete accedere, con un doppio click, a due delle altre finestre di InterfaceBuilder aperte sullo schermo, ovvero a quella contenente la barra dei menu di *Esempio1* e a quella contenente la finestra principale dell'applicazione.

Cominciate a riconoscere qualche cosa di familiare? La barra dei menu è quella che avevate visto quando abbiamo eseguito *Esempio1* lanciandolo da XCode, mentre la finestra è ovviamente quella che si era aperta automaticamente durante l'esecuzione di *Esempio1*. Bene, abbiamo trovato le risorse dell'applicazione. Vediamo brevemente la finestra restante: essa organizza, in pannelli, i vari elementi di interfaccia che potete creare ed aggiungere alla vostra applicazione. Fatevi un giro per i vari pannelli così da rendervi conto di che cosa vi offre InterfaceBuilder. Come vedete trovate di tutto: finestre, pulsanti, campi di testo, immagini, slider, pannelli, tabelle, menu e altre cose ancora.

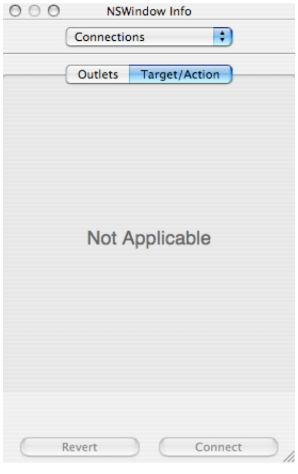




Poi, siccome ci verrà comodo averla a portata di mano in futuro, apriamo ancora un'ulteriore finestra, selezionando il comando *Show Info* dal menu *Tools*. La finestra che compare, praticamente vuota per ora, conterrà preziosissime informazioni sullo stato e sul funzionamento dei vari elementi di interfaccia che faranno parte della nostra applicazione e su come essi interagiranno col codice che scriveremo.



Se avete buon occhio avrete notato che nella finestra di informazioni che abbiamo appena aperto c'è un menu a comparsa, in cui una delle voci è *Connections*. Se la selezionate, un riquadro con due pannelli vi viene mostrato. Guarda caso, i due pannelli si intitolano *Outlets* e *Target/Action*, qualche cosa di molto simile al titolo di questo paragrafo.



Bene, ora che abbiamo capito dove si trovano le nostre risorse e abbiamo la possibilità di modificarle, dobbiamo fare in modo che facciano qualche cosa di utile. Rimandiamo ad un capitolo futuro la trattazione dei menu, e iniziamo a preoccuparci di mettere i nostri famosi campi di testo e il nostro famoso pulsante nella finestra principale dell'applicazione. Ma prima di tutto, facciamo qualche cosa di fondamentale. Creiamo una *classe di controllo*. Ora è veramente giunto il momento di capire di che cosa si tratta. Siccome è un argomento delicato e di importanza fondamentale, gli dedichiamo nientemeno che il nostro primo box di appofondimento:

Classi di controllo

Quando, nella seconda puntata di questa affasciante trilogia, abbiamo creato i nostri primi programmi Objective-C, ci eravamo preoccupati di realizzare un certo numero di classi (definendole con la loro interfaccia e scrivendone il codice nel file di implementazione). Il file main.c, poi, conteneva la funzione `main()`, ovvero la prima funzione ad essere eseguita in un qualunque programma C. Avrete sicuramente notato che anche il nostro progetto *Esempio1* è dotato di un file *main.m*, che contiene essenzialmente questo codice:

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

Di che cosa si tratta? Beh, si tratta di un vero e proprio miracolo fatto da Cocoa. Quando l'applicazione parte, la funzione `main()` viene eseguita, ed essa, come *unica cosa*, esegue un'altra funzione, `NSApplicationMain()`. Essa: carica le risorse necessarie dell'applicazione, inizializza la barra dei menu, mostra i menu e la finestra principale, *inizializza le classi di controllo* e avvia il *loop degli eventi*. Quando l'utente sceglie *Quit* dal menu principale dell'applicazione, `NSApplicationMain()` esce, e il programma termina. Mamma mia!

Ci sono due termini misteriosi in tutto questo discorso: *loop degli eventi* e le solite, dannate *classi di controllo*. Ebbene, siccome questo box si intitola *classi di controllo*, ci occupiamo innanzitutto del loop degli eventi. Si tratta di un concetto semplice. La funzione `NSApplicationMain()` avvia una sorta di blocco `while()` che viene eseguito costantemente, incessantemente. Al suo interno, va a controllare le azioni dell'utente: è stato selezionato un comando da menu? È stato cliccato un pulsante? È stato premuto un tasto? E così via. In caso affermativo, `NSApplicationMain()` valuta quale azione l'utente abbia compiuto e permette l'esecuzione del codice opportuno, sia esso codice già implementato dal sistema operativo (ad esempio quello necessario a visualizzare il menu quando l'utente clicca sul suo titolo), sia esso codice implementato dall'utente (ad esempio quello necessario ad eseguire un comando specifico implementato dalla nostra applicazione). Come tutto ciò avvenga non è importante, in quanto avviene automaticamente. La cosa importante è che Cocoa fa tutto questo per noi. Non dobbiamo preoccuparci di controllare che cosa faccia l'utente col mouse e con la tastiera; Cocoa, tramite questa miracolosa `NSApplicationMain()`, ci informerà in caso di necessità, lasciandoci felicemente ignari di tutto ciò che non riguarda direttamente la nostra applicazione.

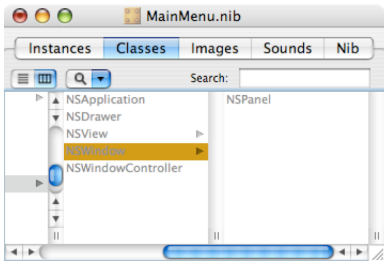
Come possa Cocoa avvisarci che l'utente ha compiuto una qualche operazione è una cosa che, finalmente, ha a che fare con le *classi di controllo*. Ogni applicazione deve averne almeno una. Siccome ci piace fare le cose semplici, le nostre applicazioni ne avranno esattamente una (con un'unica eccezione). Una classe di controllo è una classe Objective-C di cui viene allocato ed inizializzato un oggetto (e uno solo) automaticamente all'avvio dell'applicazione. L'oggetto appartenente a questa classe di controllo ha tipicamente il compito di *controllare* l'interfaccia utente; in esso, quindi, saranno contenuti i metodi che Cocoa chiamerà quando l'utente cliccherà sui pulsanti e selezionerà i comandi da menu implementati dalla nostra applicazione.

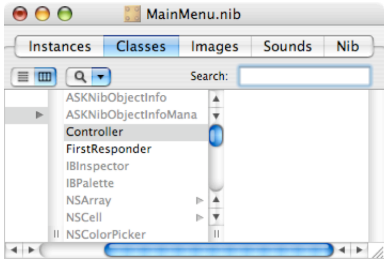
È importante che l'oggetto appartenente alla classe di controllo sia solo uno (in pratica non dovrete mai crearne uno nuovo dall'interno del vostro programma), perché una sola è l'interfaccia utente dell'applicazione. Ed è importante realizzare che l'oggetto appartenente alla classe di controllo (l'*oggetto di controllo* potremmo chiamarlo d'ora in poi) non è necessariamente l'oggetto che "contiene i dati" della vostra applicazione. Mi spiego: l'oggetto di controllo ha il compito di coordinare il funzionamento dell'applicazione, rispondendo ai comandi impartiti dall'utente tramite tastiera e mouse.

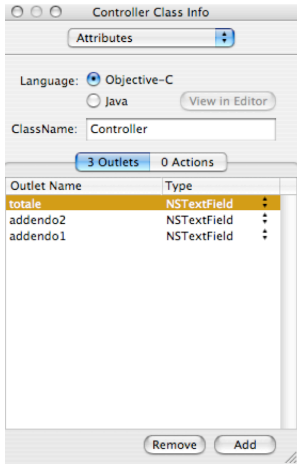
L'effetto di questi comandi sui dati che l'applicazione manipolerà può essere implementato direttamente dall'oggetto di controllo (per le applicazioni più semplici), o più tipicamente da classi Objective-C addizionali, i cui oggetti saranno allocati e inizializzati programmaticamente dall'interno dell'oggetto di controllo.

Sperando di avervi confuso le idee a sufficienza, direi di passare al pratico, creando questa benedetta classe di controllo e cercando così di capire che cosa fa e come funziona.

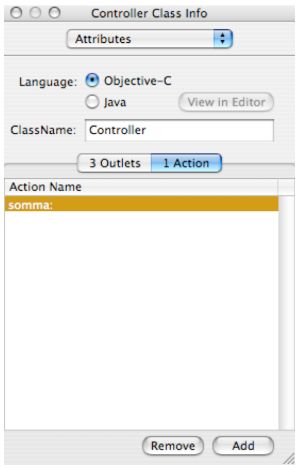
Dunque, torniamo ad InterfaceBuilder, alla finestra denominata *MainMenu.nib*. Cliccate sul pannello *Classes*, e dal menu a comparsa con la lente di ingrandimento selezionate *NSObject*. Questo menu vi consente di trovare una classe a vostra scelta all'interno dell'albero delle classi. *Albero delle classi?* Ricordate quando abbiamo detto che, scrivendo programmi in Objective-C su MacOS X, fa molto comodo che ogni classe sia sottoclasse di `NSObject`? Ecco: questo vuol dire che c'è un rapporto gerarchico tra le classi: a monte di tutte c'è `NSObject`, poi le sue sottoclassi, e le sottoclassi delle sottoclassi, e così via. Quando volete creare una nuova classe di controllo, un metodo comodo è dire ad InterfaceBuilder di trovare la classe `NSObject` col metodo ricordato sopra. Dal menu *Classes* selezionate quindi il comando *Subclass NSObject*, e date un nome alla nuova classe che InterfaceBuilder ha creato. Dal momento che stiamo creando una classe di controllo e siamo dotati di grande fantasia, chiamiamo questa classe `Controller`. Noterete come questa nuova classe compaia ora nella finestra denominata *MainMenu.nib*. Non tutte le classi che deciderete di creare ed usare nel vostro programma andranno create in questa maniera e dovranno essere elencate all'interno di *MainMenu.nib*, solo le classi di controllo. Nel corso di questo manuale torneremo ancora su questo argomento, cercando di renderlo chiaro. Ora dobbiamo rendere questa classe in grado di comunicare con l'interfaccia grafica. Cliccate sulla finestra di informazioni e, dal menu a comparsa, selezionate la voce *Attributes*. Da qui potete assegnare alla classe `Controller` degli *outlet* e delle *azioni*. Che cosa sono? Gli *outlet* non sono altro che dei puntatori a degli oggetti, solo che anziché puntare ad oggetti generici che create cammin facendo all'interno del vostro programma, puntano ad oggetti che fanno parte dell'interfaccia grafica del programma. Le *azioni* invece sono dei metodi implementati dalla classe che verranno chiamati automaticamente dal sistema operativo quando l'utente agisce su un elemento dell'interfaccia grafica (ad esempio con un click del mouse). Per rendere più chiara questa cosa, iniziamo a creare tre outlet cliccando sul pulsante *Add* e impostandoli come indicato nella figura qui di fianco. Per assegnare un tipo (*Type*) all'outlet, cliccate sulle due freccioline e trovate la classe che volete utilizzare. Il nostro programma, dicevamo, avrà bisogno di tre campi di testo: due editabili per inserire i due addendi, e uno non editabile che conterrà la somma dei due addendi. Ecco allora che i tre outlet che abbiamo creato hanno un nome che ci ricorda il loro scopo e appartengono tutti e tre alla classe `NSTextField`, la classe messaci a disposizione da Cocoa per i campi di testo. Come facciamo a sapere che è proprio `NSTextField` la classe giusta? Beh, lo scopriamo leggendo la documentazione di Cocoa, ad esempio, ma anche in un altro interessante modo che vedremo tra poco.



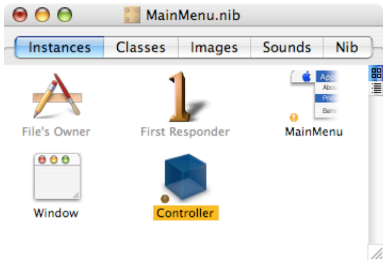


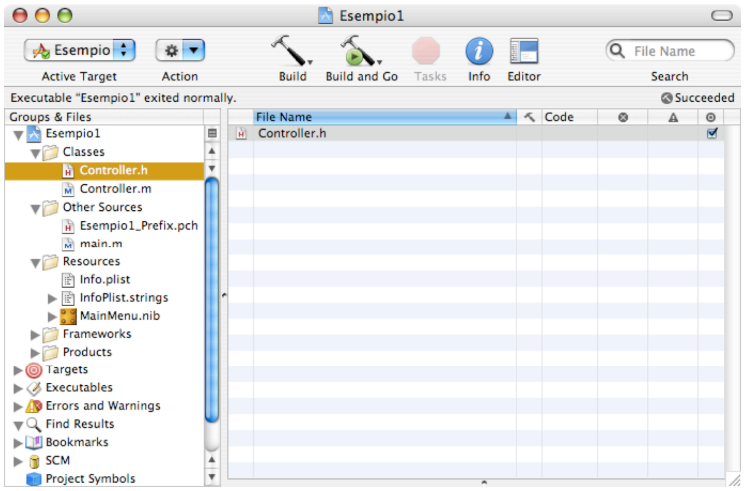


Facciamo quindi click sul pannello *Actions* e aggiungiamo un'azione, sempre cliccando sul pulsante *Add*, così da configurarla come indicato nella figura qua sotto. Notate che i due punti al termine del nome dell'azione sono indispensabili, in quanto i metodi che il sistema



operativo invoca in risposta ad un'azione da parte dell'utente hanno *sempre* un argomento (vedremo poi quale).
Aver creato questi outlet e queste azioni è importante, ma ancora non basta. Infatti dovremo collegarli in qualche maniera all'interfaccia grafica (che ancora non abbiamo creato) e, prima o poi, dovremo comunque mettere mano al codice. Ma facciamo un pezzo per volta. Cliccate su *Controller* nella finestra *MainMenu.nib* e dal menu *Classes* selezionate *Create Files for Controller*: questo comando genera automaticamente il file di interfaccia e il file di implementazione per la classe `Controller`, con gli outlet e le azioni che abbiamo appena impostato. Confermate le scelte di default nel pannello che vi si aprirà semplicemente cliccando su *Choose*. Sempre dal menu *Classes*, cliccate su *Instantiate Controller* e osservate che un'icona corrispondente alla classe `Controller` compare nella finestra *MainMenu.nib*. Vedete quel punto esclamativo in basso a sinistra di `Controller`? Indica che la classe è dotata di almeno un outlet che non risulta essere connesso a nessun elemento di interfaccia. Questo non ci deve preoccupare, perché ovviamente noi non abbiamo ancora creato l'interfaccia. Comunque, prima di ricompilare il nostro programma, dovremo accertarci che tutti gli outlet di `Controller` siano collegati, e quindi che sparisca il punto esclamativo. Salvate le modifiche con mela-S, quindi ritornate ad XCode. Notate che i file *Controller.h* e *Controller.m* sono stati aggiunti al progetto, nella cartellina *Other Sources*. Spostateli nella cartellina *Classes*, che è dove terremo tutti i file di codice e di interfaccia che creiamo noi per il nostro programma. È bene ricordare che questa organizzazione in cartelline è del tutto arbitraria,





è solo una questione di comodità. Potete lasciare i due file lì dove sono o spostarli altrove all'interno della finestra del progetto, non ha nessuna importanza. Ma quando il vostro programma crescerà di dimensioni, tenere i file ordinati è di importanza fondamentale per non perdere pezzi per strada.
Di già che ci siamo diamo un'occhiata a *Controller.h* e a *Controller.m*, facendo due click sui loro nomi nella finestra del progetto. Ecco qui che cosa ci viene mostrato per il file di interfaccia:

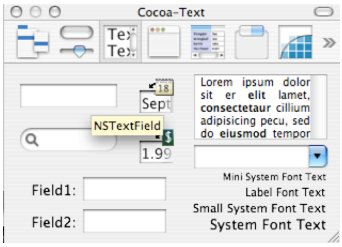
```
/* Controller */

#import <Cocoa/Cocoa.h>

@interface Controller : NSObject
{
    IBOutlet NSTextField *addendo1;
    IBOutlet NSTextField *addendo2;
    IBOutlet NSTextField *totale;
```

```
}
- (IBAction)somma:(id)sender;
@end
```

E qui per quello di implementazione:

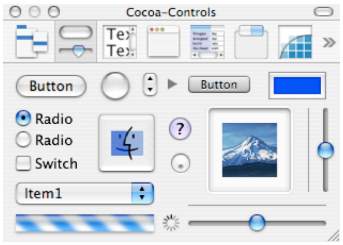
```
#import "Controller.h"

@implementation Controller

- (IBAction)somma:(id)sender
{
}

@end
```

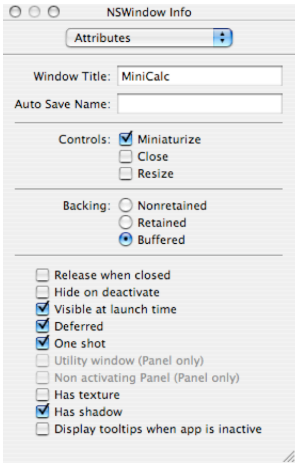
Non dovrebbe esserci nulla di sorprendente, se non la presenza, nel file di interfaccia, della parolina `IBOutlet` prima del nome della classe a cui appartengono i vari oggetti che avete dichiarato (`addendo1`, `addendo2` e `totale`). Questo è il modo che Objective-C usa per indicare che gli oggetti in questione fanno parte dell'interfaccia grafica dell'applicazione. Il metodo `somma:` ha un argomento, come dicevamo, di tipo `id` (quindi un oggetto generico) e chiamato genericamente `sender`. Esso è un puntatore che, di volta in volta, punterà all'elemento dell'interfaccia grafica che ha causato l'esecuzione del metodo. Vedremo tra un po' a che cosa ci possa servire quest'informazione. Objective-C identifica il metodo `somma:` come collegato ad un'azione mediante il suo tipo `IBAction`. È da notare che malgrado apparentemente il metodo `somma:` debba restituire qualche cosa di tipo `IBAction`, in realtà nessuna istruzione `return` è necessaria. Si tratta di un costrutto particolare, usato da Cocoa e Objective-C per tenere traccia del link del metodo in questione con l'interfaccia grafica, tutto qui. Notate comunque che il metodo `somma:` è vuoto: questo è logico, dal momento che il sistema non sa che cosa vogliamo fargli fare. Riempirlo con il codice di nostro gradimento sarà il nostro compito principale.

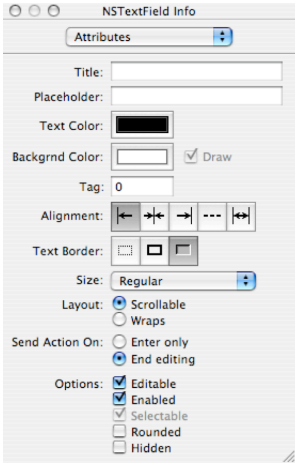


Siamo quasi a posto con i preparativi. Dobbiamo solo più creare l'interfaccia grafica e collegarla agli outlet e alle azioni della classe `Controller`. Bene, facciamolo! Attivate InterfaceBuilder e cliccate sulla finestra che contiene i vari elementi dell'interfaccia utente, e passate il puntatore del mouse sui vari oggetti. Un tooltip vi indica a quale classe appartengono. Ad esempio, il campo di testo editabile che si trova in alto a sinistra nella finestrella è di calsse `NSTextField`. Ah, ma non era la classe che avevamo assegnato agli outlet quando abbiamo creato la classe `Controller`? Infatti è proprio così. E se andate a guardare bene, è la stessa classe del testo non editabile identificato, nella finestrella, dalla scritta *System Font Text*. Allora useremo due caselle editabili di testo per inserire i due addendi e una casella non editabile di testo per inserire il totale. Siccome è una cosa carina spiegare all'utente a che cosa servono i vari campi di testo, creeremo altre tre caselle di testo non editabile che contengano alcune istruzioni. Per inserire gli elementi dell'interfaccia utente sulla finestra dell'applicazione, non dovete fare altro che trascinare i vari elementi di interfaccia nella finestra. Delle guide vi mostreranno le posizioni ottimali per collocarli così da ottenere la miglior resa estetica. Cercate di ottenere qualche cosa di simile a quanto indicato nella figura qui di fianco. I due campi di testo editabili sono ottenuti trascinando il rispettivo oggetto dalla finestrella con i vari elementi di interfaccia. I campi non editabili sono ottenuti trascinando l'elemento identificato dalla scritta *System Font Text*. Il pulsante è ottenuto trascinando l'oggetto denominato*Button* nell'apposito pannello della finestrella degli elementi di interfaccia, come mostrato nella pagina seguente.

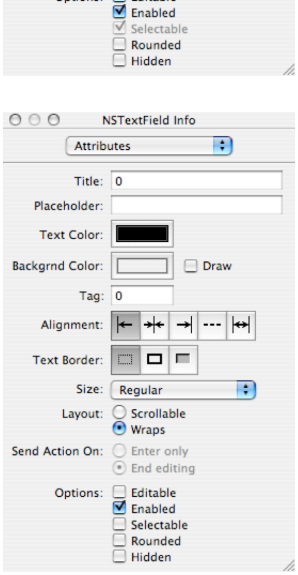
Il testo associato ad ogni elemento di interfaccia può essere modificato con un semplice doppio-click sull'elemento stesso, o dalla finestra di informazioni. Possiamo ad esempio modificare le proprietà della finestra facendo click sul suo sfondo e poi andando a modificare le sue caratteristiche come indicato nella figura. In questa maniera possiamo darle un titlo (*MiniCalc*), decidere che non è dotata di pulsanti di chiusura e ridimensionamento e stabilire che essa viene aperta e visualizzata all'avvio dell'applicazione senza alcun intervento da parte dell'utente. Similmente possiamo andare a controllare le proprietà di un campo di testo editabile e di uno non e ditabile cliccando su di essi e poi osservando il contenuto della finestra di informazioni. Potete osservare come essi differiscano per alcuni dettagli, il più importante dei quali è che quello editabile ha il simbolo di spunta sulla casella *Editable*, l'altro no. Il fatto che vi sia la spunta su *Enabled* consente comunque di associare delle azioni anche a questi campi di testo non editabile, nel caso vi venga in mente una loro qualche utilità. Per questo motivo potreste decidere di togliere la spunta alla casella *Enabled* per i campi di testo *Addendo 1*, *Addendo 2* e *Totale*.

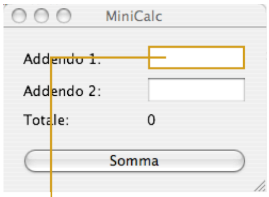


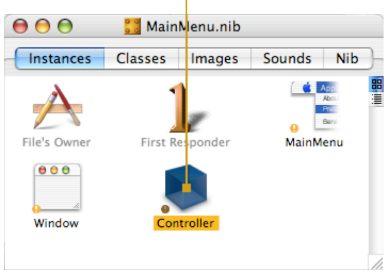


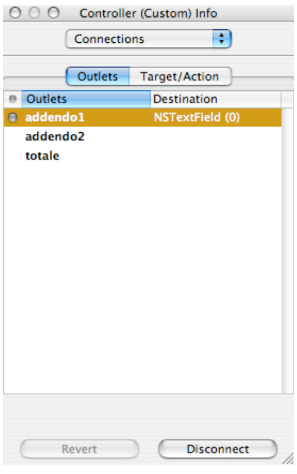


Ora non resta che fare l'ultimo passo, ovvero collegare gli elementi dell'interfaccia grafica con gli outlet e le azioni della classe `Controller`. Per fare questo, tenete premuto il tasto *control* sulla tastiera del Mac e cliccate sull'icona associata alla classe `Controller`. Senza rilasciare né il tasto *control* né il pulsante del mouse, trascinate fino alla finestra dell'applicazione, così da selezionare il campo di testo editabile in cui andrà scritto il primo addendo, come mostrato nella figura qui sotto. A questo punto la finestra di informazioni cambia di aspetto, mostrando l'elenco di tutti gli outlet dispnibili presso la classe `Controller`. Selezionate quello pertinente (se avete seguito l'esempio riportato in figura sarà quello denominato `addendo1`), e cliccate sul pulsante *Connect*. Un pallino identificherà l'outlet come già connesso con un elemento di interfaccia. Tutti gli outlet associati ad una classe compatibile con quella dell'elemento di interfaccia selezionato compariranno in grassetto. Dal momento che abbiamo selezionato un campo di testo (classe `NSTextField`) e tutti gli outlet di `Controller` sono di classe `NSTextField`, tutti e tre saranno mostrati in grassetto. Non vi è permesso associare ad un outlet un oggetto appartenente ad una classe non compatibile. Ripetete l'operazione anche per gli altri

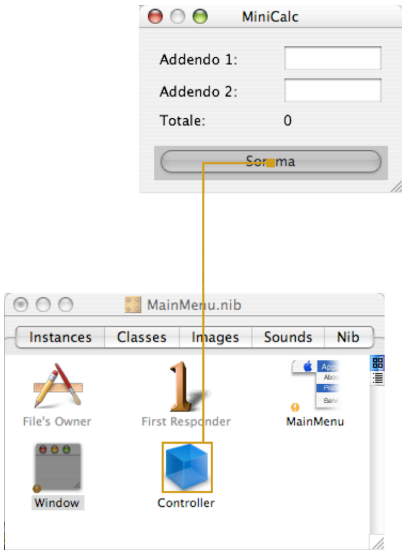








due outlet, così che risultino tutti e tre opportunamente connessi. Osservate che il punto esclamativo sull'icona della classe `Controller` sparisce.



Ora eseguiamo l'operazione speculare per collegare l'azione: tenendo premuto il tasto *control*, cliccate sul pulsante *Somma* nella finestra dell'applicazione e trascinate fino all'icona della classe `Controller` nella finestra *MainMenu.nib*. Nella finestra di informazioni, selezionate l'azione pertinente (in questo caso c'è solo l'azione `somma:`, quindi non potete selezionare altro), quindi cliccate su *Connect*. Salvate il tutto con mela-S.

Congratulazioni! Avete appena finito di collegare la vostra prima interfaccia grafica con la classe di controllo. Quello che abbiamo imparato in questo paragrafo è generale. Riassumiamolo: creiamo un nuovo progetto con XCode, e come prima cosa iniziamo ad editarne il file di risorse*MainMenu.nib* creando dall'interno di InterfaceBuilder una classe di controllo dotata degli outlet e delle azioni necessari per usare una versione minimale dell'interfaccia utente, quella che pensiamo di implementare inizialmente (potremo aggiungere outlet ed azioni in un secondo tempo, come vedremo). Creata la classe di controllo, disegnamo l'interfaccia utente in InterfaceBuilder e colleghiamo i vari elementi di essa con gli outlet e le azioni.

A questo punto, e per tornare al nostro esempio, il nostro programma sa a quale campo di testo si deve riferire quando usiamo la variabile `addendo1`, oppure `addendo2`, oppure `totale`, e il sistema operativo sa quale metodo invocare (`somma:`) quando l'utente clicca sul pulsante *Somma* presente nella finestra. Non resta che implementare il metodo `somma:` e il gioco è fatto.

Controllare i controlli

Prima di scrivere il codice relativo al metodo `somma:`, facciamo una piccola miglioria al nostro file di interfaccia. Ci servirà per definire un *flag*, una costante che assumerà il valore vero o falso, a seconda che vogliamo che la nostra applicazione sia "chiacchierona" oppure no. Siccome questa cosa strana può tornare comoda per fare *debug* (correzione degli errori), chiamiamo tale costante `kDebug`, e a scopo didattico la impostiamo a `YES` (valore logico vero). Va da sé che una volta che siete sicuri che il codice non abbia più bisogno di debug, potete tranquillamente riportarla a `NO` e non pensarci più. Allora, dicevamo, questo è il file di interfaccia:

```
/* Controller */

#import <Cocoa/Cocoa.h>

#define   kDebug            YES

@interface Controller : NSObject
{
    IBOutlet NSTextField *addendo1;
    IBOutlet NSTextField *addendo2;
    IBOutlet NSTextField *totale;
}
- (IBAction)somma:(id)sender;
@end
```

Come vedete l'unica aggiunta è la definizione della costante. Questo invece è il file di implementazione:

```
#import "Controller.h"

@implementation Controller

- (IBAction)somma:(id)sender
{
     if(kDebug)
          NSLog(@"L'utente ha cliccato sul pulsante Somma");
     [totale setDoubleValue:([addendo1 doubleValue]
                                                  +[addendo2 doubleValue])];
}

@end
```

Esaminiamolo nel dettaglio. Innanzitutto facciamo subito uso della costante `kDebug`. Se l'abbiamo impostata a `YES`, come abbiamo fatto or ora, la condizione dell'istruzione `if` è vera, quindi viene eseguita l'istruzione successiva. Essa è un'istruzione `NSLog()`, ovvero un'istruzione particolare che scrive un messaggio sulla finestra di log. A che ci serve? Beh, a noi in questo momento per imparare delle cose nuove, ma in generale serve per tenere sotto controllo il flusso di esecuzione del programma. Se, nei punti critici, e naturalmente a patto che la costante `kDebug` valga `YES`, inserite delle istruzioni `NSLog()` con opportuni messaggi, l'applicazione vi "racconterà" quello che sta facendo tramite la finestra di log. La stringa che passate ad argomento di `NSLog()` è una stringa Cocoa, quindi delimitata dalle virgolette e preceduta dal segno *at* `@""`. Al suo interno potete usare la stessa sintassi che usavate con la funzione `printf()` in C nel caso vogliate aggiungere degli argomenti a `NSLog()`, sotto forma di numeri interi, numeri decimali, ecc. Siccome la cosa è delicata, dedichiamo a questo argomento un breve box di approfondimento:

Uso di `NSLog()`

Diamo qui un breve ragguaglio della sintassi che si può utilizzare con la funzione `NSLog()`.

L'uso più semplice è quello con un solo argomento:

```
NSLog(@"Messaggio");
```

La funzione scriverà nella finestra di log la stringa `Messaggio`.

Potete decidere di includere nel messaggio anche il valore di alcune variabili. Ecco qui un caso abbastanza completo:

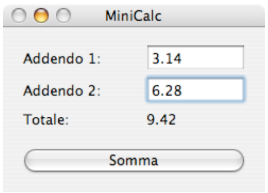
```
NSLog(@"Il cane %@ pesa %g chilogrammi
                                   ed ha %d anni",nomecane,peso,anni);
```

`nomecane` è un oggetto di classe `NSString` (o di una sua sottoclasse), `peso` è una variabile di tipo `double` e `anni` è una variabile di tipo `int`. La sintassi, quindi, è del tutto analoga a quella della funzione `printf()`, con l'aggiunta del costrutto `%@` per consentire l'output di "stringhe" Cocoa (quelle di classe `NSString`).

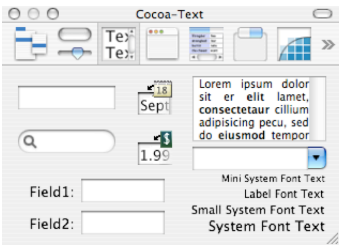
L'istruzione successiva è di fatto l'*unica* riga di codice (scritta da noi) di cui si compone il nostro programma. Essa sfrutta un'enorme comodità messaci a disposizione da Cocoa. Tutti gli oggetti appartenenti a sottoclassi di `NSControl` (e gli oggetti di classe `NSTextField` lo sono) rispondono ad una serie di metodi che consentono di leggerne ed impostarne il contenuto sotto diverse forme: stringhe, numeri interi, numeri a doppia precisione ecc. Qui facciamo esattamente questo: chiediamo agli oggetti (elementi di interfaccia) `addendo1` e `addendo2` di

comunicarci il loro contenuto sotto forma di un numero a doppia precisione, sommiamo tra di loro i due numeri così ottenuti, e comunichiamo a `totale` di impostare il suo contenuto ad un numero a doppia precisione pari appunto a questa somma. La cosa fantastica è che non abbiamo la più pallida idea di *come* i nostri tre campi di testo memorizzino al loro interno il loro contenuto. Noi ci limitiamo a dire loro in che *forma* vogliamo conoscere il loro contenuto, o sotto che *forma* vogliamo impostarlo; poi ci penseranno loro, automaticamente, a convertirlo nella *forma* opportuna quando ne faremo richiesta. Non è chiaro? Facciamo subito un esempio: l'oggetto `totale` è stato impostato con un metodo `setDoubleValue:` che ovviamente accetta un argomento di tipo `double`. In futuro potremmo senza indugio richiedere il contenuto di `totale` con una chiamata a `[totale intValue]` che ci restituirebbe il contenuto di `totale` *automaticamente convertito* in un valore di tipo `int`, oppure a `[totale stringValue]` che ci restituirebbe il contenuto di `totale` *automaticamente convertito* in un oggetto di classe `NSString`. Insomma, non sappiamo come `totale` (e `argomento1` e `argomento2`, naturalmente) memorizzino il loro contenuto, ma sappiamo che ci danno la possibilità di leggerlo e di scriverlo nella forma che ci viene più comoda in base alle circostanze. Le conversioni di tipo le fanno automaticamente loro. Questo è un comportamento di tutti gli oggetti appartenenti a sottoclassi di `NSControl`, quindi anche pulsanti, slider e altro ancora. Naturalmente il *contenuto* di tali oggetti varierà da caso a caso, ma il meccanismo è lo stesso.

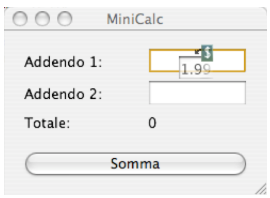
Ora non dimentichiamoci di compilare ed eseguire la nostra applicazione con una bella pressione della combinazione di tasti mela-R dall'interno di XCode. Scrivete qualche numero nei due campi di testo degli addendi, e cliccate su *Somma*. Verificate che cosa succede se inserite numeri negativi, dei numeri decimali o se non inserite affatto dei numeri. Verificate che ogni volta che cliccate su *Somma* la finestra di log riporta il messaggio (completo di data ed ora) che avete scritto ad argomento di `NSLog()`. Quando vi siete stufati di giocare con la vostra prima applicazione Objective-C/Cocoa, scegliete *Quit* dal menu denominato *Esempio1* (o fate mela-Q) e divertitevi a modificare la costante `kDebug` in modo che valga `NO` e verificate che ora la vostra applicazione (ricompilatela e rilanciatela con mela-R) funziona esattamente come prima, ma naturalmente non è più ciarliera quanto prima.



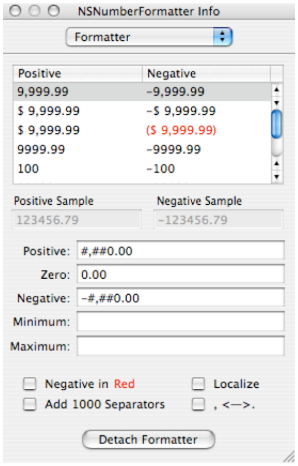
Se avete fatto qualche esperimento con l'applicazione, inserendo lettere al posto di numeri nei campi di testo, vi sarete accorti di un problema: se l'utente non inserisce un numero valido, il testo inserito viene trattato come se fosse il numero zero. Questo è logico, ma non sarebbe meglio se l'applicazione segnalasse all'utente il problema, o addirittura impedisse l'immissione di caratteri non numerici? Implementare una funzione di controllo dell'input dell'utente non è banale, perché a parte l'ovvia esclusione delle lettere e dei simboli strani dall'input, bisogna poi verificare che 25.4 è effettivamente un numero valido, mentre 3.45.11 non lo è, pur contenendo anche lui solo numeri e punti. Insomma, è una cosa fattibile ma è abbastanza una barba. Non è che per caso Cocoa ci può dare una mano? La risposta, naturalmente, è sì, e si chiama *formattatori* (oggetti di classe `NSFormatter`). Trattandosi di oggetti, è chiaro che il nostro programma li può creare e configurare come meglio crede, ma Cocoa ce ne offre due già precotti, configurabili con pochi click del mouse dall'interno di InterfaceBuilder. Uno di essi ha il compito di verificare che l'input dell'utente abbia le sembianze di una data, l'altro ha il compito di verificare che l'input dell'utente abbia le sembianze di un numero intero o decimale; proprio quello che serve a noi.



Com'è fatto un formattatore? Potete vederlo nella finestrella qua di fianco. Vedete quelle due iconcine, una con un calendario e l'altra col simbolo del dollaro $, più o meno al centro della finestra? Ecco, quei due sono i formattatori di cui parlavamo prima. Siccome siamo interessati a quello relativo ai numeri (e non alle date), clicchiamo sul quello col simbolo del dollaro e trasciniamolo sul campo editabile di testo in cui va inserito il primo addendo. La finestra di informazioni ci dà la possibillità di specificare meglio il formato dei numeri che siamo pronti ad accettare in quel



campo di testo, richiedendo che siano entro certi limiti, che abbiano o no il separatore delle migliaia, che i numeri negativi siano preceduti da un segno meno o piuttosto inclusi tra parentesi o ancora scritti in rosso, specficando quante cifre dopo la virgola vadano utilizzate e via discorrendo. I simboli # (cancelletto o *diesis*) indicano cifre opzionali, che verranno mostrate solo se necessario. I simboli 0 indicano cifre (non necessariamente pari al numero zero) che verranno sempre scritte. Quindi #,##0.00 nel campo *Positive* indica che un numero positivo è costituito da una cifra intera (le unità) seguita dal punto decimale e da due cifre decimali; se necessario, ovviamente, le cifre delle decine, delle centinaia e delle migliaia (e così via) saranno aggiunte alla bisogna. Se volete trattare solo numeri interi, potreste ad esempio voler modificare il *formattatore* dei numeri positivi in #,##0 e aggiornare di conseguenza anche gli altri. Ripetete la medesima configurazione anche per il campo di testo del secondo addendo. Potete provare subito le modifiche apportate anche senza ricompilare l'applicazione. InterfaceBuilder infatti vi permette di fare delle prove direttamente sull'interfaccia utente. Dal menu *File* selezionate *Test Interface* e giocate con i vostri nuovi campi di testo. Che cosa succede se inserite delle lettere anziché dei numeri? Provate: non riuscirete a lasciare il campo di testo, il sistema vi avvertirà con un beep che il vostro input non è conforme a quanto richiesto dal formattatore, e vi impedirà ogni altra operazione. Una volta che l'input invece è un numero valido, potrete fare quello che volete. Naturalmente, fin tanto che testate solo l'interfaccia, il pulsante *Somma* non funziona, ma potete comunque controllare quasi in tempo reale se le impostazioni date ai formattatori sono di vostro gradimento. Potete lasciare la modalità di test dell'interfaccia digitando, al solito, mela-Q. Salvate quindi le modifiche apportate a *MainMenu.nib* col consueto mela-S.



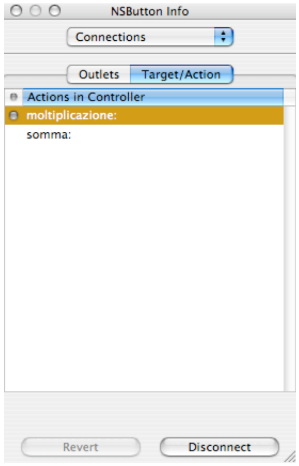
Ora potete tornare ad XCode e ricompilare l'applicazione, e divertirvi a notare che gli unici addendi accettati saranno dei numeri formattati correttamente così come avete richiesto voi. Per essere che non abbiamo scritto una sola riga di codice, non è male.

In questo capitolo abbiamo già imparato un sacco di cose nuove, ma non abbiamo ancora finito: ci sono ancora un paio di sorprese interessanti per voi, che sono di importanza fondamentale per destreggiarsi meglio con Cocoa e con gli strumenti di sviluppo messici a disposizione da Apple. Notiamo quindi che la nostra applicazione svolge senza difficoltà le addizioni (l'abbiamo fatta per quello!) e anche le sottrazioni, dal momento che basta far precedere il secondo addendo da un segno "meno". Quello che sicuramente non fa sono le moltiplicazioni. Non sarebbe forse il caso di fargliele fare? Il metodo forse più intuitivo, in base a quanto abbiamo imparato fino ad ora, per aggiungere questa funzionalità è quello di aggiungere un pulsante all'interfaccia grafica, e implementare una nuova azione collegata a questo pulsante che si prenda l'incarico di effettuare le moltiplicazioni. Per variare un po' rispetto a quanto abbiamo fatto in precedenza, modifichiamo la classe `Controller` direttamente da XCode anziché da InterfaceBuilder. Da XCode, quindi, aprite con un doppio click il file di interfaccia*Controller.h* e modificatelo come segue (le modifiche rispetto alla versione precedente sono riportate in grassetto):

```
/* Controller */

#import <Cocoa/Cocoa.h>

#define   kDebug             YES

@interface Controller : NSObject
{
    IBOutlet NSTextField *addendo1;
    IBOutlet NSTextField *addendo2;
    IBOutlet NSTextField *totale;
```

```
}
- (IBAction)somma:(id)sender;
-  (IBAction)moltiplicazione:(id)sender;
@end
```

Abbiamo aggiunto a mano (ovvero senza passare tramite la finestra di informazioni di InterfaceBuilder) un'azione al file di interfaccia della classe di controllo. Ovviamente, alla bisogna, possiamo aggiungere a mano anche degli outlet, mettendoli nel blocco racchiuso dalle parentesi graffe. Salvate le modifiche a *Controller.h* e passate a InterfaceBuilder. Nella finestra di *MainMenu.nib* cliccate sull'icona della classe `Controller` e poi sul pannello *Classes*. A questo punto le voci del menu *Classes* sono abilitate; selezionate il comando *Read Controller.h*: questo permette di importare in InterfaceBuilder il file di interfaccia della classe `Controller`, così che le modifiche che avete apportato a questo file in XCode si "propaghino" anche ad InterfaceBuilder (è l'azione simmetrica di quando, creando il nostro progetto, avevamo selezionato il comando *Create Files for Controller*, che "propagava" le modifche fatte alla classe `Controller` in InterfaceBuilder a XCode). Cliccate sul pannello *Instances* per tornare a vedere l'icona di `Controller` nella finestra *MainMenu.nib*. Modificate la finestra dell'interfaccia utente così che appaia come nella figura qui di fianco.



Tenendo quindi premuto il tasto *control* sulla vostra tastiera, cliccate sul pulsante *Moltiplica* e trascinate fino all'icona della classe `Controller` nella finestra *MainMenu.nib*. Dalla finestra di informazioni, collegate il pulsante all'azione `moltiplicazione:`. Salvate le modifiche e tornate a XCode. Ora non ci resta che implementare il metodo `moltiplicazione:`. Nel file di implementazione ancora non ce n'è traccia, naturalmente, perché questa volta non abbiamo incaricato InterfaceBuilder di creare per noi i file della classe `Controller` in base alle informazioni su outlet e azioni impostati dalla finestra di informazioni, ma abbiamo deciso di aggiornare la classe `Controller` a mano dall'interno di XCode. Naturalmente potete fare come preferite, ma io personalmente trovo più comodo creare i file con InterfaceBuilder all'inizio del progetto, quando creo la primissima versione dell'interfaccia grafica e quando i file della classe di controllo ancora non esistono. Poi, quando si presenta la necessità di modificare la classe di controllo per supportare nuovi elementi dell'interfaccia utente, preferisco agire tramite XCode e importare le modifiche in InterfaceBuilder, come abbiamo fatto ora. Non è l'unica strada possibile, ma siccome è quella con cui mi trovo meglio vi presento questa.

Da XCode, modificate quindi il file *Controller.m* così che appaia come segue (anche qui, le modifiche rispetto alla versione precedente sono riportate in grassetto):

```
#import "Controller.h"

@implementation Controller

- (IBAction)somma:(id)sender
{
     if(kDebug)
          NSLog(@"L'utente ha cliccato sul pulsante Somma");
     [totale setDoubleValue:([addendo1 doubleValue]
                                        +[addendo2 doubleValue])];
}
```

```
-  (IBAction)moltiplicazione:(id)sender
{
    if(kDebug)
        NSLog(@"L'utente ha cliccato sul pulsante Moltiplica");
    [totale setDoubleValue:([addendo1 doubleValue]
                                        *[addendo2 doubleValue])];
}

@end
```

Come dite? Non siete sorpresi da queste modifiche? Infatti, fate bene. Salvate, ricompilate ed eseguite (il solito mela-R) e divertitevi a fare qualche somma e qualche moltiplicazione. Poi tornate qui perché la parte divertente deve ancora venire.

Non sempre, infatti, creare un metodo per ogni operazione che l'utente vuole effettuare è una cosa furba. Inoltre, ancora non abbiamo fatto nessun uso dell'argomento `sender` che si trova in ogni azione. Forse è ora di rimediare a questa grave mancanza.

Che cos'è quest'argomento `sender`? Se conoscete un po' di inglese, saprete che *sender* vuol dire *mittente*. `sender` pertanto è un puntatore (infatti è di tipo `id`) all'oggetto che ha causato la chiamata all'azione. Nel nostro esempio, il metodo `somma:` è chiamato soltanto quando l'utente clicca sul pulsante *Somma*, per cui `sender` sarà sicuramente un puntatore ad un oggetto di classe `NSButton`, e al pulsante *Somma* in particolare. Similmente accade per l'argomento`sender` di `moltiplicazione:`, che sarà un puntatore all'oggetto di classe `NSButton` che rappresenta il pulsante denominato *Moltiplica*. Ma che cosa succede se vogliamo assegnare la stessa azione a più pulsanti? È possibile, ma ovviamente abbiamo bisogno di distinguere da quale oggetto sia partita l'azione, ovvero quale azione dell'utente abbia causato l'esecuzione del metodo. È l'argomento `sender` che ci dà quest'informazione: tramite esso possiamo discernere quale tra più oggetti abbia determinato l'esecuzione del metodo. Vediamo come.

Tornate ad XCode, e modificate il file *Controller.h* come segue:

```
/* Controller */

#import <Cocoa/Cocoa.h>

#define   kDebug            YES

@interface Controller : NSObject
{
    IBOutlet NSTextField *addendo1;
    IBOutlet NSTextField *addendo2;
    IBOutlet NSTextField *totale;
}
- (IBAction)calcola:(id)sender;
@end
```

Come vedete abbiamo eliminato le azioni precedenti, definendone una nuova: `calcola:`. Essa avrà il compito di gestire sia le addizioni, sia le moltiplicazioni. Fate quindi un giro in InterfaceBuilder, e con la tecnica già usata in precedenza importate il file *Controller.h*: rivediamo brevemente come. Selezionate l'icona di `Controller` nella finestra titolata *MainMenu.nib*, cliccate sul pannello *Classes*, poi dal menu *Classes* selezionate *Import Controller.h*. InterfaceBuilder vi avviserà che c'è un'inconsistenza tra quanto era precedentemente contenuto in *Controller.h* e quanto è contenuto adesso: infatti, non solo è stata aggiunta un'azione (`calcola:`), il che non avrebbe causato nessun allarme, ma ne sono state eliminate due (`somma:` e `moltiplicazione:`), il che ovviamente genera un po' di apprensione.

Siccome delle due azioni vecchie non ce ne facciamo più niente, diciamo ad InterfaceBuilder di rimpiazzare la vecchia versione di *Controller.h* cliccando sul pulsante *Replace*. Tornate quindi al pannello *Instances*. Ora, con la solita tecnica (tenete premuto *control* e trascinate dal pulsante all'icona di `Controller` nella finestra *MainMenu.nib*), collegate sia il pulsante *Somma* sia il pulsante *Moltiplica* all'azione `calcola:`. Noterete che nulla vi vieta questo. Quando avete fatto tutto, salvate le modifiche e tornate ad XCode.

Ora dobbiamo occuparci dell'implementazione del metodo `calcola:`. Possiamo liberarci dei vecchi metodi `somma:` e `moltiplicazione:` e riscrivere quanto segue:

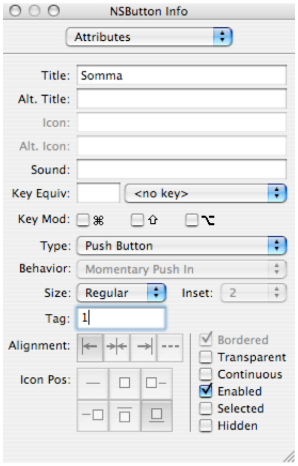
```
#import "Controller.h"

@implementation Controller

- (IBAction)calcola:(id)sender
{
     if(kDebug)
          NSLog(@"L'utente ha cliccato il pulsante %@",[sender title]);
     if([[sender title] caseInsensitiveCompare:@"Somma"]==NSOrderedSame)
          [totale setDoubleValue:([addendo1 doubleValue]
                                       +[addendo2 doubleValue])];
     else if([[sender title]
                    caseInsensitiveCompare:@"Moltiplica"]==NSOrderedSame)
          [totale setDoubleValue:([addendo1 doubleValue]
                                       *[addendo2 doubleValue])];
}

@end
```

Compilate ed eseguite, poi vediamo nel dettaglio che cosa abbiamo fatto. Iniziamo ad usare quanto abbiamo imparato nel box di approfondimento relativo alla funzione `NSLog()`. Se il flag di debug è attivo, il messaggio riporta il *titolo* del pulsante che ha chiamato l'azione `calcola:`. Infatti, tutti gli oggetti appartenenti a classi che sono sottoclassi di `NSControl` (e `NSButton`, classe a cui appartengono i pulsanti come *Somma* e *Moltiplica*, lo è), dispongono di un metodo `title` che restituisce un oggetto di classe `NSString`, nient'altro che il titolo dell'oggetto. Se per alcuni controlli il titolo può avere poco senso, per un pulsante è ovviamente il testo contenuto nel pulsante stesso. Ovviamente usiamo quest'informazione non solo per comunicare qualche cosa allo sviluppatore tramite la finestra di log (se il flag di debug è attivo), ma anche per decidere se l'operazione da eseguire sia una somma o una moltiplicazione. La chiamata a `[sender title]`, infatti, restituisce un oggetto di classe `NSString` del quale possiamo chiamare il metodo `caseInsensitiveCompare:` che accetta come argomento una stringa. Se l'oggetto restituito da `[sender title]` e la stringa messa ad argomento di `caseInsensitiveCompare:` sono uguali (a meno di variazioni nelle maiuscole), viene restituito un valore `NSOrderedSame` (definito da qualche parte da Cocoa); così facendo discerniamo tra una somma e una moltiplicazione.

Dove sta il difetto di questo approccio? Sta nel fatto che i nomi dei pulsanti possono cambiare, ad esempio se l'applicazione è localizzata in più di una lingua. Lo scopo del meccanismo di localizzazione messoci a disposizione da Cocoa (di cui ancora non abbiamo parlato) è quello di far sì che, per aggiungere una lingua all'applicazione, non sia necessario modificarne il codice. Qui, ovviamente, c'è qualche cosa che non va.

Per fortuna gli oggetti appartenenti a classi che sono sottoclassi di NSControl hanno a disposizione non solo il metodo `title`, ma anche il metodo `tag`, che restituisce un numero intero associato al controllo stesso. È ovviamente possibile decidere quale numero intero usare per oggetti diversi appartenenti all'interfaccia grafica. Ad esempio, tornando ad InterfaceBuilder, possiamo selezionare il pulsante *Somma* e dalla finestra di informazioni

impostare la casellina *tag* al valore 1 (dal pannello *Attributes*). Similmente possiamo fare per il *tag* del pulsante *Moltiplica*, che potremmo impostare a 2. Salvate le modifiche. Modifichiamo allora il file *Controller.h* (da XCode, naturalmente) in modo che diventi così:




```
/* Controller */

#import <Cocoa/Cocoa.h>

#define   kDebug              YES

#define   kTagSomma           1
#define   kTagMoltiplica      2

@interface Controller : NSObject
{
    IBOutlet NSTextField *addendo1;
    IBOutlet NSTextField *addendo2;
    IBOutlet NSTextField *totale;
}
- (IBAction)calcola:(id)sender;
@end
```

Salviamo il file, poi modifichiamo il file *Controller.m* come segue:

```
#import "Controller.h"

@implementation Controller

- (IBAction)calcola:(id)sender
{
     if(kDebug)
          NSLog(@"L'utente ha cliccato il pulsante %@",[sender title]);
     if([sender tag]==kTagSomma)
          [totale setDoubleValue:([addendo1 doubleValue]
                                       +[addendo2 doubleValue])];
     else if([sender tag]==kTagMoltiplica)
          [totale setDoubleValue:([addendo1 doubleValue]
                                       *[addendo2 doubleValue])];
}

@end
```

Salvate, ricompilate ed eseguite. Funziona esattamente come prima, ma ora è molto meglio: ogni pulsante, infatti, è identificato *per numero*, e il numero non dipende dalla lingua che l'utente ha scelto (tra quelle disponibili) per eseguire l'applicazione. Il numero non è nemmeno visibile dall'utente, è un riferimento interno del programma. Il programma *sa* (perché è il programmatore ad averlo deciso) che il pulsante responsabile di effettuare somme ha il *tag* pari ad 1, mentre quello responsabile di effettuare le moltiplicazioni ha il *tag* pari a 2. Tutto il resto non conta.

In questo capitolo abbiamo imparato a ricevere messaggi dai pulsanti e a leggere e scrivere il contenuto di campi di testo. Abbiamo inoltre imparato a che cosa servono le classi di controllo, a creare un'interfaccia utente, e varie tecniche per rapportarci con essa. Tutto quello

che abbiamo imparato qua costituisce buona parte di ciò che c'è da sapere su Cocoa. Certo, Cocoa offre decine di classi, ognuna con un sacco di metodi, ed è per questo che ci sono libri di molte centinaia di pagine per coprire tutti questi argomenti. Ma, concettualmente, abbiamo imparato buona parte di ciò che c'è da sapere. Nei prossimi capitoli impareremo il resto (per lo meno le cose più importanti), ed applicheremo quanto abbiamo imparato ad altri elementi di interfaccia utente e ad altre classi di Cocoa.

4. Finestre e menu

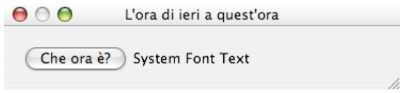
Menu e finestre

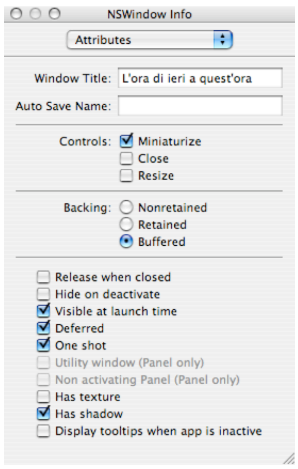
Non di soli pulsanti e campi di testo è fatta un'applicazione. Gli altri elementi di interfaccia più usati sono i menu e le finestre. E in questo capitolo proprio di essi ci occuperemo. Però dobbiamo fare una precisazione.

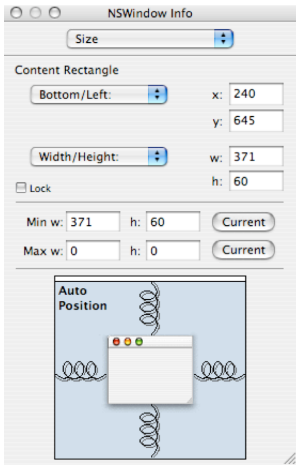
Ci sono sostanzialmente due ragioni per cui un'applicazione può voler usare più di una finestra. La prima è che l'applicazione presenta informazioni di natura diversa in finestre diverse (la finestra principale, come nell'Esempio1, consente di effettuare tutte le operazioni più importanti consentite dall'applicazione, ma poi questa avrà una finestra per le informazioni, una finestra per configurare chissà che cosa, una finestra per fare qualche operazione strana e poco comune). La seconda è che l'applicazione consente la creazione di molteplici *documenti* ognuno dei quali ha una struttura comune e contenuti diversi; è questo il caso di un elaboratore di testi, di un programma di grafica, di un browser per il web ecc. Cocoa offre due modi diversi per fronteggiare queste due esigenze (che naturalmente possono convivere). La prima esigenza viene trattata in questo capitolo, ovvero come creare finestre che si vadano ad aggiungere alla finestra principale dell'applicazione per dare all'utente informazioni aggiuntive o ulteriori possibilità di controllo. La seconda, ovvero la creazione di una *Cocoa document-based application* (avete presente quando selezionate *Cocoa application* dall'elenco delle possibilità che XCode vi offre quando create un nuovo progetto? Ecco, un'altra di queste possibilità è proprio una *Cocoa document-based application*), cioè di un'applicazione che supporta documenti multipli tutti con la stessa struttura, non sarà coperta da questo manualetto, e si rimanda alla bibliografia presentata nell'ultimo capitolo per una trattazione dell'argomento.

Detto ciò, chiediamoci che cosa dovrà mai fare il nostro programma di così sofisticato da richiedere più di una finestra, nonché l'interazione con i menu. Esso si propone di visualizzare nella sua finestra principale un'informazione misteriosa e preclusa ai più: l'ora. Poi, a scelta dell'utente mediante un apposito comando da menu, consentirà di visualizzare in un'*altra* finestra un'informazione addizionale strettamente legata all'ora, ovvero il *nome del computer*. Lascio a voi immaginare a che cosa possa servire questa formidabile applicazione e quali rivoluzionarie ricadute possa avere il suo rilascio nel mercato dell'informatica. Qui ci limiteremo a trattarla per i suoi aspetti prettamente didattici.

Chiudete, se non l'avete già fatto, il progetto relativo all'Esempio1 in XCode e le relative finestre in InterfaceBuilder. Poi, da XCode, create un nuovo progetto (*File > New Project…*) di tipo *Cocoa application*, e dategli il nome *Esempio2* (sempre senza spazi in mezzo). Dalla cartellina *Resources* nella finestra del progetto fate due click su *MainMenu.nib* così da aprire il file di risorse in InterfaceBuilder. Aggiungete alla finestra principale un pulsante e un campo di testo statico così da farla apparire come nella figura qui di fianco. Per fare ciò,

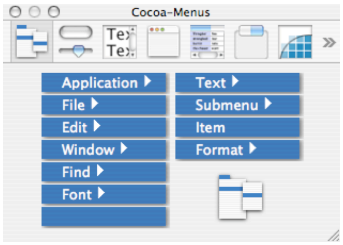




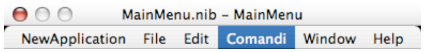


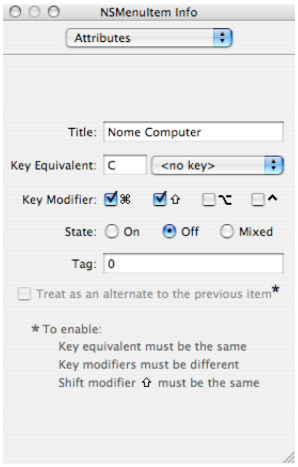
ricordatevi di aprire la finestrella di info e accertatevi di aver impostato i parametri mostrati in figura per la finestra principale. Potete rendere la finestra principale così piccola selezionando la voce *Size* dal menu a comparsa della finestra di informazioni e impostando la voce *Min h:* a 60, come mostrato in figura. Tra l'altro, incominciate a notare che questo pannello *Size* ha un aspetto molto accattivante: qualunque oggetto, dalla finestra principale ad ogni oggetto in essa contenuto (provate a fare click sul pulsante o sul campo statico di testo, ad

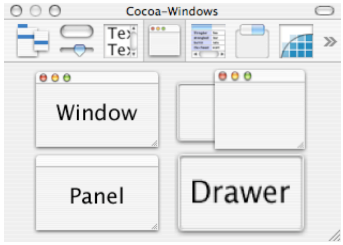
esempio) ha una posizione (sullo schermo per la finestra, all'interno della finestra per i suoi oggetti) che è impostabile di qui; inoltre, delle "molle" permettono di stabilire se l'oggetto sia "fluttuante" oppure in posizione fissa (provate a cliccare su una molla e si trasformerà in un'asta rigida). Qui ancora non ci occupiamo di questo, ma più avanti, quando inizieremo a supportare finestre ridimensionabili, aver già preso contatto con queste novità ci tornerà utile.



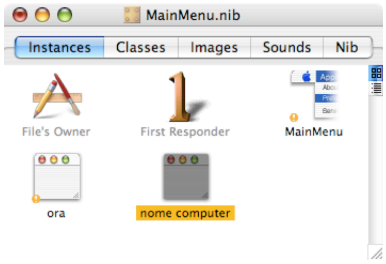
Trovate quindi la finestrella con tutti gli elementi di interfaccia e cliccate sul pannello che vi mette a disposizione i menu, come mostrato in figura. Come potete vedere vi vengono offerti un certo numero di menu precotti (*Application*, *File*, *Edit* e *Window* sono già presenti nella barra dei menu di default dell'applicazione), che potete aggiungere con un semplice drag&drop. Alcune delle voci non sono dei menu, ma sono degli *elementi* (*item* in inglese), che si possono aggiungere (sempre con drag&drop) ad un menu esistente. L'elemento vuoto è un separatore (quello spazio bianco che serve per separare le voci di un menu). Quello denominato *Submenu* serve per creare un elemento con un sottomenu all'interno di un menu, o anche per creare un nuovo menu sulla barra. Infine l'icona in basso a destra può essere trascinata sulla finestra di *MainMenu.nib* ad esempio per creare menu scollegato dalla barra o da un pulsante. Qui limitiamoci a cliccare su *Submenu* e a trascinarlo nella finestrella con la barra dei menu. Notate come, sia in fase di inserimento del menu che dopo, possiate riarrangiare l'ordine dei menu semplicemente trascinandoli sulla barra. Rinominatelo *Comandi* facendo un doppio click sul suo nome ed editate alla stessa maniera l'unica voce esistente, rinominandola *Nome Computer*. Dalla finestra delle informazioni, dal pannello *Attributes*, impostate la scorciatoia da tastiera per questa voce (mela-shift-C), come mostrato in figura.

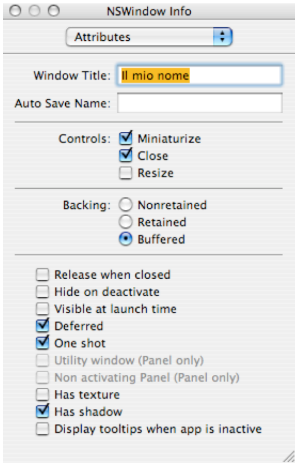




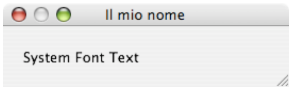


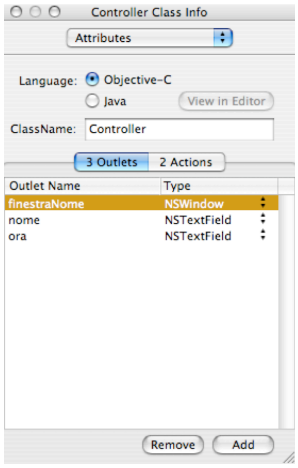
Ora creiamo la finestra che conterrà il nome del computer. Dalla finestrella di InterfaceBuilder che contiene i vari elementi di interfaccia, cliccate sul pannello che vi lascia scegliere tra varie tipologie di finestre. Trascinate quella identificata col nome *Window* nella finestra *MainMenu.nib*. Una nuova finestra è stata aggiunta alle risorse del nostro progetto. Nella finestra*MainMenu.nib* date un nome a queste finestre, giusto per identificarle con maggiore facilità; per dare un nome vi basta selezionarne l'icona, premere il tasto*return* o *a-capo* e quindi digitare il nome. Tale nome è solo un promemoria per voi, non comparirà da nessuna parte né nell'interfaccia utente né nel codice dell'applicazione. Io ho dato due nomi misteriosi a queste finestre, *ora* e *nome computer*, a ricordare a che cosa servono (si veda la figura). Accertatevi che la finestra delle informazioni sulla nuova finestra che abbiamo creato sia impostata come in figura (è importante che la casellina *Visible at launch time* non sia selezionata, così da garantire che la finestra col nome del computer non venga visualizzata automaticamente quando si lancia l'applicazione ma solo quando l'utente seleziona l'apposito comando da menu. Il contenuto di questa nuova finestra è facile, in quanto ci limitiamo ad aggiungere un campo di testo statico e a ridimensionare la finestra così che non contenga inutili spazi vuoti; potreste nuovamente essere costretti a raggiungere il pannello *Size*

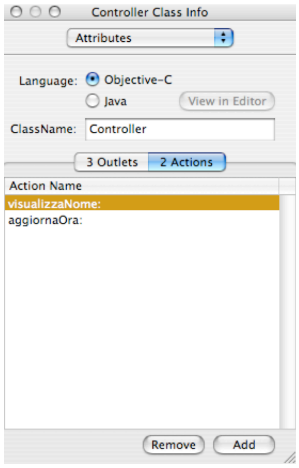




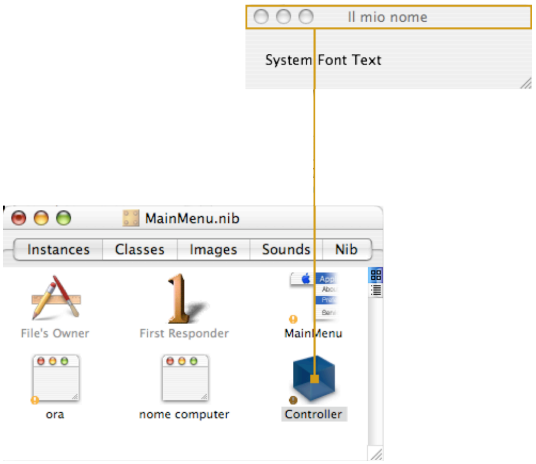
tramite il menu a comparsa della finestra di informazioni per ridurre le dimensioni minime consentite della finestra che conterrà il nome del computer. La finestra nella sua forma finale è mostrata qui in figura.



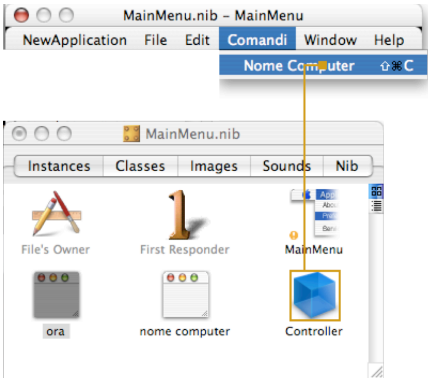




Ora dobbiamo creare la classe di controllo. La procedura è la solita. Dalla finestrella *MainMenu.nib* cliccate sul pannello *Classes*, rintracciate la classe`NSObject` e dal menu *Classes* selezionate*Subclass NSObject*. Chiamate `Controller` la classe che avete appena creato. Dalla finestra di informazioni aggiungete 3 outlet e 2 azioni, come mostrato nelle figure qui di fianco. Create i file per la classe `Controller` (*Classes* > *Create Files for Controller*) e istanziate la classe (*Classe* > *Instantiate Controller*). Con la solita tecnica (premete il tasto *control* sulla tastiera e poi cliccate e trascinate) collegate l'icona di *Contoller* nella finestra *MainMenu.nib* con il campo di testo nella finestra dell'ora (ovviamente stabilendo la connessione con l'outlet *ora*), con il campo di testo nella finestra del nome del computer (stabilendo la connessione con l'outlet *nome*) e con la *barra del titolo* della finestra preposta a mostrare il nome del computer, come mostrato in figura (stabilendo la connessione con l'outlet *finestraNome*). Per finire, collegate il pulsante *Che ora è?* con l'azione *aggiornaOra:* di *Controller* e il comando *Nome Computer* del menu *Comandi* con l'azione *visalizzaNome:*, come mostrato in figura. Salvate le modifiche e tornate ad XCode.



Individuate i file *Controller.h* e *Controller.m* che probabilmente si trovano nella cartellina denominata *Other Sources* all'interno della finestra del progetto *Esempio2*. Spostateli, se credete, nella cartellina *Sources*, così da fare un po' d'ordine. Fate quindi due click sull'icona di *Controller.h* e verificate che il file di interfaccia sia come qui riportato:




```
/* Controller */

#import <Cocoa/Cocoa.h>

@interface Controller : NSObject
{
    IBOutlet NSWindow *finestraNome;
    IBOutlet NSTextField *nome;
    IBOutlet NSTextField *ora;
}
- (IBAction)aggiornaOra:(id)sender;
- (IBAction)visualizzaNome:(id)sender;
@end
```

Come vedete è già tutto a posto, non dobbiamo fare niente. Aprite allora il file di implementazione *Controller.m* e modificatelo come segue:

```
#import "Controller.h"

@implementation Controller

- (void)awakeFromNib
{
    [self aggiornaOra:self];
}

- (IBAction)aggiornaOra:(id)sender
{
    [ora setStringValue:[[NSDate date] description]];
}

- (IBAction)visualizzaNome:(id)sender
{
    [nome setStringValue:[[NSHost currentHost] name]];
    [finestraNome makeKeyAndOrderFront:self];
}

@end
```

Salvatelo, poi compilate ed eseguite col solito comando mela-R e divertitevi a giocare un po' con l'applicazione. Fate un po' di esperimenti, provando a vedere che cosa succede se cliccate il pulsante *Che ora è?* , se selezionate il comando *Nome Computer* dal menu *Comandi*, anche quando la finestra col nome del computer è già visibile sullo schermo ma magari è nascosta dalla finestra di un'altra applicazione o è minimizzata nel Dock. Verificate che il menu *Window* riporta correttamente la presenza della finestra col nome del computer quando questa è visibile sullo schermo. Poi tornate qui a leggere, perché questo programma, seppur così corto, riserva alcune sorprese.

Iniziamo dal metodo `awakeFromNib`. Chi è costui? Da dove salta fuori? E che vuole da noi? Non è dichiarato nel file di interfaccia della classe `Controller`, e allora che ci fa qui? È un metodo che Cocoa ci mette a disposizione all'interno della classe di controllo e che viene chiamato automaticamente dal sistema operativo quando le risorse dell'applicazione sono state caricate in memoria (*nib* è l'estensione dei file contenenti le risorse dell'applicazione, come *MainMenu.nib*). È un metodo che talvolta può essere comodo implementare per effettuare alcune operazioni di inizializzazione dell'interfaccia utente. Ad esempio la finestra principale della nostra applicazione contiene un campo di testo per mostrare l'ora. Tuttavia, quando abbiamo creato l'interfaccia utente, non potevamo certo sapere che cosa scrivere in questo campo. Sarebbe molto brutto che l'utente, avviando il nostro splendido *Esempio2*, si ritrovasse con la finestra principale che, di fianco al pulsante *Che ora è?* mostrasse la scritta *System Font Text*. Allora implementiamo il metodo `awakeFromNib` (che non ha bisogno di essere dichiarato nell'interfaccia della classe) affinché, non appena le risorse della nostra applicazione (inclusa la finestra principale) sono state caricate in memoria dal file di risorse, venga aggiornata l'ora nell'apposito campo di testo.

Avete notato che l'ora è "statica", ovvero è impostata al momento in cui cliccate sul pulsante *Che ora è?* ma poi non viene aggiornata automaticamente? Questo perché ancora non sappiamo usare un *timer*, impareremo a farlo nel prossimo capitolo. Qui, lo scopo principale del gioco era far vedere come usare un menu (niente di terribile, come avete visto, basta collegarne da InterfaceBuilder le voci alle azioni da chiamare) e come visualizzare una finestra. Questa operazione viene fatta all'interno del metodo `visualizzaNome:` dove l'oggetto `finestraNome`,

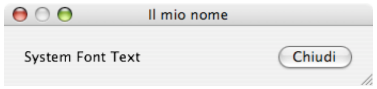
che è un outlet collegato alla finestra vera e propria, riceve la chiamata al metodo (implementato dalla classe `NSWindow`) `makeKeyAndOrderFront:`, il cui argomento di tipo `id` è il solito `sender`, ovvero `self` nel nostro caso (se preferite, potevamo mettere lo stesso `sender` che aveva chiamato `visualizzaNome:`, che poi era un oggetto di classe `NSMenuItem` essendo una voce del menu *Comandi*, oppure anche `nil`, tanto di questo argomento, qui, non ce ne facciamo nulla). `makeKeyAndOrderFront:` visualizza (se non era visibile a schermo) e porta in primo piano l'oggetto (appartenente alla classe `NSWindow`, quindi una finestra) che lo chiama.

Bene, abbiamo imparato ad aprire una finestra, ma come si fa a chiuderla? Certo, basta che l'utente clicchi sul pulsante di chiusura. Ma se noi l'avessimo disabilitato e volessimo che l'utente chiuda la finestra cliccando su un pulsante inserito all'interno della finestra? Come si fa? Questa domanda non è poi così stravagante, perché è così che sono fatti molti degli about-box delle applicazioni. Vediamo allora di modificare il nostro *Esempio2* così da poter chiudere la finestra col nome del computer non solo schiacciando il suo pulsantino di chiusura, ma anche cliccando su un apposito pulsante aggiunto all'uopo di fianco al nome stesso del computer. Non è un gran bell'esempio di design dell'interfaccia grafica (e sono quasi sicuro che sia una cosa deprecata dalle linee guida di Apple su come disegnare un'interfaccia utente), a qui è a fin di bene, a scopo didattico.

Torniamo al file di interfaccia *Controller.h* e modifichiamolo come segue (le variazioni rispetto alla versione precedente sono riportate in grassetto):

```
/* Controller */

#import <Cocoa/Cocoa.h>

@interface Controller : NSObject
{
    IBOutlet NSWindow *finestraNome;
    IBOutlet NSTextField *nome;
    IBOutlet NSTextField *ora;
}
- (IBAction)aggiornaOra:(id)sender;
- (IBAction)visualizzaNome:(id)sender;
-  (IBAction)chiudiFinestraNome:(id)sender;
@end
```

Salvate le modifiche e tornate ad InterfaceBuilder, con la finestra *MainMenu.nib* dell'*Esempio2* in primo piano. Selezionate l'icona *Controller*, poi il pannello *Classes* e infine, dal menu *Classes*, selezionate la voce *Read Controller.h*. Quindi tornate al pannello *Instances*. Rispetto a quanto avvenuto nell'*Esempio1* qui non vi viene chiesta nessuna conferma, perché non abbiamo cancellato nessun outlet e nessuna azione precedentemente definita, ci siamo limitati ad aggiungere roba (un'azione, in questo caso). Modificate la finestra che mostra il nome del computer aggiungendo un pulsante, come mostrato in figura. Quindi collegate il pulsante *Chiudi* all'azione *chiudiFinestraNome:* di *Controller*, con la solita tecnica di trascinare col mouse dal pulsante all'icona di *Controller* nella finestra *MainMenu.nib* tenendo premuto il tasto *control* sulla tastiera del Mac. Salvate le modifiche e tornate ad XCode.



Adesso, indovinate un po', dobbiamo implementare il metodo `chiudiFinestraNome:`. Sarà una cosa terribile. Armatevi di tanta pazienza e nel file di implementazione *Controller.m*, in fondo ma prima della direttiva `@end`, aggiungete quanto segue:

```
- (IBAction)chiudiFinestraNome:(id)sender
{
    [finestraNome orderOut:self];
```

```
}
```

Come vedete, per chiudere una finestra che avevate reso visibile con `makeKeyAndOrderFront:`, chiamate `orderOut:`, il cui argomento è di nuovo un bel `sender` di tipo `id` (stesso discorso fatto sopra, qui passiamo `self`).

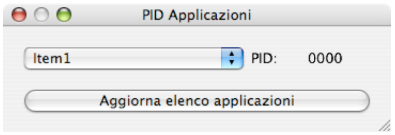
Sentite il tintinnio delle monete che incominciano ad arrivarvi a camionate, grazie alle vostre nuove rivoluzionarie applicazioni Cocoa? Non ancora? Per forza: è perché ancora non sapete come si usano i menu a comparsa.

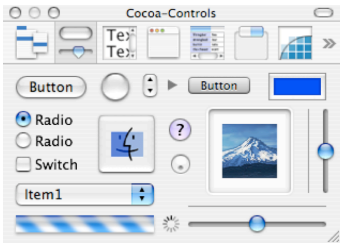
Menu a comparsa

Se vi state chiedendo che cosa sia un menu a comparsa, non siete i soli. Non sapevo come tradurre il termine inglese *Pop-up button* (che, tra l'altro, implica che un menu a comparsa è in effetti un pulsante), comunque è uno di quei menu che non sono attaccati alla barra dei menu, ma che compaiono sotto forma di pulsanti nelle finestre. Comunque, pulsanti o menu che siano, li chiameremo *menu a comparsa* e li tratteremo qui.

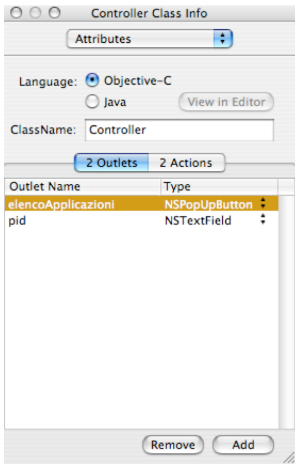
Per discuterne il funzionamento, e per approfondire comunque l'argomento inerente ai menu, decidiamo di realizzare un'applicazione, il nostro famoso *Esempio3*, che esaminerà tutte le applicazioni aperte in un dato momento, le elencherà in un menu a comparsa, e dietro selezione di una di esse dal menu ne riporterà il *Process ID*, detto *PID* dagli amici, che è il numero univoco che il sistema operativo assegna ad ogni programma in esecuzione (qui non ce ne facciamo niente del PID, ma ad esempio vi può servire se volete fare qualche cosa col Terminale, come terminare l'applicazione, verificare quanto tempo processore o quanta memoria occupa).

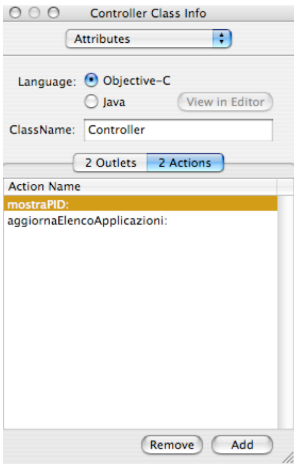
Chiudiamo quindi tutti gli esempi aperti in InterfaceBuilder e in XCode e creiamo una nuova *Cocoa application* con XCode, che chiameremo *Esempio3*. Come al solito, troviamo subito nella finestra del progetto l'icona di *MainMenu.nib* e la apriamo, così da disegnare l'interfaccia utente. Fate in modo che essa appaia come mostrato qui in figura. *PID:* e *0000* sono due campi statici di testo diversi, mentre il menu a comparsa è ottenuto trascinando l'elemento di interfaccia identificato come *Item1* nella finestrella dei vari controlli di Cocoa mostrata qui di fianco. Come avete modo di verificare soffermandovi col mouse su di esso, si tratta di un oggetto appartenente alla classe `NSPopUpButton`. Non dobbiamo preoccuparci di dare dei nomi alle varie voci che compaiono nel menu a comparsa, in quando lo faremo direttamente dal codice che tra poco scriveremo. Comunque, se avessimo voluto farlo dall'interno di InterfaceBuilder, il gioco non sarebbe stato diverso da quanto fatto per l'*Esempio2*: avremmo selezionato elementi di menu e separatori dal pannello apposito della finestrella con i vari oggetti dell'interfaccia, e mediante drag&drop li avremmo aggiunti al nostro menu a comparsa. Qui facciamo tutto dall'interno del codice sia perché così vediamo un modo alternativo di procedere, sia perché il nostro menu a comparsa conterrà l'elenco delle applicazioni attive in quel momento, e ovviamente non possiamo sapere a priori quale sarà (e non sarà nemmeno sempre lo stesso, ecco perché abbiamo aggiunto un pulsante che ci serve per aggiornare, se necessario, il menu a comparsa nel caso in cui nuove applicazioni siano state lanciate o altre applicazioni siano state chiuse).





Usiamo sempre la solita tecnica per creare la classe di controllo: dalla finestra denominata *MainMenu.nib* selezionate il pannello *Classes*, trovate la classe `NSObject` e dal menu *Classes* selezionate*Sublcass NSObject*. Date alla nuova classe così creata il nome `Controller`. Aggiungete alla classe `Controller` due outlet e due azioni, come mostrato nelle figure all'inizio della pagina seguente. Selezionate di nuovo la finestra denominata *MainMenu.nib* e da questa, all'interno del pannello *Classes*, la classe `Controller`; quindi, dal menu *Classes*, slezionate *Create Files for Controller* e a seguire *Instantiate Controller*. A questo punto potete fare i collegamenti (con la solita tecnica di tenere premuto il tasto *control* e trascinare col mouse):





l'outlet *elencoApplicazioni* andrà ovviamente collegato al menu a comparsa, mentre quello *pid* andrà collegato al campo di testo con su scritto *000*. Viceversa, il menu a comparsa andrà collegato all'azione *mostraPID:* e il pulsante all'azione restante, ovvero *aggiornaElencoApplicazioni:*. Salvate le modifiche e tornate a XCode, perché dobbiamo iniziare a scrivere un po' di codice.
Come prima cosa spostate i due file *Controller.h* e *Controller.m* dalla cartella *Other Sources* alla cartella *Classes* nella finestra principale del progetto. Poi veriricate che il file di interfaccia *Controller.h* sia come questo qua:

```
/* Controller */

#import <Cocoa/Cocoa.h>

@interface Controller : NSObject
{
    IBOutlet NSPopUpButton *elencoApplicazioni;
    IBOutlet NSTextField *pid;
}
- (IBAction)aggiornaElencoApplicazioni:(id)sender;
- (IBAction)mostraPID:(id)sender;
@end
```

Niente di nuovo sotto il sole. Aprite allora il file di implementazione *Controller.m* e scriveteci il seguente codice:

```
#import "Controller.h"

@implementation Controller

- (void)awakeFromNib
{
    [self aggiornaElencoApplicazioni:self];
}

- (IBAction)aggiornaElencoApplicazioni:(id)sender
{
    NSArray            *apps;
    NSEnumerator       *en;
    id                 obj;

    [elencoApplicazioni removeAllItems];
    [elencoApplicazioni addItemWithTitle:@"Nessuna applicazione"];
    [[elencoApplicazioni itemWithTitle:@"Nessuna applicazione"]
                        setRepresentedObject:[NSNumber numberWithInt:0]];
    apps=[[NSWorkspace sharedWorkspace] launchedApplications];
```

```
    en=[apps objectEnumerator];
    while(obj=[en nextObject])
    {
        [elencoApplicazioni addItemWithTitle:
            [obj objectForKey:@"NSApplicationName"]];
        [[elencoApplicazioni itemWithTitle:
            [obj objectForKey:@"NSApplicationName"]]
                setRepresentedObject:
                [obj objectForKey:@"NSApplicationProcessIdentifier"]];
    }
    [elencoApplicazioni selectItemAtIndex:0];
    [self mostraPID:elencoApplicazioni];
}

- (IBAction)mostraPID:(id)sender
{
    [pid setStringValue:
            [[[sender selectedItem] representedObject] stringValue]];
}

@end
```

Compilatelo, eseguitelo e divertitevi. Adesso vediamo che cosa succede. Innanzitutto c'è l'ormai consueto metodo `awakeFromNib`, che usiamo per far sì che già all'avvio la nostra applicazione mostri un elenco aggiornato e sensato delle applicazioni attive. Ovviamente per fare questo non dobbiamo inventarci nulla di nuovo, dal momento che il metodo `aggiornaElencoApplicazioni:` è lì per questo, non dobbiamo fare altro che chiamarlo).

Le prime cose veramente interessanti accadono proprio in `aggiornaElencoApplicazioni:`. Il nostro menu a comparsa viene innanzitutto privato di tutti i suoi elementi, grazie ad una chiamata a `removeAllItems`, metodo implementato dalla classe `NSPopUpButton`. Questo è indispensabile sia quando l'applicazione viene avviata (vi ricordo che nell'interfaccia utente abbiamo lasciato i tre elementi di default, genericamente chiamati *Item1*, *Item2* e *Item3*, sia quando l'utente preme il pulsante *Aggiorna elenco applicazioni*, dal momento che dovremo ricostruire il menu e quindi rimuovere ogni traccia delle voci precedenti.

Poi iniziamo a costruire il nostro menu, sfruttando i metodi che la classe `NSPopUpButton` ci offre. In particolare il metodo `addItemWithTitle:` ci consente di aggiungere una voce al menu. Poi facciamo una cosa strana; cerchiamo di capire di che cosa si tratta. Nel paragrafo precedente, quando parlavamo di menu, abbiamo collegato un'azione alla voce del menu (era la voce *Nome Computer*, ricordate?). Bene, l'azione che veniva chiamata, che si chiamava `visualizzaNome:`, aveva un argomento, il solito `sender`. Esso era ovviamente un puntatore ad un oggetto di classe `NSMenuItem`, nient'altro che l'elemento del menu che aveva causato la chiamata a `visualizzaNome:`. Siccome sapevamo che avevamo un solo elemento nel menu, non ci siamo preoccupati di come fare per distinguere una voce dall'altra. Certo, con un menu statico è sufficiente che ogni elemento del menu sia collegato ad un'azione diversa. Ma se il menu è dinamico, come in questo *Esempio3*? Ovvero, se il contenuto del menu non è noto a priori? Per non parlare del fatto, poi, che con i menu a comparsa, l'argomento `sender` dell'azione che viene chiamata non è un puntatore alla voce del menu che è stata selezionata dall'utente, ma è un puntatore al menu a comparsa vero e proprio, ovvero ad un oggetto di classe`NSPopUpButton`; l'azione che viene chiamata è quindi sempre la stessa, indipendentemente da quale sia la voce del menu selezionata dall'utente. Che si fa in questi casi? Bisogna trovare un modo per distinguere un elemento del menu da un altro. Gli oggetti di classe `NSMenuItem`, pur non ereditando da `NSControl`, dispongono anch'essi di un *tag*, così come i pulsanti o i campi di testo. Potremmo quindi associare ad ogni voce nel nostro menu un *tag* diverso, col solito metodo `setTag:`, da ripescare poi col metodo `tag` all'interno dell'azione

mostraPID: così da capire quale applicazione l'utente abbia selezionato. In realtà non c'è bisogno di usare un *tag*, perché le applicazioni sono tutte diverse per nome, quindi sarebbe sufficiente andare a vedere il titolo della voce del menu selezionata dall'utente per conoscere il nome dell'applicazione. Ma sarebbe uno spreco. Infatti, come vedremo tra poco, il metodo che invocheremo per conoscere l'elenco delle applicazioni in esecuzione sul Mac ci fornisce già direttamente anche il loro PID. Non potremmo semplicemente memorizzarlo da qualche parte, così che quando l'utente seleziona un'applicazione dal menu a comparsa, ci limitiamo a ripescarlo là dove l'abbiamo memorizzato e lo visualizziamo? Se così non facessimo, ci toccherebbe scoprire quale applicazione sia stata scelta dall'utente (per nome o per *tag*), chiedere nuovamente al sistema di fornirci l'elenco di tutte le applicazioni in esecuzione, cercare in questo elenco quella selezionata dall'utente ed estrarne il PID. Scomodo, e soprattutto è una perdita di tempo, perché faremmo due volte la stessa cosa.

Beh, che ci crediate o no, possiamo fare esattamente quello che speravamo: memorizzare da qualche parte il PID, in attesa di poterlo richiamare all'occorrenza quando l'utente seleziona una voce dal menu. Questa fantastica comodità ci è offerta dal metodo setRepresentedObject: implementato dalla classe NSMenuItem. In pratica, è possibile associare ad un elemento di un menu *un qualunque oggetto appartenente a qualunque classe*. Siccome si possono associare oggetti ma non variabili numeriche, scegliamo di associare ad ogni voce del menu un oggetto di classe NSNumber, contenente, sotto forma di numero intero, il PID dell'applicazione corrispondente alla voce del menu. Ora il primo elemento del menu a comparsa è in realtà *Nessuna applicazione*, per cui associamo a questa voce un PID pari a zero. Usiamo, per fare ciò, la classe NSNumber, che può essere allocata e inizializzata come autorelease mediante un metodo del tipo numberWith…:, e il metodo itemWithTitle: della classe NSPopUpButton che consente di ottenere l'oggetto di classe NSMenuItem che ha il titolo specificato nell'argomento.

Subito dopo chiediamo al sistema operativo, per mezzo della classe NSWorkspace, di fornirci un elenco (un oggetto di classe NSArray) delle applicazioni in esecuzione in quel momento. Quest'array contiene dei *dizionari*, un dizionario per applicazione. Ci occuperemo di dizionari in un capitolo successivo. Qui limitatevi ad osservare che all'interno del loop while percorriamo tutti gli oggetti contenuti nell'array, e da essi estraiamo in qualche maniera l'informazione riguardante il nome dell'applicazione (che usiamo come titolo della nuova voce del menu) e il PID dell'applicazione, che usiamo come *represented object* della voce del menu; il PID fornitoci con questa tecnica è già un oggetto di classe NSNumber, proprio come quello che abbiamo aggiunto noi alla prima voce del menu, quella che recita *Nessuna applicazione*. Astuto, vero? Per finire, diciamo al menu a comparsa di selezionare la prima voce (quella numero 0), e, visto che il menu è cambiato, aggiorniamo il valore del PID dell'applicazione selezionata (quella corrispondente alla prima voce del menu a comparsa, ovvero *Nessuna applicazione*), mediante una chiamata a mostraPID:.

mostraPID:, a sua volta, usa la stessa tecnica del *represented object* (ovviamente) per andare a ripescare il PID dell'applicazione selezionata dall'utente. Dapprima chiediamo a sender (un oggetto di classe NSPopUpButton, che poi è il nostro outlet elencoApplicazioni) di restituirci un oggetto di classe NSMenuItem corrispondente alla voce del menu che è stata selezionata (col metodo selectedItem), quindi con questo oggetto andiamo a richiamare il suo representedObject. Quest'oggetto già sappiamo essere di classe NSNumber, quindi ne leggiamo il contenuto sotto forma di stringa (oggetto di classe NSString) che assegnamo, mediante il metodo setStringValue:, al campo di testo statico dell'interfaccia utente (l'outlet pid). E il gioco è fatto. Tutte le volte che l'utente seleziona una voce dal menu, il metodo mostraPID: esegue tutte queste operazioni. Tutte le volte che l'utente preme il pulsante per aggiornare l'elenco delle applicazioni (o quando l'applicazione viene lanciata), il metodo aggiornaElencoApplicazioni: costruisce il nuovo menu a comparsa facendosi dire da NSWorkspace nome e PID delle applicazioni in esecuzione, e usandoli come titolo e *represented object* delle voci che costituiranno il menu.

Tag e *represented object* possono essere usati per tutte le voci dei menu, quelli ancorati alla barra dei menu, quelli a comparsa, i sottomenu, ecc. Le azioni, invece, possono essere collegate solo ai menu che compaiono nella barra dei menu o ai loro sottomenu, dal momento

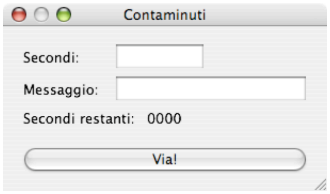
che nei menu a comparsa l'azione è collegata non già ad ogni singola voce del menu, ma al menu stesso (in effetti, esso è di classe `NSPopUpButton`, quindi dobbiamo considerarlo come se fosse un pulsante, che ha la sua azione collegata).

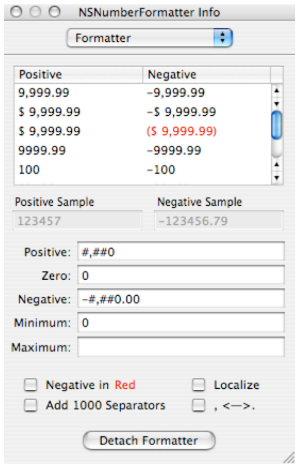
5. Timer e pannelli

Ci è già capitato, in un esempio presentato nei capitoli precedenti, di aver bisogno di eseguire automaticamente una certa operazione ad intervalli regolari. Ci avrebbe fatto comodo, ad esempio, per mantenere aggiornato il campo di testo dell'ora nell'*Esempio2*, o per mantenere aggiornata senza bisogno di intervento da parte dell'utente la lista delle applicazioni aperte nell'*Esempio3*. Ma c'è un modo per fare sì che un'operazione a nostra scelta venga eseguita automaticamente ad intervalli regolari, sempre a nostra scelta? La risposta è sì, se no non saremmo qui a farci queste domande. E la classe che ci permette di fare tutto ciò si chiama `NSTimer`.

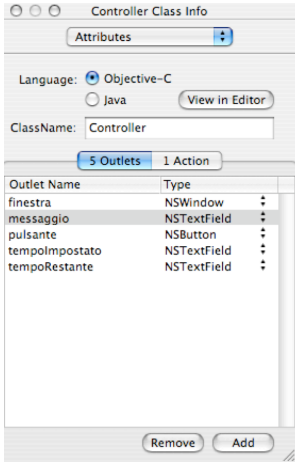
Nell'esempio che stiamo per fare, realizzeremo un rudimentale timer, sapete, di quelli che si usano come contaminuti quando si fanno le uova sode. Un banalissimo timer: l'utente imposta un certo numero di secondi, poi fa partire il timer; un orologio tiene il conto alla rovescia, aggiornato in tempo reale; quando il tempo arriva a zero, un pannello ci informerà che il tempo è scaduto.

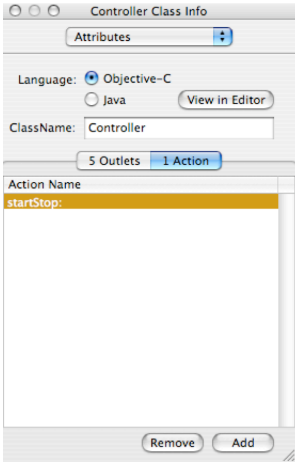
Come al solito, chiudiamo tutti i documenti aperti in XCode e InterfaceBuilder. Poi, da XCode, creiamo una nuova *Cocoa application*, che, indovinate un po', chiameremo *Esempio4*. Al solito, dalla finestra del progetto facciamo due click su *MainMenu.nib* e, da InterfaceBuilder, iniziamo a creare l'interfaccia utente. Impostate la finestra principale dell'applicazione affinché appaia come in figura. Il campo di testo contenente la scritta *0000* è a sé stante, è un campo di testo diverso rispetto a quello con la scritta *Secondi restanti:*. Nel campo editabile di testo in cui andranno inseriti i secondi per cui contare (il primo in alto), inserite un formattatore per numeri (ricordate? L'abbiamo fatto nell'*Esempio1*), e impostatelo affinché sia configurato come nella figura qui di fianco. Questo tipo di impostazione obbligherà l'utente ad inserire solo numeri interi positivi (il fatto che siano interi non è assolutamente necessario, i timer funzionano benissimo anche per intervalli di tempo non interi, in secondi, ma il fatto che siano positivi lo è, ovviamente). Quindi create alla solita maniera la classe di controllo (c'è bisogno che vi ricordi come si fa? No, vero?), chiamandola *Controller*, e aggiungetele 5 outlet e un'azione, come indicato nelle figure qui di fianco. Create i file per la classe di controllo ed istanziatela nel pannello *Instances* della finestra *MainMenu.nib*. Collegate l'outlet *tempoImpostato* al primo campo di testo editabile (quello in alto, di fianco alla scritta *Secondi:*), l'outlet *messaggio* al secondo campo di testo editabile, l'outlet*tempoRestante* al campo di testo statico con la scritta *0000*, l'outlet *finestra* con la finestra (ricordate: la *barra del titolo* della finestra) e l'outlet *pulsante* con l'unico pulsante che c'è (quello denominato *Vai!*). Collegate il medesimo pulsante con l'azione *startStop:*.





Salvate le modifiche e tornate ad XCode. Spostate i file *Controller.h* e *Controller.m* dalla cartellina *Other Sources* alla cartellina *Classes* nella finestra del progetto in XCode, quindi accertatevi che il file di interfaccia *Controller.h* abbia l'aspetto seguente (c'è parecchia roba da aggiungere, quindi datevi da fare):





```
/* Controller */

#import <Cocoa/Cocoa.h>

#define   kTempoDiAggiornamento       1 // in secondi
#define   kStartTag                   1
#define   kStopTag                    2
#define   kStartNome                  @"Via!"
#define   kStopNome                   @"Stop"

@interface Controller : NSObject
{
    IBOutlet NSTextField    *messaggio;
    IBOutlet NSButton       *pulsante;
    IBOutlet NSTextField    *tempoImpostato;
    IBOutlet NSTextField    *tempoRestante;
     IBOutlet NSWindow      *finestra;

     NSTimer                *timerTotale;
     NSTimer                *timerAggiornamento;
}

- (IBAction)startStop:(id)sender;
- (void)aggiornaConteggio:(NSTimer *)t;
- (void)scadenza:(NSTimer *)t;
- (void)inizio;
- (void)fermaTutto;
- (void)sheetDidEnd:(NSWindow *)sheet returnCode:(int)returnCode
                                     contextInfo:(void *)contextInfo;
@end
```

Poi, editate il file di implementazione *Controller.m* in modo che appaia come segue:

```
#import "Controller.h"

@implementation Controller

- (void)awakeFromNib
{
     [pulsante setTag:kStartTag];
     [pulsante setTitle:kStartNome];
}

- (IBAction)startStop:(id)sender
{
     if([sender tag]==kStartTag)
          [self inizio];
     else
          [self fermaTutto];
}

- (void)aggiornaConteggio:(NSTimer *)t
```

```
{
     if(timerTotale)
          [tempoRestante setIntValue:
                         [[timerTotale fireDate] timeIntervalSinceNow]];
     else
          [tempoRestante setIntValue:0];
}

- (void)scadenza:(NSTimer *)t
{
     NSBeginAlertSheet(@"Tempo scaduto!",@"OK",@"Riparti",nil,finestra,
                    self,@selector(sheetDidEnd:returnCode:contextInfo:),
                                        nil,nil,[messaggio stringValue]);
     [self fermaTutto];
}

- (void)inizio
{
     timerTotale=[NSTimer scheduledTimerWithTimeInterval:
                                        [tempoImpostato intValue]
                                        target:self
                                        selector:@selector(scadenza:)
                                        userInfo:nil
                                        repeats:NO];
     timerAggiornamento=[NSTimer scheduledTimerWithTimeInterval:
                                    kTempoDiAggiornamento
                                    target:self
                                    selector:@selector(aggiornaConteggio:)
                                    userInfo:nil
                                    repeats:YES];
     [messaggio setEnabled:NO];
     [tempoImpostato setEnabled:NO];
     [pulsante setTag:kStopTag];
     [pulsante setTitle:kStopNome];
}

- (void)fermaTutto
{
     if(timerTotale)
     {
          [timerTotale invalidate];
          timerTotale=nil;
     }
     if(timerAggiornamento)
     {
          [timerAggiornamento invalidate];
          timerAggiornamento=nil;
     }
     [messaggio setEnabled:YES];
     [tempoImpostato setEnabled:YES];
     [pulsante setTag:kStartTag];
     [pulsante setTitle:kStartNome];
```

```
}

- (void)sheetDidEnd:(NSWindow *)sheet
              returnCode:(int)returnCode contextInfo:(void *)contextInfo
{
     if(returnCode==NSAlertDefaultReturn)
     {
          // basta così
     }
     else if(returnCode==NSAlertAlternateReturn)
     {
          // ricominciamo
          [self inizio];
     }
     else if(returnCode==NSAlertOtherReturn)
     {
          // non abbiamo un "other button"
     }
     else if(returnCode==NSAlertErrorReturn)
     {
          NSLog(@"Misterioso errore nell'alert");
     }
}

@end
```

Compilate ed eseguite il programma, e notate che c'è un piccolo bug (per ora non abbiamo gli strumenti per risolverlo): scrivete i secondi, scrivete il messaggio, ma se non premete "a-capo" dopo aver scritto il messaggio questo non comparirà nel pannello che viene visualizzato alla fine del conto alla rovescia. Fatevi anche un paio di uova sode usando questo contaminuti. Poi, mentre lasciate che si raffreddino, discutiamo un po' insieme delle tante novità qui introdotte.

Andiamo per ordine logico. Il primo metodo che consideriamo è `startStop:`, dal momento che è quello che viene chiamato quando fate click sul pulsante che avvia il timer. Esso sfrutta il vecchio trucchetto dei *tag* assegnati ai pulsanti, ma in maniera nuova. Infatti, se in passato avevamo scelto di associare la stessa azione a due pulsanti diversi, differenziandoli per il *tag*, qui abbiamo un pulsante solo e una sola azione, ma il pulsante può effettuare due cose diverse a seconda del momento (far partire il timer o fermarlo se è già partito). Per ognuno dei due compiti, quindi, definiamo un *tag* diverso, e differenziamo il da farsi in base ad esso. In particolare, se il pulsante nel momento in cui è stato premuto aveva un *tag* pari a `kStartTag`, allora nessun timer era in corso, e chiamiamo il metodo `inizio`, che ha il compito di avviarne uno; se invece il *tag* era `kStopTag`, vuol dire che un timer era già in esecuzione, e allora chiamiamo il metodo `fermaTutto`, che ha il compito di fermare il timer in corso.

Il metodo `inizio` definisce due timer, mediante un costrutto che vi diventerà familiare. Il metodo di creazione ed inizializzazione vuole, come primo argomento (quello di `scheduledTimerWithInterval:`) il tempo, in secondi, di durata del timer; può anche essere un numero decimale. Il secondo argomento (quello di `target:`) indica l'oggetto destinato ad essere avvisato dal sistema operativo quando il timer scade (è passato il numero di secondi che gli avete passato a primo argomento); nel nostro caso abbiamo deciso che il nostro oggetto di controllo deve essere avvisato, e quindi abbiamo passato `self`. Il terzo argomento (quello di `selector:`) è il metodo che dovrà essere implementato dall'oggetto che viene chiamato alla scadenza del timer; il formato di questo metodo è standard, restituisce `void` e richiede un argomento; ma ci ritorneremo tra poco; notate comunque la tecnica: tutte le volte che bisogna comunicare a Cocoa il nome di un metodo che Cocoa stesso richiamerà al momento opportuno,

esso va presentato all'interno della *direttiva* `@selector()`; naturalmente, il formato del metodo che passiamo dipenderà da caso a caso; per quelli di voi che hanno una certa familiarità col C, è un po' come se stessimo passando un puntatore ad una funzione. Il quarto argomento, quello di `userInfo:`, è un oggetto qualunque a scelta del programmatore; potete mettere qui quello che volete; potrete recuperare questo oggetto (e quindi tutte le informazioni che avete memorizzato in esso) nel momento che preferite, ad esempio quando il conto alla rovescia scade (se `mioTimer` è l'oggetto di classe `NSTimer` a cui avete assegnato un certo oggetto come argomento di `userInfo:`, potete richiamare tale oggetto mediante `[mioTimer userInfo];`); qui, visto che non avevamo niente di particolare da ricordarci, abbiamo messo `nil`. L'ultimo argomento è un *flag* booleano (vero/falso); se è vero indica che il conto alla rovescia, una volta giunto a zero e chiamato il metodo scelto dall'utente, riparte automaticamente; se è falso, il timer termina lì.

Come vedete il metodo `inizio` crea due timer; uno ha una cadenza di 1 secondo, e verrà utilizzato per aggiornare il tempo residuo nella finestra principale del programma; l'altro ha come scadenza il tempo impostato dall'utente nell'apposito campo di testo. Infine il metodo `inizio` disabilita la possibilità di editare il testo nei campi relativi al messaggio e al tempo totale (a timer in corso non vogliamo consentire che questi parametri vengano modificati), e cambia *tag* e titolo al pulsante, trasformandolo in un pulsante in grado, se premuto, di fermare il timer appena avviato.

Ogni volta che `timerAggiornamento` scade (una volta al secondo, visto il valore che abbiamo dato a `kTempoDiAggiornamento` nel file di interfaccia), viene chiamato il metodo `aggiornaConteggio:`. Cocoa si aspetta che il metodo implementato dall'oggetto target di un timer restituisca `void` e accetti, come unico argomento, un oggetto di classe `NSTimer` (sì, è proprio il timer che ha causato la chiamata al metodo). In linea di massima questo argomento non ci serve, perché il metodo `aggiornaConteggio:` nel nostro programma risponde alla chiamata di un solo timer (`timerAggiornamento`), di cui abbiamo tenuto traccia in una variabile,`timerAggiornamento` appunto. Tuttavia nulla vieta di far sì che un unico metodo risponda a più timer diversi; poterli distinguere, allora, grazie all'argomento passato al metodo (e poter recuperare l'oggetto eventualmente passato come argomento di `userInfo:`) diventa allora indispensabile. Sì, ma che cosa fa `aggiornaConteggio:`? Come prima cosa verifica che il timer principale esista ancora. Perché questa verifica? Beh, potreste aver impostato il timer principale a un numero *non intero* di secondi (in questo programma l'abbiamo vietato per via del formattatore dato al campo di testo, ma in linea di principio è possibile), o in generale ad un tempo che non è un multiplo intero del tempo associato al timer secondario (quello di aggiornamento). È quindi possibile che il timer principale scada tra una chiamata e quella successiva a `aggiornaConteggio:`; che succede in questo caso? Succede che `timerTotale` non è più un timer valido, perché già scaduto, e non è più possibile ottenere da esso la data di scadenza (essendo già scaduto). Come prima cosa, quindi, verifichiamo che `timerTotale` sia ancora attivo (l'istruzione `if`), poi, in caso affermativo, chiediamo a `timerTotale` di comunicarci la sua data di scadenza (il metodo `fireDate`). Esso restituisce un oggetto di classe `NSDate`, riportante una data (comprensiva di giorno, mese, anno, ora, minuti e secondi) nel futuro (dal momento che `timerTotale` deve ancora scadere). La classe `NSDate` ci mette a disposizione un metodo, `timeIntervalSinceNow`, che restituisce il numero di secondi che devono ancora passare (o che sono già passati) per raggiungere nel futuro (o nel passato) la data memorizzata nell'oggetto. Esattamente quello che ci serve mettere nel campo di testo statico che riporta il tempo residuo. Ovviamente, se `timerTotale` è già scaduto, metteremo `0` come tempo residuo. Infine notate che non è affatto necessario dover usare sempre due timer, uno totale e uno con cadenza più ravvicinata; quest'ultimo, infatti, serve solo se volete dare un feedback del tempo che passa (aggiornamento del tempo residuo, aggiornamento di una barra di progressione, ecc.) o se comunque vi serve nel vostro programma; in molti casi, non dovete fare altro che usare il solo timer principale, perché di quello ne avrete già in abbondanza.

Nel momento in cui è trascorso tutto il tempo richiesto dall'utente, `timerTotale` scade, e il metodo `scadenza:` viene chiamato. Esso fa due cose. La prima è nuova ed è molto importante: visualizza un pannello attaccato alla finestra principale del programma. Come fa? Ma con la funzione `NSBeginAlertSheet()`, naturalmente! Il primo argomento di questa

funzione è il testo che comparirà in grassetto nel pannello; abbiamo scelto di scrivere *Tempo scaduto!*, ma ovviamente la cosa è del tutto arbitraria (potevamo scrivere qui il messaggio scelto dall'utente, ad esempio). L'argomento successivo è il titolo del pulsante *principale* del pannello, quello che obbligatoriamente ci deve essere (serve almeno un pulsante da premere, perché l'utente possa chiudere il pannello); l'abbiamo chiamato *OK*. L'argomento successivo è il titolo del pulsante *alternativo* del pannello; è opzionale (se non lo volete passate `nil` a questo argomento); noi abbiamo scelto di chiamarlo *Riparti*. L'argomento successivo è il titolo del pulsante*addizionale* del pannello; è opzionale anch'esso, e noi abbiamo deciso di non implementarlo, passando `nil`. L'argomento successivo è un oggetto di classe `NSWindow`, ovvero la finestra alla quale dovrà essere attaccato il pannello; passiamo `finestra`, dal momento che è l'outlet collegato alla finestra principale del nostro programma. Quindi dobbiamo prenderci cura di tre argomenti, il cui compito è quello di informare il programma su quale pulsante sia stato premuto dall'utente per chiudere (*dismettere* si dice anche ogni tanto) il pannello; il primo di questi tre argomenti dice quale oggetto dovrà essere informato di ciò (abbiamo scelto il nostro oggetto di controllo, ovvero `self`); il secondo di questi tre argomenti specifica il metodo che dovrà essere chiamato *nel momento in cui l'utente preme uno dei pulsanti del pannello*, ma *prima* che questo venga effettivamente chiuso; anche qui la tecnica è quella di usare la direttiva `@selector()`, passando il nome di un metodo che richiede necessariamente tre argomenti ben precisi e definiti (e che chiamerete *sempre* `sheetDidEnd:returnCode:contextInfo:` per evitarvi inutili mal di pancia). Se non ve ne frega niente di sapere quando l'utente ha chiuso il pannello (e ovviamente di sapere premendo quale pulsante l'ha chiuso), potete tranquillamente passare `nil` a questo argomento; l'ultimo di questi tre argomenti è il metodo (da passare sempre con la direttiva `@selector()`) che verrà chiamato *subito dopo che il pannello si è chiuso*; non mi è venuto in mente nessun caso in cui uno possa voler essere informato subito prima e subito dopo che il pannello si chiuda, visto che nel frattempo non succede nulla di interessante, ma nel caso vogliate esserne informati, il prototipo del metodo che dovrete implementare è questo:

```
sheetDidDismiss:(NSWindow *)sheet returnCode:(int)returnCode
                                        contextInfo:(void *)contextInfo;
```

Siccome non ci interessa essere avvisati due volte delle scelte compiute dall'utente, implementiamo solo il metodo precedente e non questo, passando `nil` come argomento. L'argomento successivo è un oggetto qualunque, quello che volete voi, lo stesso oggetto che vi verrà messo a disposizione come argomento di `contextInfo` nell'implementazione di `sheetDidEnd:returnCode:contextInfo:` e di `sheetDidDismiss:returnCode:contextInfo:`.

Per finire, un oggetto di classe `NSString` individua il messaggio che volete scrivere più in piccolo nel pannello; abbiamo scelto qui di mettere il messaggio scritto dall'utente.

Subito dopo aver visualizzato il pannello (ma *prima* che l'utente lo chiuda), il metodo `scadenza:` chiama `fermaTutto` (esattamente come quando l'utente ferma manualmente il timer prima che scada).

Sì, ma che cosa fa `fermaTutto`? Due cose molto importanti. Innanzitutto, se `timerTotale` e `timerAggiornamento` sono ancora validi, li invalida (il timer viene cancellato dalla lista delle operazioni periodiche che il sistema operativo mantiene), poi ripristina *tag* e titolo del pulsante ai valori necessari per consentire un nuovo avvio del conto alla rovescia, riabilitando anche la possibilità di editare il contenuto dei campi di testo contenenti il tempo totale e il messaggio da visualizzare.

Il metodo `sheetDidEnd:returnCode:contextInfo:` invece riceve come primo argomento un puntatore al pannello la cui chiusura ha causato la chiamata al metodo. La ragione è sempre la stessa: potreste avere mille pannelli e un solo metodo chiamato a rispondere della chiusura di tutti e mille, e volete poterli distinguere l'uno dall'altro. Noi di pannello ne abbiamo solo uno, e allora non ci curiamo del primo argomento. Non ci curiamo nelleno del terzo, `contextInfo`, perché abbiamo passato `nil` quando abbiamo chiamato `NSBeginAlertSheet()`,

ma naturalmente se avessimo voluto qui avremmo potuto recuperare questo oggetto. Il secondo argomento è invece un numero intero che può assumere tre valori distinti: `NSAlertDefaultReturn` quando l'utente chiude il pannello premendo il pulsante *principale* (quello che noi abbiamo chiamato *OK*), `NSAlertAlternateReturn` quando l'utente chiude il pannello premendo il pulsante *alternativo* (quello che noi abbiamo chiamato *Ripeti*), e `NSAlertOtherReturn` se l'utente preme il pulsante *addizionale* (quello che noi non abbiamo implementato). L'unico caso che ci interessa è quello in cui l'utente sceglie di ripartire con lo stesso timer, nel qual caso chiamiamo il metodo `inizio`.

Per finire, `awakeFromNib` si assicura che, non appena le risorse dell'applicazione vengono caricate da file, nome e *tag* del pulsante abbiano il valore corretto.

Se tutta questa faccenda del pannello e dei metodi che vengono chiamati quando l'utente lo chiude vi sembra macchinosa (ma, credetemi, non lo è, bisogna solo usarla due o tre volte e ci si fa l'abitudine) e volete qualche cosa di più semplice, potete visualizzare un *box di dialogo* in luogo di un pannello; esso, oltre a non offrirvi l'effetto spettacolare del pannello che spunta dalla barra del titolo della finestra, ha l'inconveniente che sospende qualunque cosa stia facendo la vostra applicazione finché l'utente non lo chiude premendo uno dei pulsanti forniti; un pannello, invece, consente all'applicazione di continuare a funzionare e di rispondere ai comandi dell'utente, anche se il pannello resta ancora attivo (potete ad esempio interagire con i menu dell'applicazione e con altre finestre). Dicevamo, se vi va bene un box di dialogo in luogo di un pannello, potete usare questa funzione:

```
int NSRunAlertPanel(NSString *title, NSString *msg,
                    NSString *defaultButton, NSString *alternateButton,
                    NSString *otherButton,...)
```

Il primo argomento è il titolo del box di dialogo, il secondo il testo da visualizzare come messaggio, seguono i titoli dei soliti tre puslanti (*principale*, *alternativo* e *addizionale*, passate `nil` per quelli che non volete usare). Per finire ci sono argomenti opzionali, che servono se avete formattato il messaggio da visualizzare con una sintassi simile a quella che usate con `printf()` (avete presente il nostro box di approfondimento su `NSLog()`? Ecco, quella roba lì). La funzione blocca la vostra applicazione finché non premete uno dei pulsanti del box di dialogo. Il numero del pulsante che avete premuto, rappresentato da una delle costanti che abbiamo usato anche in precedenza, viene restituito e potete assegnarlo ad una variabile così da poter poi prendere le decisioni che vi paiono più opportune. Se usate questa funzione in luogo di quella che crea i pannelli, non vi serviranno più i metodi usati in precedenza, `sheetDidEnd:returnCode:contextInfo:` e `sheetDidDismiss:returnCode:contextInfo:`.

È anche possibile usare pannelli e box di dialogo diversi da quelli standard, creandoseli con InterfaceBuilder e poi visualizzandoli a schermo (attaccati ad una finestra, per i pannelli); come questo si possa fare, va al di là degli obiettivi di questo manuale (ma quanto avete imparato in questo capitolo dovrebbe consentirvi di leggere senza troppi patemi la documentazione di Cocoa in merito a questi argomenti).

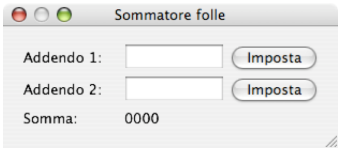
6. Notifiche

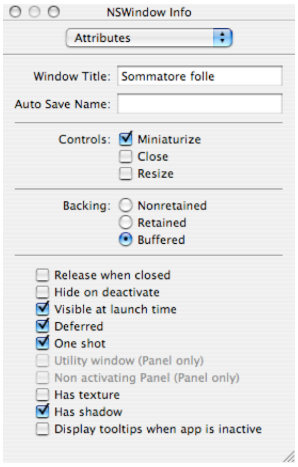
In questo capitolo ci occupiamo di un altro argomento fondamentale, cruciale per come è implementato Cocoa, e che voi potete usare per far fare cose inaudite alle vostre applicazioni: le *notifiche*. Se state pensando al vigile che viene a casa vostra a notificarvi una multa, non siete troppo lontani dal vero. Solo che qui si tratta di cose più piacevoli.

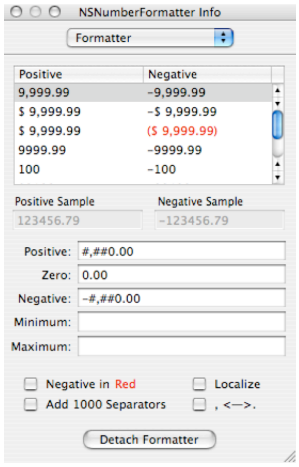
Facciamo finta che io sia alla ricerca di un lavoro. Individuerò allora un potenziale datore di lavoro, e gli manderò il mio *curriculum vitæ*, sperando che sortisca qualche effetto. In questo tipo di comunicazione c'è un mittente (io), un destinatario (il datore di lavoro), un messaggio (prendi il mio CV), e un oggetto da trasferire (il CV). Così è come avvengono normalmente le comunicazioni tra oggetti in Objective-C. Se `iomemedesimo` fosse l'oggetto corrispondente a me stesso, `datoreDiLavoro` fosse un puntatore ad un oggetto rappresentante un potenziale datore di lavoro e `currVitae` fosse un puntatore ad un oggetto rappresentante il mio CV, tale comunicazione potrebbe essere implementata all'interno del codice di `iomemedesimo` in questa forma: `[datoreDiLavoro beccatiStoCurriculum:currVitae];`.

Diverso è il discorso se analizzato dal punto di vista del datore di lavoro che cerca dipendenti. È difficile, infatti, che vada a bussare di porta in porta chiedendo a tutti coloro che aprono se siano alla ricerca di un lavoro e se siano disposti a lavorare per lui. Più facilmente, metterà un annuncio su un quotidiano, con i dettagli essenziali del lavoro e i recapiti a cui rivolgersi. Questo tipo di comunicazione, che non è uno-a-uno ma uno-a-molti, è quella che in Objective-C è implementata dal meccanismo delle notifiche. Un oggetto (il datore di lavoro) fa sapere (rende noto, ovvero *notifica*) che è successo qualche cosa, fornendo i dettagli essenziali e i propri estremi (una notifica). Tutti gli oggetti interessati (che si sono *registrati* per quel tipo di notifiche, un po' come, nella nostra analogia, essere abbonati ad un periodico di annunci economici) possono impiegare le informazioni ricevute mediante la notifica nel modo che ritengono più opportuno, ed eventualmente ricontattare chi ha *sollevato* la notifica. Astenersi perditempo.

L'esempio che svilupperemo in questo capitolo mostrerà come si possa rispondere alle notifiche, ma anche come se ne possano generare (*sollevare*). Mostrerà come questo meccanismo sia implementato da Cocoa per fornire ai nostri programmi un sacco di informazioni utili, e impiegherà in maniera smodata le notifiche per svolgere dei compiti che potrebbero essere compiuti in maniera più semplice con un normale scambio di messaggi tra oggetti; tuttavia, quest'uso esagerato delle notifiche sarà qui giustificato dal nobile intento di erudire, e sono sicuro che mi saprete perdonare. Facciamoci quindi coraggio, chiudiamo tutti i documenti aperti in XCode e in InterfaceBuilder, e creiamo, in XCode, un nuovo progetto di tipo *Cocoa application*, che chiameremo, manco a dirlo, *Esempio5*. Aprite con un doppio click il file di risorse *MainMenu.nib* e, dall'interno di InterfaceBuilder, disegnate l'interfaccia utente per la finestra principale dell'applicazione, come mostrato nella figura qui di fianco. Vi ricorda qualche cosa? Accertatevi che la finestra abbia le proprietà mostrate qui di fianco nella rispettiva finestra di informazioni, e aggiungete, per ognuno dei due campi editabili di testo, un formattatore con proprietà come quelle indicate in figura. Sempre dalla finestrella di informazioni, date al pulsante di fianco al primo addendo un *tag* pari ad 1 e al pulsante di fianco al secondo addendo un *tag* pari a 2.

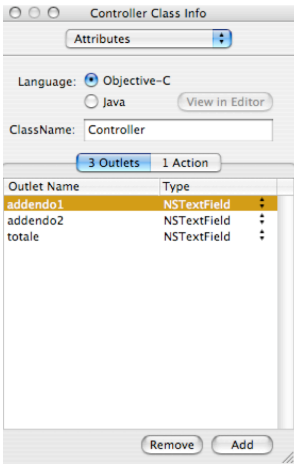


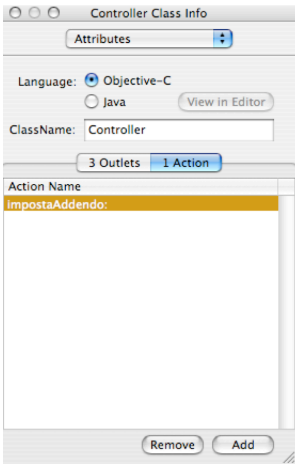




Dal pannello *Classes* della finestra *MainMenu.nib* sottoclassate poi `NSObject`

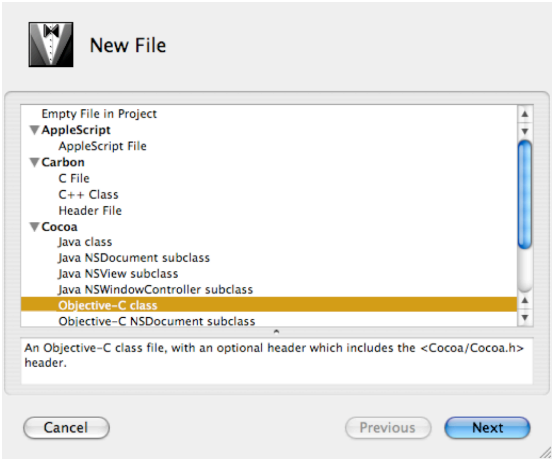
e create la classe `Controller`, alla quale assegnerete 3 outlet e un'azione, come mostrato nelle figure seguenti. Create i file per la classe di controllo e istanziatela. Collegate infine l'outlet *addendo1* al primo campo di testo editabile, l'outlet *addendo2* al secondo campo di testo editabile e l'outlet *somma* al campo di testo statico che contiene la scritta *0000*. Collegate entrambi i pulsanti all'azione*impostaAddendo:*. Salvate le modifiche e tornate ad XCode.





Qui, come al solito, iniziate a spostare le icone dei file *Controller.h* e *Controller.m* dalla cartellina *Other Sources* a quella *Classes*, così da fare un po' d'ordine. Quindi verificate che *Controller.h* abbia l'aspetto seguente (notate che c'è della roba da aggiungere):

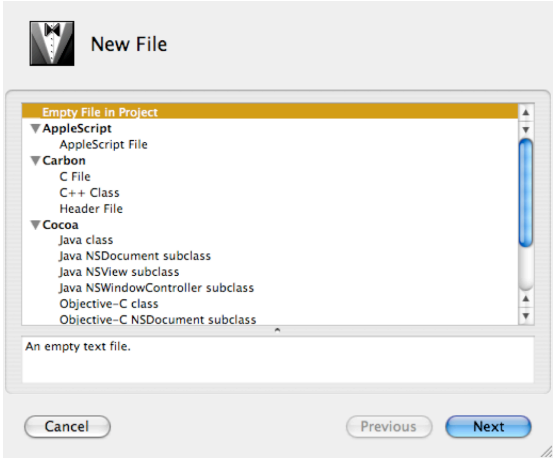
```
/* Controller */

#import <Cocoa/Cocoa.h>
#import "CostantiGenerali.h"
#import "Addendo.h"

#define   kDebug              YES

@interface Controller : NSObject
{
    IBOutlet NSTextField *addendo1;
    IBOutlet NSTextField *addendo2;
    IBOutlet NSTextField *totale;

     Addendo        *a1,*a2;
}
- (IBAction)impostaAddendo:(id)sender;
- (void)impostazioniPreliminari:(NSNotification *)n;
- (void)aggiornaSomma:(NSNotification *)n;
@end
```



Prima di editare il file di implementazione, raggiungete il menu *File* di XCode e selezionate *New File…*; scegliete di creare una nuova *Objective-C class*, come mostrato in figura. Chiamatela *Addendo*, lasciando invariate le altre impostazioni, in particolare quella che creerà automaticamente anche il file di interfaccia della classe (e non solo quello di implementazione). Osservate che nella cartellina *Classes* della finestra del progetto si sono aggiunti i due file *Addendo.h* e *Addendo.m*. Selezionate ancora *New File…* dal menu *File* e questa volta create un *Empty File in Project* (che vuol dire un file vuoto, che non contiene nulla), come mostrato nella figura seguente. Chiamatelo

CostantiGenerali.h, e osservate che anch'esso va ad aggiungersi alla lista di file presenti nella cartellina *Classes* della finestra del progetto.

Ora che abbiamo tutti i file che ci servono, iniziamo ad editarli. Partiamo proprio da *CostantiGenerali.h*, che è un file di header che ci servirà in più di un posto. Esso, in realtà, consta di una sola riga:




```
#define   kAddendoModificatoNotif
              @"addendo modificato"
```

Salvate le modifiche e passate al file *Addendo.h*, ovvero il file di interfaccia di una nuova classe (*non* di controllo) creata da noi:

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


@interface Addendo : NSObject
{
    double        valore;
}

- (id)init;
- (void)impostaValore:(double)val;
- (double)valore;

@end
```

Anche qui salvate le modifiche, poi passate al file di implenentazione *Addendo.m*:

```
#import "Addendo.h"

@implementation Addendo

- (id)init
{
    [super init];
    valore=0.0;
    return self;
}

- (void)impostaValore:(double)val
{
    valore=val;
    [[NSNotificationCenter defaultCenter]
                        postNotificationName:kAddendoModificatoNotif
                        object:self
                        userInfo:nil];
}
```

```
- (double)valore
{
     return valore;
}

@end
```

Salvate le modifiche e passate all'ultimo file, *Controller.m*:

```
#import "Controller.h"

@implementation Controller

- (id)init
{
     [super init];
     a1=[[Addendo alloc] init];
     a2=[[Addendo alloc] init];
     [[NSNotificationCenter defaultCenter] addObserver:self
                         selector:@selector(impostazioniPreliminari:)
                         name:@"NSApplicationDidFinishLaunchingNotification"
                         object:nil];
     return self;
}

- (void)dealloc
{
     [a1 release];
     [a2 release];
}

- (void)impostazioniPreliminari:(NSNotification *)n
{
     if(kDebug)
          NSLog(@"abbiamo ricevuto la notifica
                         di caricamento dell'applicazione");
     [[NSNotificationCenter defaultCenter] addObserver:self
                                  selector:@selector(aggiornaSomma:)
                                  name:kAddendoModificatoNotif
                                  object:nil];
}

- (IBAction)impostaAddendo:(id)sender
{
     switch([sender tag])
     {
          case 1:
               [a1 impostaValore:[addendo1 doubleValue]];
               break;
          case 2:
               [a2 impostaValore:[addendo2 doubleValue]];
               break;
```

```
    }
}

- (void)aggiornaSomma:(NSNotification *)n
{
    if(kDebug)
        NSLog(@"abbiamo ricevuto la notifica
                                    che dobbiamo aggiornare la somma");
    [totale setDoubleValue:([a1 valore]+[a2 valore])];
}

@end
```

Salvate anche questo file, poi compilate ed eseguite. E godetevi lo spettacolo. Tra le altre cose, provate a vedere che cosa succede se modificate ad esempio il primo addendo ma poi premete il pulsante *Imposta* di fianco al secondo addendo (e viceversa). Divertitevi a verificare che le somme siano corrette. Ora rilassatevi, perché dobbiamo discutere di un sacco di cose nuove.

Partiamo dalla classe `Addendo`. Come vedete è di una semplicità estrema, consta di una sola variabile di tipo `double`, e ci sono i metodi accessori in lettura e scrittura per tale variabile. Ma nel metodo `impostaValore:` succede una cosa interessante. Subito dopo che `valore` è stata modificata, chiediamo al sistema che si occupa di gestire le notifiche (`[NSNotificationCenter defaultCenter]`) di sollevare una notifica per noi. Le notifiche sono identificate per nome (nient'altro che un oggetto di classe `NSString` o una costante di tipo `@""`, noi abbiamo scelto questa seconda possibilità grazie alla costante `kAddendoModificatoNotif` definita in *CostantiGenerali.h*); tale nome è il primo argomento che passiamo a `postNotificationName:`. Il secondo argomento, quello di `object:`, è l'oggetto che sta sollevando la notifica (`self`, nel nostro caso); infine il terzo argomento, quello di `userInfo:`, è come al solito un oggetto a nostra scelta, ma obbligatoriamente di classe `NSDictionary` (quindi un *dizionario*, non ce ne siamo ancora occupati), che potrà contenere tutto ciò che vogliamo, e che sarà messo a disposizione di chi riceverà la notifica, chiunque esso sia. Siccome noi nel nostro esempio non abbiamo grandi cose da comunicare, passiamo `nil` a questo argomento. Congratulazioni! Avete appena imparato come si fa a sollevare una notifica. Scegliete un nome per essa, chiedete a `[NSNotificationCenter defaultCenter]` di sollevarla per voi, specificando il nome della notifica, chi siete e se volete allegare un oggetto. Punto. Fine del film.

Naturalmente questa è solo metà della storia. In genere, se si solleva una notifica, è perché ci si aspetta che ci sia qualcuno da qualche altra parte che sia in ascolto, che sia interessato a quello che stiamo comunicando a tutti gli oggetti interessati della nostra applicazione. Sì, ma come fa un oggetto ad *ascoltare* una notifica, ad essere consapevole del fatto che una notifica di suo interesse è stata sollevata? Lo vediamo subito, esaminando la classe `Controller`.

`Controller` inizia con un metodo `init`. Fino ad ora l'avevamo sempre allegramente omesso, perché negli esempi precedenti la classe `Controller` non aveva bisogno di allocare ed inizializzare nessun oggetto. Infatti, gli outlet, pur essendo oggetti, sono allocati e inizializzati automaticamente dal sistema operativo. In questo esempio, invece, il nostro oggetto di controllo ha a che fare con due oggetti che non sono outlet, ma appartengono ad una classe definita da noi, `Addendo`. Essi allora vanno allocati ed inizializzati, ovviamente nel metodo `init`. Siamo così costretti a sovrascrivere il metodo `init` della classe genitrice di `Controller` (`NSObject`), implementandolo noi in prima persona, e seguendo la solita vecchia prassi già imparata nel secondo volume di questa trilogia: si chiama il metodo `init` della classe genitrice, si allocano ed inizializzano tutti gli oggetti necessari, si restituisce `self`. Poi, nel metodo `dealloc` (che qui siamo nuovamente costretti ad implementare), mandiamo un bel messaggio `release` agli oggetti che abbiamo allocato nel metodo `init` (`a1` e `a2`, nel nostro esempio).

Ma la storia non è finita. Che cosa sono quelle istruzioni misteriose che riguardano il

nostro caro amico `[NSNotificationCenter defaultCenter]` che compaiono nel metodo `init`? Ecco, quelle istruzioni mostrano come si fa a mettersi in ascolto di una notifica. È un po' come abbonarsi ad una rivista che ci interessa; tutte le volte che un nuovo numero è fresco di stampa, l'editore ne consegna alle Poste una copia per noi, e il postino ce la consegna più presto che può. Con le notifiche è la stessa cosa. Il nostro oggetto di controllo sta dicendo a `[NSNotificationCenter defaultCenter]` di essere fortemente interessato alla notifica di nome `@"NSApplicationDidFinishLaunchingNotification"` (vedremo tra poco che cos'è), e se per cortesia tutte le volte che il mittente specificato ad argomento di `object:` ne emette una di chiamare il metodo specificato ad argomento di `selector:`. Abbiamo passato `nil` ad argomento di `object:`, ad indicare il fatto che non ci interessa *chi* ha sollevato la notifica; chiunque esso sia, noi siamo interessati alla notifica che riporta questo nome. Similmente, avremmo potuto passare un oggetto vero e proprio (quindi non `nil`) ad argomento di `object:` e passare `nil` come primo argomento (quello col nome della notifica), ad indicare che siamo interessati a tutte le notifice, non importa quali, purché siano emesse dall'oggetto specificato (un po' come abbonarsi a tutte le riviste, non importa quali, pubblicate da un certo editore). Il nome della notifica e l'argomento di `object:` non possono essere entrambi `nil` (ma possono essere entrambi diversi da `nil`).

Allora, che cos'è questa strana e per noi nuova notifica denominata `@"NSApplicationDidFinishLaunchingNotification"`? Osservate che non è quella che sollevano gli oggetti di classe `Addendo`, perché ha un altro nome. È una notifica sollevata dal sistema operativo, o da Cocoa se preferite (non ci interessa più di tanto chi di preciso solleva la notifica). Cocoa comunica al nostro programma, mediante notifiche, un sacco di cose interessanti. Ad esempio gli oggetti di classe `NSApplication`, e il nostro *Esempio5*, che è un'applicazione Cocoa (ricordate? L'abbiamo creato tale creando un nuovo progetto con XCode), lo è (sì: anche le applicazioni intere sono oggetti, di classe `NSApplication` per l'appunto), sollevano un sacco di notifiche interessanti. Una è per l'appunto questa qua. Essa viene sollevata per segnalare a chiunque (qualunque oggetto dell'applicazione) sia interessato che l'applicazione ha finito di caricarsi da disco ed ha finalmente iniziato ad essere eseguita, è entrata in quello che si chiama il suo *loop degli eventi* (ovvero si è conclusa tutta la parte iniziale di inizializzazione e si è messa pazientemente in attesa che l'utente impartisca qualche comando con mouse e tastiera). Mettersi in ascolto di questa notifica è un modo per eseguire alcune operazioni che devono essere effettuate prima che l'utente inizi ad interagire con l'applicazione, ma che sarebbe troppo presto fare nel metodo di inizializzazione dell'oggetto di controllo. È quindi qualche cosa di molto simile all'implementare il metodo `awakeFromNib`. Tuttavia, `awakeFromNib` (se implementato) dell'oggetto di controllo viene chiamato *prima* che venga sollevata la notifica di cui stiamo parlando. Non conosco una regola semplice ed universale per decidere se una certa operazione che va eseguita all'inizio possa essere messa nel metodo `init`, o vada posticipata ad `awakeFromNib` o addirittura al ricevimento di questa notifica. Comunque in generale è bene riservare al metodo `init` l'allocazione e inizializzazione degli oggetti e tutt'al più la registrazione alle notifiche sollevate da `NSApplication`, e affidare a `awakeFromNib` dell'oggetto di controllo e al metodo che risponde alla notifica `@"NSApplicationDidFinishLaunchingNotification"` tutto il resto. Nel dubbio, provate.

Va beh, detto ciò, com'è fatto il metodo che risponde ad una notifica? È molto semplice: restituisce `void` ed accetta come unico argomento un oggetto di classe `NSNotification` (che guarda caso contiene proprio la notifica che è stata solelvata e catturata). `impostazioniPreliminari:` è il metodo che viene chiamato quando la notifica sollevata da `NSApplication` viene ricevuta. Noi non ne abbiamo bisogno, ma possiamo recuperare (attraverso l'argomento `n` passato al metodo) il nome della notifica che abbiamo catturato chiamando `[n name]`, l'oggetto che ha sollevato la notifica chiamando `[n object]` e il famoso *dizionario* contenuto in `userInfo` (noi avevamo passato `nil`) chiamando `[n userInfo]`. Noi, nel metodo `impostazioniPreliminari:`, non facciamo altro che registrare il nostro oggetto di controllo (`self`) come osservatore di un'*altra* notifica, quella che si chiama `kAddendoModifcatoNotif` (guarda caso quella sollevata dagli oggetti di classe `Addendo`), chiedendo che venga chiamato il metodo `aggiornaSomma:`. Come mai facciamo qui tutto

questo e non ad esempio in `init`? Ma perché se no non avrei avuto occasione di informarvi del fatto che le nostre applicazioni possono ricevere non solo le notifiche esplicitamente sollevate dagli oggetti da noi creati, ma anche quelle che Cocoa in una maniera o nell'altra solleva per informarci di un sacco di cose interessanti che avvengono nel sistema e nella nostra applicazione (come dirci che ha finito di caricarsi ed è entrata nel suo loop degli eventi).

Il metodo `impostaAddendo:` usa il *tag* associato ad ognuno dei due pulsanti per stabilire quale addendo aggiornare. La chiamata al metodo `impostaValore:` di uno degli addendi comporta che esso sollevi la notifica `kAddendoModificatoNotif`, che a sua volta verrà intercettata dal nostro oggetto di controllo (che si è registrato come osservatore per questa notifica) e il metodo `aggiornaSomma:` verrà chiamato, che ovviamente non fa nient'altro che sommare il valore memorizzato nei due addendi e riportarlo nel campo di testo statico contenente il totale.

È bene fare ancora qualche osservazione conclusiva, prima di passare ad occuparci di altro. Innanzitutto notate che abbiamo trovato un modo incredibilmente complicato di fare le somme. Il nostro *Esempio1* era infinitamente più semplice, manipolava direttamente i dati e non c'era bisogno di tutto questo meccanismo delle notifiche. Verissimo, ma le notifiche, non solo quelle sollevate dal sistema operativo a nostro uso e consumo, sono un sistema estremamente utile per far comunicare gli oggetti (qui, obiettivamente, per mantenere l'esempio semplice l'abbiamo usato un po' fuori luogo, ma il fine era la didattica, come sapete). Infatti, la comunicazione diretta tra oggetti comporta che ci sia un mittente e *un solo* destinatario, e che i due *si conoscano*: il mittente deve annoverare in qualche maniera tra le sue variabili locali un puntatore all'oggetto destinatario. La notifica elimina entrambi i problemi: il mittente può comunicare con *molti* destinatari, perché un numero arbitrario di oggetti può registarsi come osservatore di una certa notifica, e non è tenuto a conoscerli, non è nemmeno tenuto a preoccuparsi se ci sia qualcuno in ascolto della notifica, essa viene sollevata, poi chi è interessato la ascolta, se a nessuno interessa, pazienza. Il rovescio della medaglia è una minore velocità di trasmissione del messaggio, la notifica viaggia più lentamente di una comunicazione diretta tra oggetti.

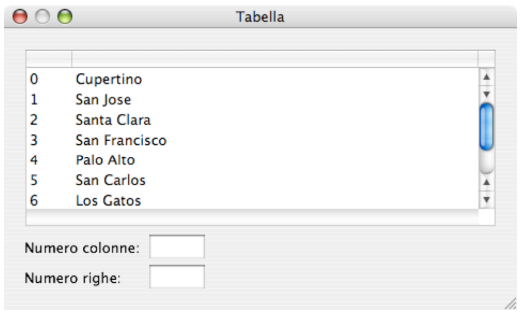
Un uso tipico delle notifiche è questo: l'oggetto di controllo dell'applicazione comunica direttamente con gli oggetti definiti tra le sue variabili locali; tuttavia essi in genere non dispongono di un puntatore all'oggetto di controllo. Se all'interno di uno di questi oggetti avviene un cambiamento di cui l'oggetto di controllo non è consapevole (per via di un'interazione diretta da parte dell'utente, di un timer o di altro) ma ne deve essere informato, l'oggetto modificato solleva una notifica che l'oggetto di controllo intercetta, ed agisce di conseguenza. Un altro uso tipico delle notifiche è quello di far comunicare tra di loro oggetti tutti singolarmente noti all'oggetto di controllo, ma ignoti l'uno all'altro. Nel nostro *Esempio5*, come avremmo fatto a far sapere al secondo addendo che è avvenuto un cambiamento del valore del primo addendo? Non ce ne frega niente, è vero, è una cosa inutile, ma fate finta che per qualche ragione il secondo addendo abbia necessità di sapere quando il valore del primo addendo cambia, e viceversa naturalmente. Come si fa? I due addendi *non si conoscono*, non hanno l'uno il puntatore dell'altro. Certo, il primo addendo potrebbe sollevare una notifica, l'oggetto di controllo la intercetta, scopre analizzando il mittente della notifica stessa che esso è il primo addendo, e comunica allora al secondo addendo il cambiamento avvenuto. Complicato, e fonte di un sacco di guai se dovessimo espandere la nostra applicazione così da consentire un numero arbitrario di addendi. Molto più comodo se il primo addendo solleva la notifica, e *tutti gli addendi* la ricevono, anche il primo che l'ha sollevata (e che potrà ignorarla grazie al fatto che troverà, come mittente della notifica ricevuta, sé stesso).

Ma ora basta con le chiacchiere. I *delegati*, con le loro interessanti relazioni con le *notifiche*, ci stanno aspettando nel prossimo capitolo.

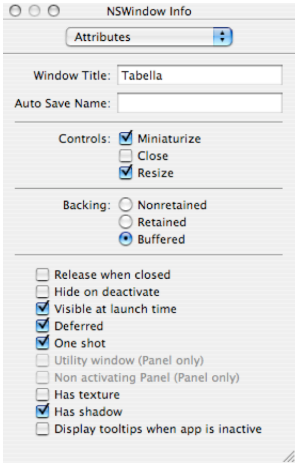
7. Delegati

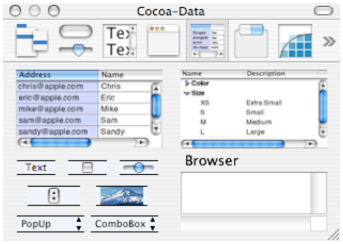
Avete presente che cos'è un delegato? È qualcuno che mandate in vece vostra da qualche parte, per fare qualche cosa che non sapete o non volete o non potete fare. Ecco, in Cocoa/Objective-C un *delegato* è esattamente la stessa cosa. È un oggetto che implementa uno o più metodi di un'altra classe, metodi che gli oggetti di quella classe non possono implementare; non si tratta di pigrizia da parte loro, si tratta di fornire un sistema il più possibile versatile e configurabile da parte del programmatore.

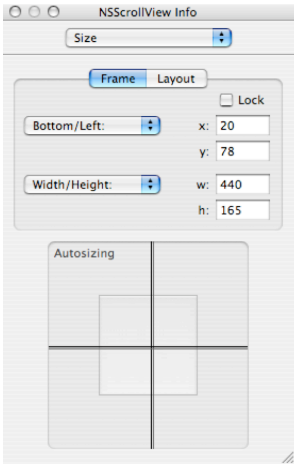
Molte classi Cocoa offrono ai loro delegati la possibilità di implementare dei metodi, ma si tratta per lo più di cose opzionali. Esistono però alcune eccezioni, classi che richiedono necessariamente un delegato (o una sua forma particolare, chiamata *data source*), e che questi implementi necessariamente alcuni metodi. In questo capitolo vedremo un paio di esempi, uno per tipo; ma, come tradizione, non ci occuperemo solo di delegati, dal momento che discuteremo anche di finestre ridimensionabili e di tabelle (classe `NSTableView`).



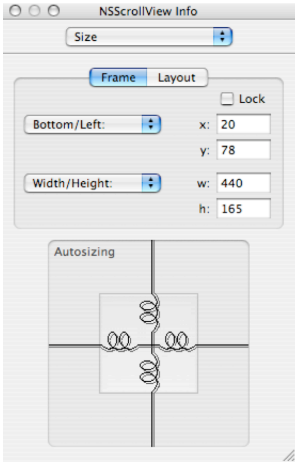
Procediamo allora come al solito, chiudendo tutti i documenti aperti in InterfaceBuilder e XCode, e creando in quest'ultimo un nuovo progetto, di tipo *Cocoa application*, che chiameremo *Esempio6*. Poi, come sempre, apriamo il suo file di risorse *MainMenu.nib* ed editiamo, in InterfaceBuilder, la finestra principale di modo che appaia come indicato in figura. Notate chel'elemento principale dell'interfaccia è una *tabella*, ottenuta trascinando l'oggetto di classe `NSTableView` che si trova nella finestrella con i vari elementi di interfaccia, come mostrato nella figura subito qui sotto (è l'oggetto in alto a sinistra). Date alla finestra le proprietà qui mostrate nella sua finestra di informazioni. Notate che, a differenza del solito, questa volta abbiamo lasciato selezionata la casellina che consente di ridimensionare la finestra. Questa piccola modifica richiede però non pochi interventi da parte nostra per essere pienamente supportata. Abbiamo infatti bisogno di precisare come si devono comportare i vari elementi di interfaccia utente, uno per uno, quando la finestra si ridimensiona: devono ridimensionarsi pure loro? Devono restare ancorati ad uno o più margini della finestra? E come? Facciamo un passo per volta. Selezionate con un click del mouse la tabella all'interno della finestra principale della nostra applicazione, poi, dalla finestrella di informazioni, selezionate la voce *Size* dal menu a comparsa. Dovreste vedere qualche cosa di simile a quanto mostrato qui in figura. All'interno del riquadro denominato *Autosizing* potete osservare un riquadro interno, e una croce fatta da linee orizzontali e verticali. Orbene, il riquadro esterno rappresenta la finestra dell'applicazione, il riquadro interno rappresenta l'oggetto che abbiamo selezionato (la tabella, in questo caso). Le linee orizzontali e verticali indicano le posizioni relative dell'oggetto selezionato rispetto all'oggetto esterno (quindi della tabella rispetto alla finestra). Come? Provate a cliccare sulle linee orizzontali e verticali: che cosa succede? Cambiano aspetto, diventando delle "molle". Cliccate ancora su di esse per farle tornare

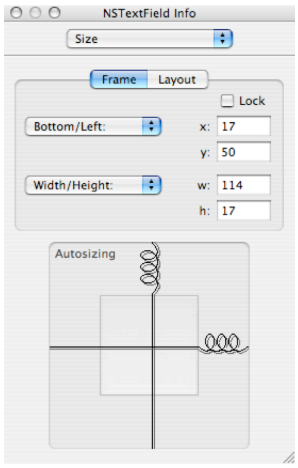




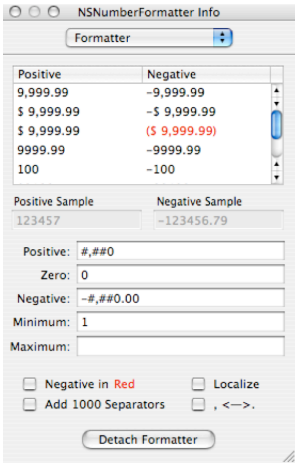


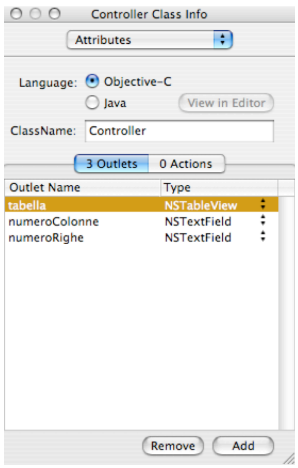
simili a delle “sbarre”. Osservate che potete cambiare separatamente le linee orizzontali e verticali all’interno del riquadro che rappresenta la tabella, e tra questo e il riquadro esterno che rappresenta la finestra. Fate allora sì che la configurazione applicata per la tabella sia quella qui mostrata in figura. Che cosa indica? Le “molle” all’interno del riquadro interno indicano che esso non ha una dimensione fissa, ma si può ridimensionare, sia in orizzontale che in verticale. Viceversa, le “sbarre” all’esterno indicano che l’oggetto selezionato (sempre la nostra tabella) mantiene una distanza fissa, costante, da tutti e quattro i bordi della finestra. Come dire: quando l’utente ridimensiona la finestra, la tabella mantiene i suoi bordi sempre alla stessa distanza dal bordo della finestra, ridimensionando di conseguenza il suo contenuto.





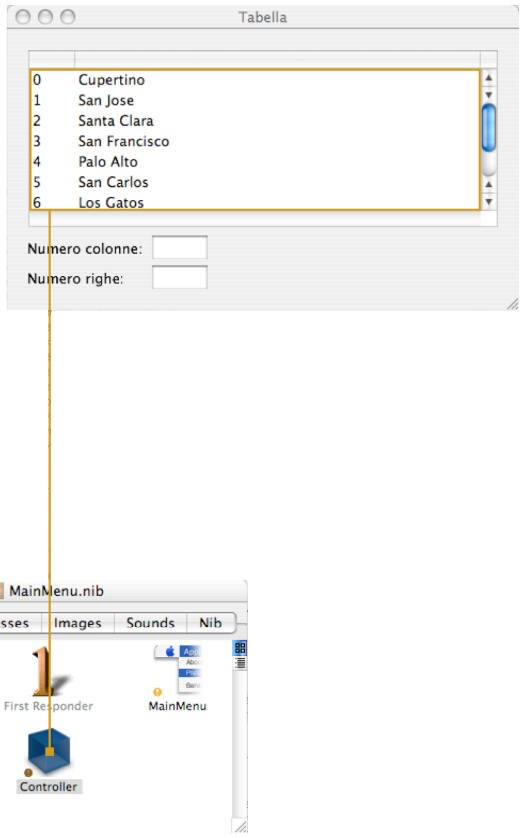
Diverso è il discorso che vogliamo fare per *ciascuno dei quattro campi di testo*, ovvero i due campi editabili e i due campi non editabili. Per ognuno di essi, vogliamo che il pannello *Size* della finestra di informazioni sia come quello indicato in figura (dovete selezionare, uno per volta, ciascuno dei quattro campi di testo nella finestra principale dell’applicazione, poi, dalla finestra di informazioni, configurare opportunamente “sbarre” e “molle” per il campo di testo selezionato). Le sbarre all’interno del riquadro interno indicano che il campo di resto non deve ridimensionarsi: le sue dimensioni orizzontali e verticali sono sempre le stesse indipendentemente da quanto sia grande la finestra. Così pure la distanza del campo di testo dal bordo sinistro e dal bordo inferiore della finestra. Lasciamo invece libera di variare la distanza del campo di testo dal bordo destro e dal bordo superiore (grazie alle due “molle”). Non siete sicuri di aver fatto tutto per bene? Sempre dall’interno di InterfaceBuilder digitate mela-R o, se preferite, selezionate il comando *Test Interface…* dal menu *File*, e provate a ridimensionare la finestra. I vari elementi dell’interfaccia riscalano correttamente? Uscite dalla modalità di test digitando mela-Q, e provate a modificare “sbarre” e “molle” per i vari elementi di interfaccia, provando subito dopo a vedere che cosa succede digitando mela-R e ridimensionando la finestra. Prendete dimestichezza con questa tecnica che vi permette di definire come riscala ogni oggetto e come questo si vada a posizionare rispetto ai bordi della finestra, poiché sarà importante non fare pasticci quando avrete a che fare con interfacce utente più complesse (finestre più affollate di oggetti rispetto a quella di questo esempio). Quando avete preso sufficiente confidenza con questo nuovo meccanismo, ripristinate le leggi di riscalamento della tabella e dei campi di testo alla configurazione mostrata nelle figure precedenti, quindi fate un ultimo sforzo ed assegnate ad entrambi i campi editabili di testo dei formattatori numerici, con le proprietà mostrate qui in figura. In questi campi di testo dovremo inserire quante righe e quante colonne ci siano nella nostra tabella, e naturalmente vogliamo che l’utente possa inserire solo numeri interi positivi e non nulli. Fatto tutto ciò, iniziate a salvare le modifiche, visto che non si sa mai.



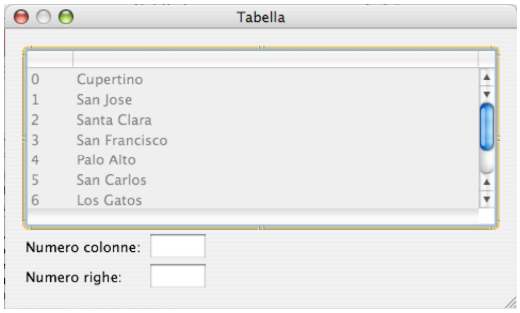


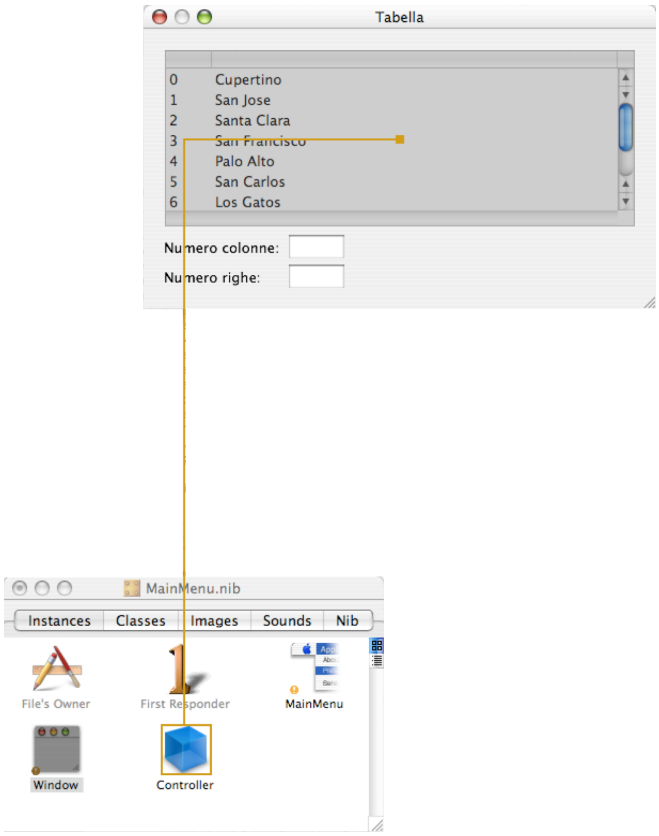
Ora naturalmente dobbiamo creare la classe di controllo, la solita `Controller` sottoclasse di `NSObject` che si può ottenere dal pannello *Classes* della finestra denominata *MainMenu.nib* e agendo sugli opportuni comandi del menu *Classes*. Assegnate tre outlet a questa classe,

come mostrato in figura. Sorprendentemente (e grazie al fatto che useremo dei *delegati*), non è necessario assegnare nessuna azione. Create i file per la classe `Controller`, quindi istanziatela. Ora effettuate i collegamenti. Collegare i due campi editabili di testo con i rispettivi outlet non rappresenta un problema. Interessante è invece il collegamento della tabella. Osservate che quando trascinate (tenendo premuto il tasto *control*) dall'icona *Controller* alla tabella, si seleziona *l'interno* della tabella. Infatti, la tabella è fatta da due oggetti uno dentro l'altro, ovvero un oggetto di classe `NSScrollView` all'esterno, che ha il compito di occuparsi dello *scrolling* di qualunque cosa ci sia all'interno (lo scrolling è quell'insieme di operazioni necessarie per far sì che possano essere implementate, e che funzionino a dovere, le barre di scorrimento orizzontale e verticale), e un oggetto di classe`NSTableView` all'interno, la nostra tabella vera e propria. Noi non abbiamo creato esplicitamente l'oggetto esterno, Cocoa l'ha fatto automaticamente per noi, sapendo che ci sarebbe servito. Ai fini pratici possiamo sostanzialmente dimenticarci della sua esistenza, si comporterà come deve in ogni circostanza senza nessun bisogno di interventi da parte nostra, tuttavia è bene essere consapevoli della sua presenza. Infatti, quando cercate di fare il collegamento dell'outlet *tabella*, potete provare a fermarvi col mouse sul bordo della tabella, ad esempio dove c'è la barra di scorrimento. Vedete che si seleziona il contorno più esterno? Sì, però così facendo non potete collegare quest'oggetto all'outlet *tabella*, perché non è di classe `NSTableView`. Selezionate quindi la parte interna quando fate il collegamento.



Un trucchetto simile è da tenere in mente nel momento in cui volete selezionare la *data source* della tabella. E che sarà mai questa *data source*? Il nome inglese vi dà già un'idea: è la *fonte dei dati*. Infatti, Cocoa sa che vogliamo maneggiare una tabella, ma naturalmente non sa che cosa vogliamo metterci dentro questa tabella, quali sono i dati che vorremo visualizzare con essa. Pertanto avrà bisogno di sapere *dove potrà andare a pescare questi dati*, ovvero quale oggetto avrà il compito di fornire alla tabella (intesa come un elemento dell'interfaccia utente) i dati da visualizzare. Nel nostro caso, tale oggetto sarà proprio quello di controllo. Dobbiamo quindi innanzitutto selezionare *l'interno* della tabella: se fate un click solo su di essa, selezionate *l'involucro*, ovvero l'oggetto di classe `NSScrollView`; ma se fate un bel doppio-click, selezionate*l'interno*, come mostrato in figura. A questo punto, tenendo premuto il tasto *control*, potete trascinare sull'icona *Controller* nella finestra *MainMenu.nib*, come mostrato in figura. Potete finalmente fare il collegamento all'outlet *dataSource* di `NSTableView`. Già, perché la nostra classe di controllo non è l'unica ad avere degli outlet, anche le classi messeci a disposizione da Cocoa ne hanno. `NSTableView`, ad esempio, ne ha vari (di cui uno si chiama





delegate, guarda un po'… ma qui useremo delegati di altre classi, non di `NSTableView`), come ad esempio *dataSource*. Questo outlet punta ad un oggetto creato da noi (pur non essendo letteralmente un *delegato*, perché non è connesso all'outlet *delegate*, assomiglia molto ad un delegato) che ha il compito di implementare alcuni metodi indispensabili al corretto funzionamento di un oggetto di classe `NSTableView`; naturalmente si tratta di metodi che Cocoa non può implementare autonomamente, dal momento che, come dicevamo, hanno il compito di informare il sistema su quali dati debbano essere visualizzati nella tabella; questo Cocoa non lo può sapere senza il nostro aiuto.
Travolti da questa valanga di novità, salvate le modifiche in InterfaceBuilder e tornate ad XCode, dove tanto per cambiare sposterete i file *Controller.h* e *Controller.m* nella cartellina *Classes* all'interno della finestra del progetto.
Quindi aprite con un doppio click il file di interfaccia *Controller.h* e modificatelo in modo che appaia così:

```
/* Controller */

#import <Cocoa/Cocoa.h>

#define   kDebug              YES

@interface Controller : NSObject
{
    IBOutlet NSTextField *numeroColonne;
    IBOutlet NSTextField *numeroRighe;
    IBOutlet NSTableView *tabella;

     int            numRighe,numColonne;
}

- (void)creaColonna:(int)indice;
- (void)cancellaColonne;

// metodi implementati dal delegato di NSApplication
- (void)applicationDidFinishLaunching:(NSNotification *)n;

// metodi implementati dal dataSource di NSTableView
- (int)numberOfRowsInTableView:(NSTableView *)aTableView;
- (id)tableView:(NSTableView *)aTableView
                    objectValueForTableColumn:(NSTableColumn *)aTableColumn
```

```
                    row:(int)rowIndex;
- (void)tableView:(NSTableView *)aTableView
                    setObjectValue:(id)anObject
                    forTableColumn:(NSTableColumn *)aTableColumn
                    row:(int)rowIndex;

// metodi implementati dal delegato di NSTextField
- (void)controlTextDidChange:(NSNotification *)aNotification;

@end
```

Salvatelo, poi editate il file di implementazione *Controller.m*:

```
#import "Controller.h"

@implementation Controller

- (id)init
{
     [super init];
     [[NSApplication sharedApplication] setDelegate:self];
     numRighe=5;
     numColonne=5;
     return self;
}

- (void)applicationDidFinishLaunching:(NSNotification *)n
{
     int            i;

     if(kDebug)
          NSLog(@"Stiamo eseguendo il delegato di NSApplication");

     [numeroColonne setDelegate:self];
     [numeroRighe setDelegate:self];
     [self cancellaColonne];
     for(i=1;i<=numColonne;i++)
          [self creaColonna:i];
     [tabella reloadData];
     [numeroColonne setIntValue:numColonne];
     [numeroRighe setIntValue:numRighe];
}

- (void)creaColonna:(int)indice
{
     NSTableColumn      *tc;

     if(kDebug)
          NSLog(@"Stiamo creando la colonna di indice %d",indice);
     tc=[[NSTableColumn alloc] init];
     [[tc headerCell] setStringValue:
                         [[NSNumber numberWithInt:indice] stringValue]];
```

```
    [[tc headerCell] setTag:indice];
    [tabella addTableColumn:tc];
    [tc release];
}

- (void)cancellaColonne
{
    NSEnumerator  *en;
    id                obj;

    if(kDebug)
        NSLog(@"Stiamo cancellando tutte le colonne");
    en=[[tabella tableColumns] objectEnumerator];
    while(obj=[en nextObject])
        [tabella removeTableColumn:obj];
}

- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
    if(kDebug)
        NSLog(@"Quante righe ha la tabella? %d",numRighe);
    return numRighe;
}

- (id)tableView:(NSTableView *)aTableView
                  objectValueForTableColumn:(NSTableColumn *)aTableColumn
                  row:(int)rowIndex
{
    int           c,coordinate;
    c=[[aTableColumn headerCell] tag];
    coordinate=10*(rowIndex+1)+c;
    if(kDebug)
        NSLog(@"Coordinata della riga %d -
                          colonna %d: %d",(rowIndex+1),c,coordinate);
    return [NSNumber numberWithInt:coordinate];
}

- (void)tableView:(NSTableView *)aTableView
   setObjectValue:(id)anObject
   forTableColumn:(NSTableColumn *)aTableColumn
              row:(int)rowIndex
{
    NSLog(@"Il nostro esempio non supporta ancora l'inserimento manuale
                                              di valori nella tabella!");
}

- (void)controlTextDidChange:(NSNotification *)aNotification
{
    int           i;

    if(kDebug)
        NSLog(@"Stiamo eseguendo il delegato di NSControl");
```

```
    [self cancellaColonne];
    numColonne=[numeroColonne intValue];
    for(i=1;i<=numColonne;i++)
        [self creaColonna:i];
    numRighe=[numeroRighe intValue];
    [tabella reloadData];
}

@end
```

Salvatelo e compilate ed eseguite il programma. Divertitevi a modificare il numero di righe e il numero di colonne, notando come la tabella venga aggiornata in tempo reale. Notate che potete ridimensionare le colonne, cambiarle di ordine e "scrollare" la tabella a vostro piacimento. Avete già capito come funziona?

Andiamo con ordine, iniziando dal metodo `init`, nel quale abbiamo inserito una strana riga: `[[NSApplication sharedApplication] setDelegate:self];`. Come dicevamo nel capitolo scorso, ogni applicazione è in realtà un oggetto della classe `NSApplication`. Ogni applicazione conosce sé stessa, ovvero conosce *quale oggetto è*, grazie al metodo (di classe) `sharedApplication` implementato dalla classe `NSApplication`. C'è una scorciatoia, ed è la variabile globale `NSApp` definita da Cocoa esattamente come `NSApp=[NSApplication sharedApplication]`. Si dà il caso, che un oggetto di classe `NSApplication` gradisca molto la presenza di un delegato, che viene impostato grazie al metodo `setDelegate:`. Nel metodo `init`, quindi, stiamo dicendo che l'oggetto di controllo è il delegato di `NSApplication` per quanto riguarda l'oggetto che a tutti gli effetti *è* la nostra applicazione. Sì, ma perché? Consultando la documentazione relativa alla classe `NSApplication` si scopre una cosa molto interessante:*tutte* le notifiche che `NSApplication` può mandare (una delle quali abbiamo intercettato nell'*Esempio5*) hanno un corrispettivo *metodo del delegato*, ovvero il programmatore può scegliere (qualora sia interessato) di richiedere che gli venga segnalata una notifica, oppure designare un oggetto a propria scelta come delegato di `NSApplication` e implementare in questo un metodo ben preciso che riceve, come argomento, esattamente la stessa notifica che avremmo potuto intercettare. Non è chiaro? Spieghiamo meglio: nell'*Esempio5* registravamo il nostro oggetto di controllo come interessato alla notifica `@"NSApplicationDidFinishLaunchingNotification"`, cosicché potessimo essere informati del completamento del caricamento e dell'inizializzazione di tutto ciò che riguardava la nostra applicazione. Qui, nell'*Esempio6*, facciamo la stessa cosa, ma anziché intercettare la notifica decidiamo di registrare il nostro oggetto di controllo come *delegato* di `NSApplication`, implementare il metodo `applicationDidFinishLaunching:`, che è tra quelli previsti per l'implementazione da parte del delegato, cosicché tale metodo venga automaticamente chiamato nel momento in cui l'applicazione è stata interamente caricata ed inizializzata; non abbiamo avuto bisogno di registrarci presso `[NSNotificationCenter defaultCenter]`, ma essendo delegati *la stessa identica notifica* che ci veniva recapitata nell'*Esempio5* ci viene ora recapitata nell'*Esempio6* ad argomento di `applicationDidFinishLaunching:`. Perché mai questa dualità? Perché posso fare la stessa identica cosa in due modi diversi? Azzardo una risposta, una mia interpretazione. Se siete interessati ad una sola informazione (come ad esempio sapere quando l'applicazione ha finito di caricarsi), usare il meccanismo delle notifiche o quello dei delegati non fa alcuna differenza. Tuttavia, se siete interessati ad esempio a 4 diverse informazioni (tra quelle che la classe `NSApplication` fornisce tramite il meccanismo delle notifiche, naturalmente), vi tocca registrarvi come osservatori per 4 differenti notifiche e implementare 4 metodi per gestirle, oppure registrarvi come osservatori di *tutte* le notifiche che giungono da `NSApp` e poi scegliere, in un metodo opportuno, quali siano di vostro interesse. Se invece decidete di affidarvi al meccanismo dei delegati, vi basta dire nel metodo `init` che il vostro oggetto è delegato di `NSApp`, poi implementare *i soli metodi del delegato* che vi interessano. È quindi una questione di comodità.

Ma c'è di più: se andate a vedere bene, le notifiche non possono fare altro che *notificare* che qualche cosa è avvenuto o sta per avvenire. I metodi del delegato di `NSApp`, invece, sono di più, perché oltre a tutti quelli strettamente imparentati con le notifiche, ce ne sono altri che non si limitano a notificare, ma *chiedono il permesso* di effettuare una certa operazione (ad esempio uscire dall'applicazione stessa) oppure chiedono al delegato di eseguire una certa operazione (ad esempio aprire un file sulla cui icona l'utente ha fatto doppio-click nel Finder). Insomma, quello dei delegati è un meccanismo più complesso e più articolato, che permette all'applicazione e al sistema di interagire più strettamente: ad esempio il delegato di `NSApp` può impedire che l'applicazione venga chiusa (anche se l'utente ha scelto il comando *Quit* dal menu dell'applicazione) se ci sono dei file ancora da salvare; una notifica può solo informare un oggetto che l'applicazione sta per chiudersi, un delegato, invece, può impedire che questo avvenga.

Questo discorso è in realtà più generale di quanto fatto fin qui: abbiamo ristretto il nostro interesse al caso della classe `NSApplication`, delle sue notifiche e dei suoi metodi del delegato, ma lo stesso discorso vale per tutte quelle classi di Cocoa che supportano notifiche: esse avranno anche metodi del delegato equivalenti, e potrebbero avere metodi addizionali per consentire l'implementazione di funzioni più complesse.

Evviva i delegati, quindi. Ma le notifiche riguadagnano un punto su un terreno diverso: un oggetto può avere *un solo delegato*, ma *molti altri oggetti* possono essere in ascolto delle sue notifiche. Quindi ascoltate le notifiche, e se proprio non ne potete fare a meno siate voi il delegato dell'oggetto da cui volete ricevere informazioni, ma ricordatevi che non può esistere più di un delegato per ogni oggetto (ma un oggetto può fare da delegato per molti oggetti diversi).

Appurato ciò, ora sappiamo che il metodo `applicationDidFinishLaunching:` viene chiamato automaticamente non appena l'applicazione ha terminato di caricarsi ed inizializzarsi, grazie al fatto che l'oggetto di controllo è diventato il delegato di `NSApp`. In questo metodo iniziamo col rendere l'oggetto di controllo il delegato di due altri oggetti, ovvero i due campi di testo editabile. Questo perché la classe `NSTextField` consente al delegato di implementare vari metodi, compresi alcuni che informano il delegato del fatto che il testo contenuto nel campo è stato modificato (altri, invece, chiedono il permesso di modificare il testo). Questo è estremamente comodo, perché ci consente di monitorare con continuità in tempo reale il contenuto dei due campi di testo contenenti il numero di righe e di colonne, senza bisogno di aggiungere all'interfaccia utente un pulsante che forzi il riaggiornamento della tabella dopo che l'utente l'ha premuto. Tra l'altro, nel nostro *Esempio4* avremmo potuto eliminare elegantemente quel fastidioso problema che impediva al messaggio di essere visualizzato nel pannello se l'utente non premeva "a-capo" al termine dell'inserimento: se avessimo usato l'oggetto di controllo come delegato del campo di testo, avremmo potuto renderci conto istantaneamente di qualsiasi modifica del testo del messaggio. Torneremo su questo aspetto tra poco.

Notiamo poi che nel metodo `applicationDidFinishLaunching:` facciamo un po' d'ordine: la tabella che abbiamo creato con InterfaceBuilder presumibilmente contiene già delle colonne, ma quante? Tagliamo la testa al toro, leviamo tutte le colonne eventualmente presenti (`[tabella cancellaColonne]`) e creiamone di nuove (5, in base al valore iniziale che abbiamo voluto dare alla variabile `numColonne`), chiamando ripetutamente il metodo `creaColonna:`. Per ragioni di semplicità, in questo *Esempio6* scegliamo di distinguere una colonna dall'altra mediante un numero identificativo, un *indice*, che passiamo ad argomento di `creaColonna:`. Scriviamo poi nei due campi di testo l'attuale numero di righe e di colonne.

`creaColonna:` fa il vero e proprio lavoro sporco di aggiungere una colonna alla tabella. Innanzitutto crea un nuovo oggetto di classe `NSTableColumn`, la classe a cui appartengono le colonne di una tabella. Gli oggetti di classe `NSTableColumn` contengono una `headerCell`, che non è nient'altro che un oggetto sottoclasse di `NSControl`, al quale possiamo assegnare un titolo (un valore stringa) e un *tag* con i modi che abbiamo già visto in passato. La `headerCell` è l'intestazione della colonna, quella casellina che compare in alto e sulla quale potete cliccare per spostare la colonna, ridimensionarla o selezionarla. In essa comparirà il titolo che assegnamo alla colonna (il suo numero d'indice); dopo aver impostato anche il *tag*, il metodo `addTableColumn:` di `NSTableView` consente di aggiungere la colonna alla tabella. È da notare

che *non abbiamo ancora detto quale sia il contenuto della tabella*. Esso infatti *non è memorizzato nell'oggetto di classe* `NSTableView`, ma nell'oggetto che ne è il `dataSource`. Quindi, tutte le volte che aggiungiamo colonne ad un oggetto di classe `NSTableView`, non è a lui che dobbiamo dire che cosa metterci dentro; sarà l'oggetto che ne è il `dataSource` a doversi preoccupare di contenere tutte le informazioni necessarie per riempire di roba dotata di un minimo di senso tutte le colonne esistenti nella tabella.

Che cosa ci sia nella tabella e come sia possibile modificarlo è invece oggetto di interesse per i tre metodi che *obbligatoriamente* il `dataSource` di un oggetto di classe `NSTableView` *deve* implementare. Il primo di essi è `numberOfRowsInTableView:`: esso deve restituire un numero intero (positivo o nullo) che indica quante righe abbia la tabella (oggetto di classe `NSTableView`) passata ad argomento; questo perché l'oggetto definito come `dataSource` di una certa tabella, può esserlo anche per altre tabelle, ma `numberOfRowsInTableView:` è uno solo per tutte, e bisogna poter capire da quale tabella viene la richiesta di sapere quante righe abbia. Noi abbiamo una tabella sola, quindi senza fare ulteriori controlli rispondiamo dicendo che essa contiene `numRighe` righe. Il secondo di questi tre metodi obbligatori è `tableView:objectValueForTableColumn:row:`. Esso ha il dovere di restituire un oggetto *qualunque* (infatti il tipo restituito è `id`), che non è nient'altro che ciò che è contenuto alla riga di indice passato ad ultimo argomento della colonna passata come secondo argomento della tabella passata come primo argomento. Notate che la riga è identificata per numero (la prima riga è la numero 0), mentre la colonna è identificata mediante un oggetto di classe `NSTableColumn`, perché l'utente potrebbe aver modificato l'ordine delle colonne e quindi riferirsi ad esse mediante un semplice numero d'ordine non andrebbe bene. Invece, avendo a disposizione l'intero oggetto `NSTableColumn`, possiamo risalire (ad esempio andando a vedere il *tag* della sua `headerCell`, come facciamo noi) di quale colonna realmente si tratti. L'oggetto qualunque che viene restituito da questo metodo è ovviamente l'oggetto che è contenuto in quella casella lì. Noi, per semplicità in questo esempio che contiene già abbastanza cose nuove, abbiamo scelto di mettere come contenuto di ogni casella della tabella un semplice numero (oggetto di classe `NSNumber`), ottenuto dalle "coordinate" della cella stessa, ovvero dal numero di riga e dal numero della colonna (ovvero il valore del *tag* della sua `headerCell`): osservate che la prima riga della prima colonna ha "coordinate" 11, la seconda riga della prima colonna ha coordinate 21, la prima riga della seconda colonna ha coordinate 12 e così via. Osservate anche che se scambiate di posto due colonne, la numerazione resta consistente col "numero" della colonna riportato nel suo titolo (che poi è lo stesso del *tag* della `headerCell`). Il terzo metodo implementato dal `dataSource` è `tableView:setObjectValue:forTableColumn:row:`. Esso viene chiamato tutte le volte che l'utente fa un doppio-click su una casella della tabella (provate anche voi!), cosa che consente all'utente di modificare il contenuto della casella stessa (a meno che non abbiate vietato questo comportamento chiamando il metodo `setEditable:` con argomento `NO` dell'oggetto di classe `NSTableColumn` a cui appartiene la casella); questo metodo informa il `dataSource` di quale sia il nuovo contenuto (il secondo argomento) della casella identificata come al solito grazie a riga, colonna e tabella. In questo esempio ignoriamo semplicemente una simile azione da parte dell'utente. In effetti, avremmo potuto aggiungere la riga `[tc setEditable:NO];` *prima* di `[tabella addTableColumn:tc];` nel metodo `creaColonna:` e quest'ultimo metodo implementato dal `dataSource` non sarebbe mai stato chiamato (provate, provate: aggiungete questa riga, ricompilate ed eseguite: che succede se fate doppio-click su una casella?).

Ecco quindi come funziona il `dataSource` di una tabella: tutte le volte che ce n'è bisogno, Cocoa chiama automaticamente uno di questi tre metodi, specificando nel dettaglio riga e colonna (non per il primo, ovviamente) cosicché il `dataSource` possa comunicare il numero di righe oppure il contenuto della casella oppure impostare il nuovo contenuto di una casella. Queste informazioni non vengono memorizzate all'interno dell'oggetto di classe `NSTableView`, Cocoa chiama questi metodi per sapere *che cosa visualizzare a schermo all'interno della tabella*. Che cosa ci sia al suo interno sono fatti dell'applicazione, non di Cocoa. Quello che Cocoa vuole sapere, casella per casella *e solo quando è necessario*, è che cosa c'è dentro. Così chiamerà all'occorrenza questi metodi tutte le volte che ne avrà bisogno e nell'ordine che riterrà

più opportuno. Naturalmente un minimo di controllo ce l'ha anche il programmatore, che può chiamare il metodo `reloadData` di `NSTableView` (come facciamo all'interno del metodo `applicationDidFinishLaunching:` subito dopo aver creato le nuove colonne) obbligando così Cocoa a ridisegnare tutto il contenuto (visibile a schermo) della tabella; una chiamata a `reloadData` è indispensabile tutte le volte che il contenuto di una tabella cambia non per effetto dell'azione diretta dell'utente (che ha manualmente editato una casella facendoci doppio-click sopra), ma per via di un'operazione fatta dall'applicazione stessa (ovvero tutte le volte che il contenuto della tabella cambia senza che si passi attraverso il metodo `tableView:setObjectValue:forTableColumn:row:`).

Il metodo `cancellaColonne`, che prima abbiamo bellamente ignorato, si fa dare dalla tabella un elenco delle sue colonne (sotto forma di oggetto di classe `NSArray`) mediante il metodo `tableColumns`, quindi itera su tutte le colonne presenti nell'array (grazie al solito meccanismo dell'*enumerator*), passandole ad argomento di `removeTableColumn:`, metodo di `NSTableView` che consente, incredibile a dirsi, di rimuovere la colonna (oggetto di classe `NSTableColumn`) passata ad argomento. Ovviamente, iterando su tutte le colonne della tabella, si finisce per toglierle tutte (che poi è lo scopo di `cancellaColonne`).

Per concludere, e poi con questo capitolo così denso di novità abbiamo veramente finito, vediamo il metodo `controlTextDidChange:`. Esso, come avrete capito, è un metodo implementato dal *delegato* dei campi editabili di testo, ovvero dal nostro oggetto di controllo (dal momento che in `applicationDidFinishLaunching:` esso si era preso la briga di proclamarsi delegato dei due campi di testo). Di tutti i numerosi metodi che gli oggetti di classe `NSTextField` mettono a disposizione del delegato (nessuno, tali metodi sono in effetti messi a disposizione da `NSControl`, di cui `NSTextField` è sottoclasse, così come `NSButton` e un sacco di altre classi di altri elementi dell'interfaccia grafica), abbiamo scelto `controlTextDidChange:`, che viene chiamato tutte le volte che il testo contenuto nell'oggetto di classe `NSControl` (o sua sottoclasse, come nel caso dei campi di testo) cambia (per azione dell'utente, ma anche a seguito di un esplicito comando dato dal programma). Quindi, ogni volta che l'utente modifica il numero di righe o il numero di colonne in uno dei due campi di testo, questo metodo viene chiamato (il dettaglio delle informazioni riguardanti la causa che ha comportato la chiamata di questo metodo è contenuto nella notifica passata ad argomento); esso, nella nostra implementazione, cancella tutte le colonne presenti nella tabella e ne crea di nuove, in base ai parametri specificati dall'utente (numero di righe e di colonne). È da notare che se avessimo scelto di implementare il metodo `controlTextDidEndEditing:` saremmo stati informati del cambiamento del testo contenuto nel campo solo *al termine* dell'editing da parte dell'utente (quando clicca da un'altra parte, ad esempio, facendo perdere il *focus* al campo di testo), e non in tempo reale. Implementando ad esempio il metodo `control:textShouldEndEditing:` invece avremmo potuto verificare, ad editing avvenuto, se il contenuto fosse di nostro gradimento (ad esempio potremmo verificare che l'utente non scelga tabelle con 13 righe o 13 colonne perché porterebbe sfortuna), restituire un valore logico falso (`NO`) qualora volessimo annullare (impedire) le modifiche fatte dall'utente (costringendolo a scegliere un altro numero di righe o di colonne), oppure vero (`YES`) per acconsentire al cambiamento. Volete un esempio? Aggiungete al file *Controller.m*, al fondo ma *prima* della direttiva `@end`, questo metodo:

```
- (BOOL)control:(NSControl *)aControl
                  textShouldEndEditing:(NSNotification *)aNotification
{
     if([numeroColonne intValue]==13 || [numeroRighe intValue]==13)
          return NO;
     else
          return YES;
}
```

Inserite ovviamente anche il suo prototipo nel file di interfaccia *Controller.h*. Ricompilate

ed eseguite nuovamente il programma. Provate a scrivere 13 come numero di righe o di colonne. Che cosa succede? La tabella assume sì 13 righe oppure 13 colonne, ma non potete passare ad editare l'altro campo di testo (provate a cliccarvi dentro, o a premere il tasto tabulatore), perché avete inserito un numero "proibito" di righe o di colonne, e il metodo `control:textShouldEndEditing:` implementato dal delegato restituisce `NO`. Gran cosa, i delegati, nevvero?

8. Manipolare i dati

Classi collettive

Le abbiamo già usate, già a partire dal secondo volume, quello sull'Objective-C. Sto parlando delle *classi collettive*, ovviamente. Non è il loro nome ufficiale, ma è il nome che do io loro. Si tratta di classi Cocoa che consentono di manipolare *collezioni di oggetti*, in blocco o singolarmente; sono le *array*, i *dizionari* e, da un certo punto di vista, le *stringhe* di caratteri. Esse sono disponibili in una forma *fissa* (il contenuto di queste classi è definito in fase di inizializzazione e non può più cambiare) o *mutevole* (il loro contenuto può essere modificato a piacimento). Per le array Cocoa ci mette a disposizione le classi `NSArray` (forma fissa) e `NSMutableArray` (forma mutevole), per i dizionari le classi `NSDictionary` e `NSMutableDictionary` e per le stringhe le classi `NSString` e `NSMutableString`. Non sono le uniche classi collettive fornite da Cocoa, ma sono probabilmente quelle di uso più generale, e comunque le uniche di cui ci occuperemo qui.

Tutte queste classi dispongono di metodi (sia normali che autorelease) per l'inizializzazione di nuovi oggetti a partire da altri oggetti della stessa classe, o fornendo l'elenco degli oggetti che andranno collezionati, o specificando un file o un URL da cui prelevare gli oggetti necessari. Consentono poi di accedere agli oggetti contenuti in forma collettiva o singolarmente, usando indici o chiavi di ricerca, e, nella loro forma mutevole, consentono di aggiungere e rimuovere oggetti a piacimento all'interno della collezione. Infine consentono di registrare su file o su URL quanto contenuto in esse.

Questo in generale. Vediamo qualche dettaglio in più, prima di passare al nostro nuovo esempio, che si concentrerà sui dizionari, visto che sono una novità (essendoci già occupati, anche se non in maniera sistematica, di array e di stringhe nel volume sull'Objective-C e, saltuariamente, anche in questo manuale).

Gli oggetti di classe `NSArray` contengono collezioni di altri oggetti, che possono appartenere ad una qualunque classe e non devono necessariamente appartenere tutti alla stessa classe. Gli oggetti contenuti in una `NSArray` sono rintracciabili mediante un numero d'ordine (un *indice*), e un metodo `objectEnumerator` consente di ottenere un oggetto di classe `NSEnumerator` da usare per scorrere in sequenza tutti gli oggetti contenuti nell'array. Nella sua forma *mutevole*, la classe offre metodi per aggiungere, inserire e rimuovere oggetti.

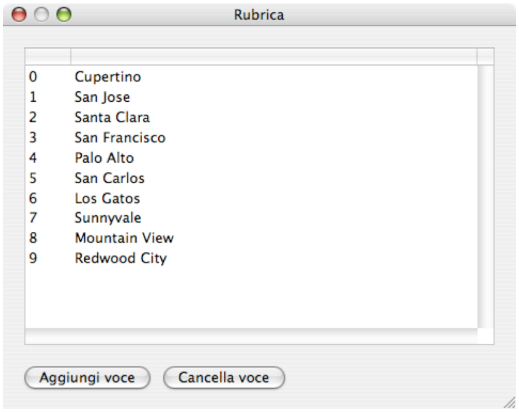
Gli oggetti di classe `NSDictionary`, o *dizionari*, sono molto simili alle array, ma gli oggetti contenuti anziché essere catalogati mediante un indice sono catalogati mediante una *chiave*. Proprio come in un dizionario possiamo cercare la *definizione* di un vocabolo usando questo come *chiave* di ricerca, in un `NSDictionary` possiamo usare una stringa (oggetto di classe `NSString`) come chiave per rintracciare un oggetto di classe qualunque associato ad essa. Se avete esperienza con altri linguaggi di programmazione, un dizionario è simile ad una *hash table* (se non sapete che cos'è una *hash table*, sappiate che è simile ad un dizionario; se pensate che sia un circolo vizioso avete ragione). Un oggetto di classe `NSDictionary` può fornire un'array (`NSArray`) di tutte le chiavi memorizzate o di tutti gli oggetti contenuti. Nella sua forma *mutevole*, la classe consente di aggiungere e rimuovere voci a piacimento.

Gli oggetti di classe `NSString` sono collezioni ordinate di *caratteri* (lettere, numeri, segni di interpunzione… caratteri, insomma). I caratteri memorizzati possono essere manipolati in blocco (la stringa nel suo complesso), o singolarmente (accedendo ad essi di nuovo mediante un indice). Gli oggetti di classe `NSString` offrono numerosi metodi per le conversioni di tipo (ottenere il valore `int` o `double` di un numero espresso sotto forma di stringa) e per la manipolazione di *path* (i percorsi dei file sul disco, con tutte le *slash* / del caso, il simbolo ~ per identificare la cartella home di un utente, le estensioni dei file ecc.). Nella loro forma mutevole, consentono di effettuare operazioni di ricerca/sostituzione di caratteri al loro interno, di aggiungere o rimuovere caratteri, prefissi, suffissi e quant'altro.

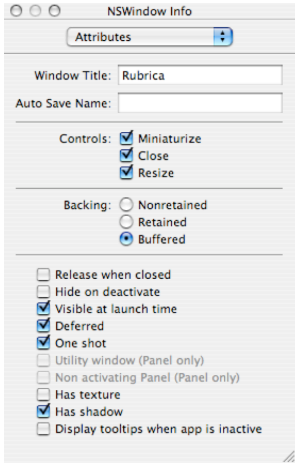
Trattare nel dettaglio ognuno dei metodi implementati da queste classi richiederebbe un volume intero. Qui abbiamo fatto questa breve introduzione generale solo per fare chiarezza sulle cose già viste in passato, e su quelle che stiamo per vedere tra poco.

File e URL

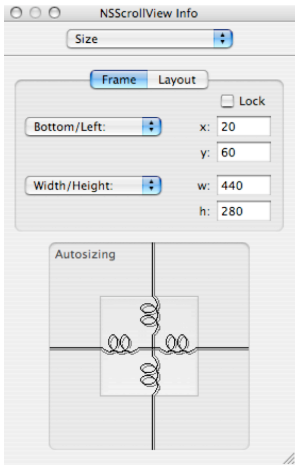
Tenetevi forte. Stiamo per realizzare l'esempio più complesso di tutto il manuale, forse l'unico che non sia totalmente inutile. In questo esempio metteremo in pratica praticamente tutto quello che abbiamo imparato fino ad ora (tranne i timer), useremo array e dizionari, e salveremo il tutto su file. Sì, ma che cosa? Una rudimentale rubrica indirizzi. Avete capito bene, faremo una rudimentale rubrica indirizzi. Organizzati in colonne (che chiameremo *categorie*: nome, cognome, residenza, indirizzo e-mail, ecc…), i dati faranno parte di una tabella editabile dall'utente e salvabile su disco. Sarà un lavoro impegnativo, abbiamo parecchia roba da fare. E allora, non indugiamo oltre. Chiudete tutti i documenti aperti con XCode e InterfaceBuilder, e da XCode create una nuova *Cocoa application*, che chiameremo *Esempio7*.

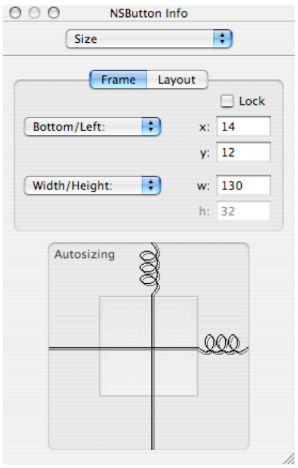


Aprite il file di risorse *MainMenu.nib* in InterfaceBuilder, e iniziate a costruire l'interfaccia della finestra principale dell'applicazione, come mostrato in figura: userete una tabella e due pulsanti; chiamerete questi ultimi *Aggiungi voce* e *Cancella voce*.

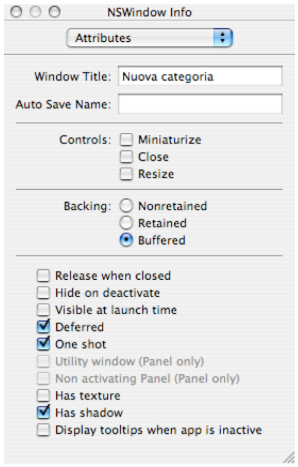


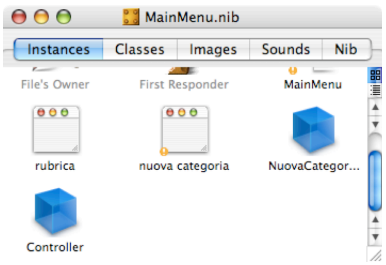
Accertatevi che la finestra di informazioni relativa alla finestra principale dell'applcazione riporti le configurazioni qui indicate in figura. Siccome abbiamo scelto di rendere la finestra ridimensionabile, impostate i criteri di ridimensionamento della tabella come indicato nella parte di sinistra della figura qui sotto, e dei due pulsanti come indicato nella parte di destra.



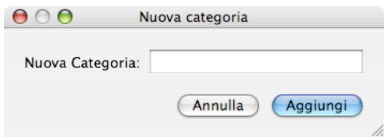


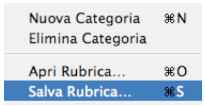
Aggiungete poi una nuova finestra alle risorse. Per dinstinguere più facilmente una finestra dall'altra anche nella finestra *MainMenu.nib*, rinominatele così come indicato in figura. Fate in modo che questa nuova finestra, che servirà all'utente per introdurre nuove categorie, appaia come mostrato in figura, e abbia le proprietà indicate nella finestrella di informazioni riportata. Osservate che vogliamo che la finestra *non* sia automaticamente visibile all'avvio del programma e che *non* sia dotata di pulsanti di minimizzazione, chiusura o ridimensionamento.





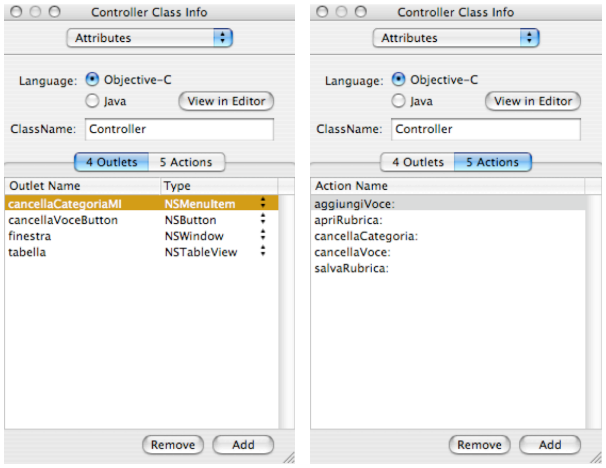
Per finire, eliminate il menu *File* dalla barra dei menu ed aggiungete

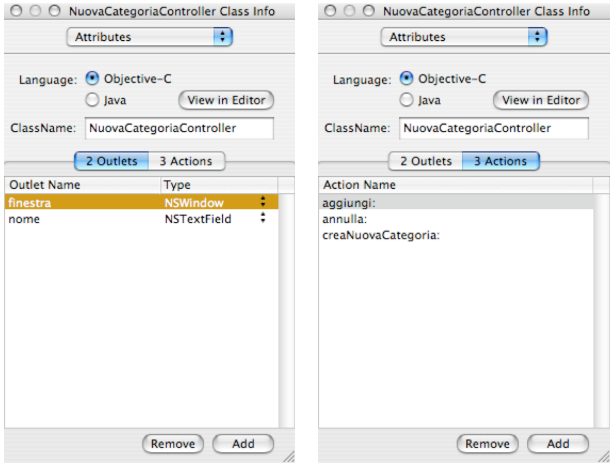




al suo posto un nuovo menu intitolato *Rubrica*, le cui voci sono quelle mostrate qui in figura. Giusto per evitare guai, iniziate a salvare le modifiche.

Ora dobbiamo occuparci delle classi di controllo. Ho detto *classi* e non *classe*, perché in questo esempio facciamo una cosa nuova: siccome abbiamo due finestre (una, quella pricipale, con la tabella, e l'altra che serve per aggiungere categorie -colonne- alla tabella), facciamo due classi di controllo, una per finestra. Dal pannello *Classes* della finestra *MainMenu.nib* subclassate quindi `NSObject`, denominate `Controller` la nuova classe, e dotatela di 4 outlet e 5 azioni, come mostrato in figura. Create i file e istanziate la classe. Collegate l'outlet *cancellaCategoriaMI* alla voce *Elimina Categoria* del menu *Rubrica*, l'outlet *CancellaVoceButton* al pulsante *Cancella voce* della finestra principale, l'outlet *finestra* alla finestra principale e l'outlet *tabella* alla tabella (ricordate, all'*interno* della tabella). Collegate poi il pulsante *Aggiungi voce* all'azione *aggiungiVoce:*, la voce *Apri Rubrica…* del menu *Rubrica* all'azione *apriRubrica:*, la voce *Elimina Categoria* del menu *Rubrica* all'azione *cancellaCategoria:*, il pulsante *Cancella voce* della finestra principale all'azione *cancellaVoce:* e per finire la voce *Salva Rubrica…* del menu *Rubrica* all'azione *salvaRubrica:*. Assegnate poi alla tabella l'oggetto *Controller* come *dataSource*.





Tornate al pannello *Classes* della finestra *MainMenu.nib* e sottoclassate un'altra volta `NSObject`, creando una nuova classe che chiamerete `NuovaCategoriaController`. Ad essa assegnate 2 outlet e 3 azioni, come mostrato in figura. Create i file e istanziate la classe. Poi collegate l'outlet *finestra* con la finestra denominata *Nuova categoria* e l'outlet nome col campo di testo editabile. Similmente, collegate il pulsante *Aggiungi* all'azione *aggiungi:*, il pulsante *Annulla* all'azione *annulla:* e la voce *Nuova Categoria* del menu *Rubrica* all'azione *creaNuovaCategoria:*. Salvate le modifiche e tornate ad XCode, dove sposterete i quattro file creati (*Controller.h*, *Controller.m*, *NuovaCategoriaController.h* e *NuovaCategoriaController.m*) nella cartellina *Classes* della finestra del progetto.

Prima di iniziare a scrivere il codice, vi invito a riflettere su quanto abbiamo fatto: abbiamo creato un'interfaccia relativamente complessa, e per mantenere piccoli e facilmente maneggiabili gli oggetti di controllo abbiamo deciso di crearne due. Come vedete la divisione tra elementi di interfaccia ed oggetti di controllo non è netta, nel senso che ad esempio tre voci su quattro del menu interagiscono con `Controller`, ma una interagisce con `NuovaCategoriaController`, perché è più sensato così. Entrambi gli oggetti di controllo saranno caricati ed inizializzati all'avvio dell'applicazione, e quale sia l'oggetto *principale* è solo una questione di gusti, una scelta che facciamo noi. Faremo sì che sia `Controller`, ma non c'è scritto da nessuna parte che debba essere lui. Semplicemente, implementando `Controller`, metteremo lì buona parte della funzionalità (e le variabili che memorizzeranno i dati), per cui sarà esso l'oggetto principale di controllo.

Iniziamo a scrivere un po' di codice. Come prima cosa creiamo un nuovo file vuoto, che

chiameremo *CostantiGenerali.h*, nel quale inseriremo un paio di definizioni che ci verranno comode più avanti:

```
#define   kNuovaCategoriaNotif              @"nuova categoria"
#define   kNomeCategoria                    @"nome categoria"
```

Salvatelo e notate che abbiamo definito due stringhe. La prima sarà usata per identificare una notifica, la seconda sarà la *chiave* per recuperare un oggetto in un dizionario (guarda guarda, che sia il dizionario che si può mettere ad argomento di `userInfo:` quando si solleva una notifica?).

Editate poi il file *Controller.h* in modo che appaia come segue:

```
/* Controller */

#import <Cocoa/Cocoa.h>
#import "CostantiGenerali.h"

#define   kStringaIncognita       @"Nuova voce"
#define   kDebug                  YES

@interface Controller : NSObject
{
    IBOutlet NSButton           *cancellaVoceButton;
    IBOutlet NSTableView        *tabella;
    IBOutlet NSMenuItem         *cancellaCategoriaMI;
    IBOutlet NSWindow           *finestra;
    NSMutableDictionary         *rubrica;

    int           numeroVoci;
}

- (IBAction)aggiungiVoce:(id)sender;
- (IBAction)apriRubrica:(id)sender;
- (IBAction)cancellaCategoria:(id)sender;
- (IBAction)cancellaVoce:(id)sender;
- (IBAction)salvaRubrica:(id)sender;

- (int)numberOfRowsInTableView:(NSTableView *)aTableView;
- (id)tableView:(NSTableView *)aTableView
                  objectValueForTableColumn:(NSTableColumn *)aTableColumn
                  row:(int)rowIndex;
- (void)tableView:(NSTableView *)aTableView
                  setObjectValue:(id)anObject
                  forTableColumn:(NSTableColumn *)aTableColumn
                  row:(int)rowIndex;

- (void)applicationDidFinishLaunching:(NSNotification *)n;
- (void)nuovaCategoria:(NSNotification *)n;
- (void)tableViewSelectionDidChange:(NSNotification *)n;
- (void)salvaPanelDidEnd:(NSSavePanel *)sheet
              returnCode:(int)returnCode contextInfo:(void *)contextInfo;
- (void)apriPanelDidEnd:(NSOpenPanel *)sheet
              returnCode:(int)returnCode contextInfo:(void *)contextInfo;
```

```
- (void)sincronizzaTabellaConRubrica;

@end
```

Notate che, oltre ad aver importato il file *CostantiGenerali.h*, abbiamo dichiarato due variabili in aggiunta agli outlet: una è un dizionario, e sarà la struttura entro la quale verrà memorizzata la rubrica. Infatti, scegliamo di far sì che ogni *categoria* (nome, cognome, indirizzo ecc. che l'utente vorrà creare) sia una *chiave* del dizionario; l'oggetto associato a questa chiave sarà un oggetto di classe `NSMutableArray` che conterrà le varie voci. L'altra variabile tiene semplicemente il conto di quante voci siano già state inserite nella rubrica. Tra i metodi, oltre alle azioni, spiccano i tre metodi del *dataSource* della tabella, tre metodi legati a notifiche o a metodi del delegato, due metodi che ricordano molto il meccanismo con cui si implementano i pannelli (la gestione delle azioni dell'utente in risposta ad un pannello), e un metodo di utilità per l'oggetto (`sincronizzaTabellaConRubrica`). Salvate *Controller.h*. Ora che ne direste di editare il file *Controller.m*, così da implementare questa classe?

```
#import "Controller.h"

@implementation Controller

- (id)init
{
     [super init];
     if(kDebug)
          NSLog(@"inizializziamo il controller");
     [[NSApplication sharedApplication] setDelegate:self];
     rubrica=[[NSMutableDictionary alloc] initWithCapacity:1];
     numeroVoci=0;
     return self;
}

- (void)dealloc
{
     [rubrica release];
}

- (void)applicationDidFinishLaunching:(NSNotification *)n
{
     NSEnumerator       *en;
     id                 obj;

     if(kDebug)
          NSLog(@"mettiamo a posto la baracca");
     en=[[tabella tableColumns] objectEnumerator];
     while(obj=[en nextObject])
          [tabella removeTableColumn:obj];
     [tabella reloadData];
     [[NSNotificationCenter defaultCenter] addObserver:self
                                       selector:@selector(nuovaCategoria:)
                                       name:kNuovaCategoriaNotif
                                       object:nil];
     [tabella setDelegate:self];
     [cancellaVoceButton setEnabled:NO];
```

```
    [cancellaCategoriaMI setAction:nil];
}

- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
    if(kDebug)
        NSLog(@"nella tabella ci sono %d righe",numeroVoci);
    return numeroVoci;
}

- (id)tableView:(NSTableView *)aTableView
                objectValueForTableColumn:(NSTableColumn *)aTableColumn
                row:(int)rowIndex
{
    if(kDebug)
        NSLog(@"diamo il contenuto di una cella");
    return [[rubrica objectForKey:
                [aTableColumn identifier]] objectAtIndex:rowIndex];
}

- (void)tableView:(NSTableView *)aTableView
                setObjectValue:(id)anObject
                forTableColumn:(NSTableColumn *)aTableColumn
                row:(int)rowIndex
{
    if(kDebug)
        NSLog(@"impostiamo il contenuto di una cella");
    [[rubrica objectForKey:
            [aTableColumn identifier]] replaceObjectAtIndex:rowIndex
                                                withObject:anObject];
}

- (IBAction)aggiungiVoce:(id)sender
{
    NSEnumerator       *en;
    id                      obj;

    if(kDebug)
        NSLog(@"aggiungiamo una voce");
    numeroVoci++;
    en=[[rubrica allKeys] objectEnumerator];
    while(obj=[en nextObject])
        [[rubrica objectForKey:obj] addObject:kStringaIncognita];
    [tabella reloadData];
}

- (IBAction)apriRubrica:(id)sender
{
    NSOpenPanel        *theOpenPanel;

    if(kDebug)
        NSLog(@"apriamo la rubrica");
```

```
    theOpenPanel=[NSOpenPanel openPanel];
    [theOpenPanel setAllowsMultipleSelection:NO];
    [theOpenPanel setCanChooseDirectories:NO];
    [theOpenPanel setCanChooseFiles:YES];
    [theOpenPanel setResolvesAliases:YES];
    [theOpenPanel beginSheetForDirectory:nil
                        file:nil
                        types:nil
                        modalForWindow:finestra
                        modalDelegate:self
                        didEndSelector:@selector(apriPanelDidEnd:
                                                 returnCode:contextInfo:)
                        contextInfo:nil];
}

- (void)apriPanelDidEnd:(NSOpenPanel *)sheet returnCode:(int)returnCode
                                            contextInfo:(void *)contextInfo
{
    if(returnCode)
    {
        if(kDebug)
            NSLog(@"apriamo il file");
        [rubrica release];
        rubrica=[[NSMutableDictionary alloc] initWithContentsOfFile:
                                                            [sheet filename]];
        [self sincronizzaTabellaConRubrica];
    }
    else
    {
        if(kDebug)
            NSLog(@"l'utente ha annullato l'apertura del file");
    }
}

- (IBAction)cancellaCategoria:(id)sender
{
    int           indice;

    if(kDebug)
        NSLog(@"cancelliamo una categoria");
    indice=[tabella selectedColumn];
    if(indice!=-1)
        [rubrica removeObjectForKey:
            [[[tabella tableColumns] objectAtIndex:indice] identifier]];
    [tabella removeTableColumn:
                        [[tabella tableColumns] objectAtIndex:indice]];
    [tabella reloadData];
}

- (IBAction)cancellaVoce:(id)sender
{
    NSEnumerator       *en;
```

```
    id                      obj;
    int                     indice;

    if(kDebug)
        NSLog(@"cancelliamo una voce");
    indice=[tabella selectedRow];
    if(indice!=-1)
    {
        if(numeroVoci>1)
        {
            numeroVoci--;
            en=[[rubrica allKeys] objectEnumerator];
            while(obj=[en nextObject])
                [[rubrica objectForKey:obj] removeObjectAtIndex:indice];
        }
    }
    [tabella reloadData];
}

- (IBAction)salvaRubrica:(id)sender
{
    NSSavePanel        *theSavePanel;

    if(kDebug)
        NSLog(@"salviamo la rubrica");
    theSavePanel=[NSSavePanel savePanel];
    [theSavePanel setCanSelectHiddenExtension:YES];
    [theSavePanel setExtensionHidden:[theSavePanel isExtensionHidden]];
    [theSavePanel beginSheetForDirectory:nil
                                file:nil
                                modalForWindow:finestra
                                modalDelegate:self
                                didEndSelector:@selector(salvaPanelDidEnd:
                                            returnCode:contextInfo:)
                                contextInfo:nil];
}

- (void)salvaPanelDidEnd:(NSSavePanel *)sheet returnCode:(int)returnCode
                                        contextInfo:(void *)contextInfo
{
    if(returnCode)
    {
        if(kDebug)
            NSLog(@"salviamo su file");
        [rubrica writeToFile:[sheet filename] atomically:YES];
    }
    else
    {
        if(kDebug)
            NSLog(@"L'utente ha annullato il salvataggio");
    }
}
```

```
- (void)nuovaCategoria:(NSNotification *)n
{
     NSMutableArray     *contenuto;
     int                i;
     NSTableColumn      *tc;
     NSString           *titolo;

     if(kDebug)
          NSLog(@"notifica: creiamo una categoria");
     contenuto=[NSMutableArray arrayWithCapacity:numeroVoci];
     for(i=0;i<numeroVoci;i++)
          [contenuto addObject:kStringaIncognita];
     titolo=[NSString stringWithString:
                              [[n userInfo] objectForKey:kNomeCategoria]];
     if(kDebug)
          NSLog(@"Il titolo di questa categoria e': %@",titolo);
     [rubrica setObject:contenuto forKey:titolo];
     tc=[[NSTableColumn alloc] init];
     [[tc headerCell] setStringValue:titolo];
     [tc setIdentifier:titolo];
     [tabella addTableColumn:tc];
     [tc release];
     [tabella reloadData];
}

- (void)tableViewSelectionDidChange:(NSNotification *)n
{
     int            indice;

     if(kDebug)
          NSLog(@"cambiata la selezione nella tabella");
     indice=[tabella selectedColumn];
     if(indice!=-1)
          [cancellaCategoriaMI setAction:@selector(cancellaCategoria:)];
     else
          [cancellaCategoriaMI setAction:nil];

     indice=[tabella selectedRow];
     if(indice!=-1 && numeroVoci>1)
          [cancellaVoceButton setEnabled:YES];
     else
          [cancellaVoceButton setEnabled:NO];
}

- (void)sincronizzaTabellaConRubrica
{
     NSEnumerator       *en;
     id                      obj;

     en=[[tabella tableColumns] objectEnumerator];
     while(obj=[en nextObject])
```

```
          [tabella removeTableColumn:obj];
     en=[[rubrica allKeys] objectEnumerator];
     while(obj=[en nextObject])
     {
          NSTableColumn      *tc;

          tc=[[NSTableColumn alloc] init];
          [[tc headerCell] setStringValue:obj];
          [tc setIdentifier:obj];
          [tabella addTableColumn:tc];
          [tc release];
          numeroVoci=[[rubrica objectForKey:obj] count];
     }
     [tabella reloadData];
}

@end
```

Commenteremo dopo questo codice. Per ora completiamo l'operazione di scrittura, agendo sul file *NuovaCategoriaController.h*, che renderete simile a questo:

```
/* NuovaCategoriaController */

#import <Cocoa/Cocoa.h>
#import "CostantiGenerali.h"

#define   kDebug             YES

@interface NuovaCategoriaController : NSObject
{
     IBOutlet NSWindow       *finestra;
     IBOutlet NSTextField    *nome;
     NSMutableString         *nomeCategoria;
}
- (IBAction)aggiungi:(id)sender;
- (IBAction)annulla:(id)sender;
- (IBAction)creaNuovaCategoria:(id)sender;

- (void)controlTextDidChange:(NSNotification *)n;
@end
```

Per finire, dopo aver salvato le modifiche, scrivete l'implementazione *NuovaCategoriaController.m*:

```
#import "NuovaCategoriaController.h"

@implementation NuovaCategoriaController

- (id)init
{
     [super init];
     nomeCategoria=[[NSMutableString alloc] initWithString:@""];
     return self;
```

```
}

- (void)awakeFromNib
{
     [nome setDelegate:self];
}

- (void)dealloc
{
     [nomeCategoria release];
}

- (IBAction)aggiungi:(id)sender
{
     if(kDebug)
          NSLog(@"aggiungiamo una categoria");
     [[NSNotificationCenter defaultCenter]
               postNotificationName:kNuovaCategoriaNotif
               object:self
               userInfo:[NSDictionary dictionaryWithObject:nomeCategoria
                                                    forKey:kNomeCategoria]];
     [finestra orderOut:sender];
}

- (IBAction)annulla:(id)sender
{
     [finestra orderOut:sender];
}

- (IBAction)creaNuovaCategoria:(id)sender
{
     [finestra makeKeyAndOrderFront:sender];
     [nome setStringValue:@""];
}

- (void)controlTextDidChange:(NSNotification *)n
{
     if(kDebug)
          NSLog(@"aggiorniamo il nome della nuova categoria");
     [nomeCategoria setString:[nome stringValue]];
}

@end
```

Uff! Che fatica! Salvate tutto, mi raccomando, non vorrete mica ridigitare tutto da capo, vero? Ora bevetevi una spremuta d'arancia, poi, belli rinfrescati, affrontate con me l'analisi di tutto questo codice.

Partiamo dalla classe `Controller`. Il metodo di inizializzazione imposta l'oggetto di controllo a delegato di `NSApp`, poi crea il dizionario mutevole che conterrà la rubrica. Inizialmente essa è vuota, e allora il dizionario viene creato con una capacità minima (1 coppia chiave-oggetto associato), che però lasciamo comunque vuota. Sempre giusto per mettere un numero da qualche parte, diciamo che `numeroVoci` è pari a zero (che poi è la verità). La prima cosa interessante succede nel metodo `applicationDidFinishLaunching:`, che viene

chiamato automaticamente al momento opportuno in virtù del fatto che il nostro oggetto di controllo si è già registrato come delegato di NSApp. Qui iniziamo col rimuovere tutte le colonne eventualmente presenti nella tabella salvata nel file di risorse, così da iniziare con una bella tabella vuota e pulita. Poi registriamo l'oggetto di controllo come interessato ad una certa notifica, il cui nome abbiamo definito in *CostantiGenerali.h*. Perché facciamo ciò? Perché avrete notato che tutta la faccenda di permettere all'utente di creare una nuova categoria per la nostra rubrica è gestita da NuovaCategoriaController. Come facciamo a far sapere a Controller che è stata creata una nuova categoria? Il metodo più semplice è proprio quello di usare le notifiche: all'occorrenza, NuovaCategoriaController solleverà una notifica di nome kNuovaCategoriaNotif nella quale specificherà mediante il dizionario passato come argomento di userInfo: il nome della categoria scelto dall'utente. Controller resterà in ascolto di questa notifica così da poter intraprendere le azioni necessarie. Successivamente, nel metodo applicationDidFinishLaunching: stabiliamo che l'oggetto di controllo funge anche da *delegato* per la tabella. Forse avrete notato, quando avete collegato *Controller* come *dataSource* della tabella in InterfaceBuilder, che un outlet della tabella era denominato *delegate*. Naturalmente, avremmo potuto stabilire che l'oggetto di controllo fungeva da delegato della tabella direttamente lì, in InterfaceBuilder. Perché abbiamo scelto di farlo qui, in XCode, scrivendo il codice di implementazione dell'oggetto di controllo? Non c'è una ragione precisa, diciamo che se una cosa è scritta nel codice è più difficile dimenticarsela (per lo meno per me, voi fate quello che volete). Ancora non sappiamo quali siano i metodi del delegato di NSTableView ai quali siamo interessati, né ancora sappiamo perché siamo interessati ad essi. Vi lascio la sorpresa a tra un po'. Per finire, il metodo rende inattivo il pulsante per cancellare le voci dalla rubrica (dal momento che non ci sono voci, perché lasciare il pulsante attivo?) e disabilita anche il comando del menu per cancellare una categoria (di nuovo, visto che la rubrica è vuota, è meglio disabilitare i comandi di cancellazione, che tanto non servono). Notate la differenza: un pulsante si disabilita inviando NO ad argomento del messaggio setEnabled:. Un elemento da menu, invece, si disabilita automaticamente se rimuovete l'azione a cui è collegato.

L'analisi dei tre metodi del *dataSource* della tabella, invece, ci aiuta a capire come abbiamo deciso di organizzare la rubrica. Che il numero di righe della tabella sia indissolubilmente legato al valore di numeroVoci non dovrebbe essere niente di sconvolgente. Più interessante è l'analisi del metodo tableView:objectValueForTableColumn:row:. Qui iniziamo ad identificare la colonna mediante il suo metodo identifier, anziché tag. Come sapete un *tag* è un numero intero, e non può essere altro che un numero intero. Può fare comodo in certe circostanze, ma nel nostro caso, dove non sappiamo quali e quante colonne l'utente vorrà creare, né che cosa conterranno, può non essere una scelta particolarmente felice. Fortunatamente gli oggetti di classe NSTableColumn offrono la possibilità di associare loro un oggetto *qualunque*, impostabile mediante il metodo setIdentifier: e richiamabile col metodo identifier. Allora come ci regoliamo noi? Come decidiamo di distinguere le colonne l'una dall'altra? La maniera più semplice è distinguerle per nome. Infatti, abbiamo già detto che ogni categoria della nostra rubrica è una voce del dizionario rubrica. Le voci dei dizionari, dicevamo, sono stringhe (di classe NSString); perché non usare il nome della categoria come titolo della colonna, come identifier della colonna e come voce del dizionario rubrica? Così dapprima chiamiamo [aTableColumn identifier] e passiamo la stringa risultante come chiave per la ricerca dell'array contenente le voci ([rubrica objectForKey:[aTableColumn identifier]]). Quindi estraiamo dall'array l'oggetto che si trova all'indice corrispondente al numero di riga che ci è stato richiesto. Naturalmente, quando l'utente fa doppio-click su una casella per editarne il contenuto, verrà chiamato il metodo che consente di impostare l'oggetto contenuto nella casella; la tecnica impiegata è la stessa, ma anziché usare il metodo objectAtIndex: di NSArray (e quindi di NSMutableArray che ne è sottoclasse) usiamo il metodo replaceObjectAtIndex:withObject:.

Vediamo ora che cosa succede quando l'utente sceglie la voce *Nuova Categoria* dal menu *Rubrica*. Come prima cosa viene chiamato il metodo creaNuovaCategoria: di NuovaCategoriaController. Questo metodo non fa altro che visualizzare a schermo la finestra apposita e azzerare il campo di testo editabile denominato nome. Da qui in avanti il

controllo passa all'utente, che potrà iniziare a digitare il nome desiderato per la categoria che vuole aggiungere. Per evitare che l'utente debba necessariamente premere "a-capo" alla fine dell'inserimento per confermare quanto digitato (cosa fastidiosa e che può portare confusione dell'utente che potrebbe non capire come mai il nome inserito apparentemente non è stato accettato), abbiamo dichiarato nel metodo `awakeFromNib` che `NuovaCategoriaController` è intenzionata a fare da delegato per il campo editabile di testo `nome`. È da notare che non possiamo qui implementare un metodo `applicationDidFinishLaunching:` perché `Controller` è già delegato di `NSApp`, e non può esservi più di un delegato per oggetto. Avremmo potuto registrare `NuovaCategoriaController` per ricevere la notifica `@"NSApplicationDidFinishLaunching"`, ma era più comodo implementare semplicemente il metodo`awakeFromNib`. Perché non scriviamo, ancora più semplicemente, `[nome setDelegate:self]` all'interno del metodo `init` di `NuovaCategoriaController`? La ragione è semplice, ma subdola: quando il metodo `init` viene eseguito, le risorse ancora non sono state caricate da file, e l'outlet `nome` ancora vale `nil`, ovvero non è collegato a nulla. Dobbiamo aspettare che tutte le risorse siano state caricate (cosa che sappiamo essere avvenuta nel momento in cui viene chiamato `awakeFromNib`) per poterci registrare come delegato di `nome`.

Ora che siamo delegati di un oggetto di classe `NSTextView`, possiamo implementare tutti i metodi che vogliamo che `NSTextView` e le sue superclassi mettono a disposizione del delegato. In particolare, siamo interessati al metodo `controlTextDidChange:`, che ci informa, in tempo reale, delle modifiche che l'utente apporta al testo digitato nel campo. Usiamo questa informazione per assegnare il contenuto corretto alla stringa (`NSMutableString`) `nomeCategoria`. L'utente potrà poi premere il pulsante *Annulla*, nel qual caso la finestra verrà semplicemente tolta dallo schermo, oppure il pulsante *Aggiungi*, nel qual caso verrà chiamato il metodo `aggiungi:`. Questo, prima di chiudere a sua volta la finestra, ha il dovere di far sapere a `Controller` che è stata aggiunta una categoria. Dal momento che la classe `NuovaCategoriaController` non ha un outlet che punta a `Controller`, decidiamo di sollevare una notifica di nome `kNuovaCategoriaNotif`. Naturalmente è indispensabile che `Controller` non solo sia informato del fatto che l'utente ha creato una nuova categoria, ma anche di quale sia il nome scelto dall'utente per essa. Ci viene in aiuto l'argomento che possiamo passare a `userInfo:` nel momento in cui solleviamo la notifica. Esso è un dizionario il cui contenuto è lasciato interamente a nostra scelta. Creiamo quindi un dizionario contenente un'unica coppia chiave-oggetto associato, che dichiariamo esplicitamente. La chiave è una stringa definita in *CostantiGenerali.h*, così che anche `Controller` possa rintracciarla. L'oggetto associato non è nient'altro che `nomeCategoria`.

Ora che la notifica è stata sollevata, `Controller`, che si era messo in ascolto di essa già a partire dal metodo `applicationDidFinishLaunching:`, verrà informato della novità con una chiamata effettuata automaticamente dal sistema al metodo `nuovaCategoria:`. Come prima cosa creiamo una nuova array mutevole, `contenuto`, che sarà destinata a contenere le voci di questa categoria. Dal momento che assoceremo questa array ad una chiave all'interno di un dizionario, essa sarà già memorizzata altrove e non abbiamo necessità di conservare un puntatore ad essa; pertanto, essa viene creata con un metodo autorelease ed associata ad una variabile locale del metodo (`contenuto`). Dopo aver creato l'array, la dotiamo di tanti elementi quanto è il valore di `numeroVoci`; ovviamente, essendo la categoria nuova, non abbiamo nulla di sensato da inserirci, così creiamo `numeroVoci` voci tutte uguali con la stringa `kStringaIncognita`. Subito dopo estraiamo dalla notifica passata ad argomento di `nuovaCategoria:` il dizionario che avevamo passato quando abbiamo sollevato la notifica, grazie ad una chiamata a `[n userInfo]`. Dal dizionario risultante estraiamo l'oggetto associato alla chiave `kNomeCategoria`, che non è nient'altro che la stringa (`NSString`) col nome scelto dall'utente. Assegnamo questa stringa alla variabile locale `titolo`, che poi usiamo come titolo della`headerCell` e come `identifier` di una nuova `NSTableColumn` che creiamo ed aggiungiamo a `tabella`. Ora che abbiamo fatto tutto non ci resta che chiamare `[tabella reloadData]` per aggiornare la tabella a schermo.

A questo punto l'utente può decidere di aggiungere una voce alla rubrica. Il metodo `aggiungiVoce:` viene chiamato, e la variabile `numeroVoci` viene incrementata di una unità. Chiediamo al dizionario `rubrica` di fornirci un'array con tutte le sue chiavi, così da poter scorrere su ognuna di esse in un ciclo `while()` grazie ad un `NSEnumerator`. In questo modo possiamo prelevare, in sequenza, tutte le `NSMutableArray` associate ad ogni chiave (categoria) del dizionario (rubrica) ed aggiungere un nuovo elemento (voce) ad esse, che naturalmente sarà una `kStringaIncognita`. Alla fine, chiamiamo `[tabella reloadData]` così da forzare il ridisegno a schermo della tabella.

Il meccanismo implementato dal metodo `cancellaVoce:` è assolutamente identico. Esso sfrutta il metodo `selectedRow` di `NSTableView` che ci informa sull'indice della riga (e quindi dell'array mutevole associata ad ogni categoria) che è stata selezionata dall'utente. Per tranquillità, verifichiamo che tale indice non valga -1 (che è il valore restituito da `selectedRow` quando nessuna riga è selezionata). Inoltre, non facciamo nulla se l'utente sta cercando di cancellare l'ultima voce.

Un po' più complesso è il metodo `cancellaCategoria:`. Dopo aver ottenuto, mediante il metodo `selectedColumn` di `NSTableView`, l'indice della colonna selezionata ed aver verificato anche in questo caso che non valga -1 (valore che assumerebbe se nessuna colonna fosse selezionata), chiediamo innanzitutto a tabella di fornirci un'array di tutte le sue colonne (`[tabella tableColumns]`) così che possiamo estrarne l'oggetto (di classe `NSTableColumn`) all'indice restituito da `selectedColumn`. Di tale oggetto estraiamo poi l'`identifier`, che usiamo come chiave per il dizionario `rubrica`, a cui comunichiamo di cancellare l'oggetto (`NSMutableArray`) associato. Cancelliamo poi la colonna dalla tabella (`[tabella removeColumnAtIndex:]`) e infine forziamo il ridisegno della tabella a schermo.

C'è un dettaglio che abbiamo trascurato, e la cui discussione non è più rinviabile. Nel metodo `applicationDidFinishLaunching:` avevamo disabilitato il pulsante per rimuovere voci dalla rubrica e il comando da menu per rimuovere categorie, dal momento che non ce n'erano disponibili. Ora però l'utente ha aggiunto 35 categorie e 864 voci, e deve avere la possibilità di poterne rimuovere alcune se lo desidera. Dov'è che riabilitiamo il pulsante e il comando del menu? Usiamo uno stratagemma: avevamo registrato `Controller` come delegato di `tabella`. Si dà il caso che i delegati di `NSTableView` possano implementare un metodo convenientemente denominato `tableViewSelectionDidChange:`. Esso viene chiamato tutte le volte che l'utente, agendo su una tabella, ne modifica la selezione (viene selezionata o deselezionata una riga o una colonna). A che ci serve ricevere questa informazione? Chiamando i metodi `selectedRow` e `selectedColumn` di `tabella` nell'implementazione del metodo `tableViewSelectionDidChange:` possiamo sapere se vi sia una riga o una colonna selezionata. Nel caso una riga risulti selezionata, abiliteremo, passando `YES` ad argomento di `setEnabled:`, il pulsante che consente di cancellare voci (sempre che `numeroVoci` sia maggiore di 1); nel caso una colonna sia selezionata, assegneremo al comando *Elimina Categoria* nel menu *Rubrica* l'azione `cancellaCategoria:`, e questo abiliterà automaticamente l'elemento nel menu. Gli elementi di interfaccia saranno invece disabilitati qualora non siano selezionate righe o colonne. In questo modo, implementando un delegato di `NSTableView`, abbiamo ottenuto un risultato molto interessante: gli elementi di interfaccia che non sono compatibili con la selezione fatta dall'utente sono disabilitati, impedendo così all'utente di usarli (e riducendo le sue possibili cause di confusione). Non male, vero?

Per concludere vogliamo dare all'utente la possibilità di salvare su file la sua rubrica, e di poterla richiamare da file in futuro. Iniziamo dal salvataggio. Il comando *Registra Rubrica…* nel menu *Rubrica*, se selezionato, chiama l'azione `salvaRubrica:`. Essa usa un oggetto, `theSavePanel`, che appartiene ad una classe Cocoa denominata `NSSavePanel`. Abbastanza intuitivamente, tale classe si prende la briga di visualizzare un pannello standard che consenta all'utente di navigare nella struttura ad albero del disco e decidere dove e con che nome salvare un file. L'oggetto `theSavePanel` viene creato con la chiamata a `[NSSavePanel savePanel]` (è di tipo autorelease, quindi non abbiamo di che preoccuparci per la memoria), poi possiamo configurarlo precisando ad esempio che vogliamo consentire all'utente di scegliere se visualizzare o no l'estensione da assegnare al file. Infine visualizziamo il pannello, attaccandolo

alla finestra collegata all'outlet `finestra`, e facendo iniziare la navigazione del disco al percorso indicato al primo argomento di `beginSheetForDirectory:…`. Abbiamo passato `nil`, ad indicare che lasciamo al sistema il compito di ricordarsi quale sia stata l'ultima cartella utilizzata dall'utente col nostro *Esempio7* (sarà la cartella home dell'utente, se è la prima volta che si usa il programma); se avessimo passato una stringa con un percorso valido, il pannello si sarebbe aperto visualizzando il contenuto della cartella da noi specificata. Similmente possiamo specificare un file da selezionare automaticamente all'apertura del pannello passandolo come secondo argomento. Poi si innesca il meccanismo che avevamo già visto con i pannelli nel capitolo 5: dobbiamo dire quale oggetto si occuperà di gestire le conseguenze delle scelte fatte dall'utente (`self`), e quale metodo in particolare se ne prenderà carico. Il metodo deve iniziare con un nome a scelta e supportare tre argomenti fissi (i nomi degli ultimi due argomenti sono obbligatoriamente quelli qui riportati), quindi deve sempre essere della forma `qualchecosa:returnCode:contextInfo:`. Qui abbiamo deciso che `qualchecosa` è `salvaPanelDidEnd:`. Come per gli altri pannelli, anche qui, se vogliamo, possiamo passare un oggetto che contenga quello che più ci aggrada come argomento di `contextInfo:`.

Il metodo `salvaPanelDidEnd:returnCode:contextInfo:` verrà allora chiamato automaticamente nel momento in cui l'utente avrà chiuso il pannello confermando o annullando le sue scelte. L'argomento `returnCode` assume un valore diverso da zero se l'utente ha cliccato sul pulsante che consente di salvare il file su disco. In questo caso, `[sheet filename]` contiene una `NSString` col percorso completo del file da salvare, così come l'ha scelto l'utente. Non dobbiamo fare altro che passare questo percorso al metodo `writeToFile:atomically:` implementato da `NSDictionary` e dalla sua sottoclasse `NSMutableDictionary` (e da `NSArray` e da `NSString` con le loro sottoclassi mutevoli) perché il dizionario memorizzato in `rubrica` venga automaticamente salvato su disco al percorso indicato. È impressionante vedere come l'operazione di salvataggio si "propaghi" automaticamente agli oggetti contenuti in `rubrica` (delle `NSMutableArray` che contengono a loro volta delle `NSString`), che vengono salvati su disco sotto forma di un file xml. Una rapida ispezione (ad esempio con TextEdit) di un file generato dalla nostra rubrica indirizzi mostrerà infatti una struttura di questo tipo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
     <key>cognome</key>
     <array>
          <string>Jobs</string>
          <string>Wozniak</string>
     </array>
     <key>e-mail</key>
     <array>
          <string>steve@apple.com</string>
          <string>woz@apple.com</string>
     </array>
     <key>nome</key>
     <array>
          <string>Steve</string>
          <string>Steve</string>
     </array>
</dict>
</plist>
```

Il meccanismo di recupero di questi dati da file è assolutamente analogo. Usiamo la classe

`NSOpenPanel` per creare e configurare un pannello standard di navigazione nel disco ed apertura di file. Qui possiamo scegliere se consentire all'utente di selezionare più di un file per volta, se possa selezionare anche cartelle, se gli alias vadano considerati come tali o come file a sé. Infine, il metodo da noi scelto (le regole sono le stesse che per `NSSavePanel`) viene chiamato, e da esso possiamo inizializzare il nostro dizionario `rubrica` semplicemente usando il metodo `initWithContentsOfFile:` a cui passiamo l'argomento `[sheet filename]` che contiene il nome del file, col percorso completo, scelto dall'utente. Il dizionario con le sue `NSMutableArray` che contengono `NSString` viene ricostruito automaticamente per noi. Non ci resta che chiamare il metodo `sincronizzaTabellaConRubrica` per eliminare tutte le colonne presenti nella tabella e crearne di nuove, una per ogni categoria (chiave) memorizzata in `rubrica`, ed assegnare a `numeroVoci` il valore opportuno.

Notate che l'ordine con cui compaiono le colonne nella tabella dopo che avete richiamato il file da disco non è necessariamente lo stesso che avevate quando l'avete salvato. La ragione è che Cocoa non vi dà nessuna garanzia sull'ordine con cui vengono memorizzate le coppie chiave-oggetto associato in un dizionario (mentre l'ordine è preservato nelle array, ovviamente). Se volessimo correggere questo problema, dovremmo memorizzare nel file ad esempio un'array (il cui ordine viene preservato) che contenga l'elenco ordinato dei titoli delle colonne (sempre usando il metodo `writeToFile:atomically:`, ad esempio, che è implementato anche da `NSArray`), così da poter ripristinare il corretto ordine in `sincronizzaTabellaConRubrica`.

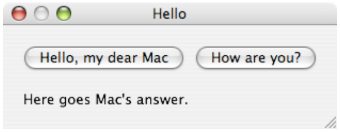
Congratulazioni! Siete usciti (incolumi, spero) dall'esempio più complesso di tutto il manuale, un'applicazione vera e propria (anche se un po' rudimentale) nella quale abbiamo dato un impiego e una ragione d'essere a quasi tutto quello che abbiamo imparato in questo volume. Tuttavia, utilità a parte, la nostra applicazione non è ancora pronta per la distribuzione al grande pubblico: bisogna ancora curare alcuni dettagli, piccoli ma fondamentali. È di questo che ci occuperemo nel prossimo capitolo, non rielaborando l'*Esempio7* ma su un esempio appositamente studiato così da non essere inutilmente complicato e da mostrare in un colpo solo tutti i temi che dobbiamo ancora trattare.

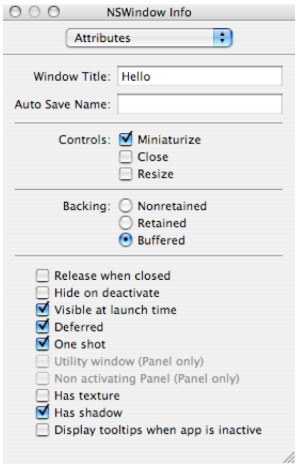
9. Ultimi ritocchi

Localizzazione

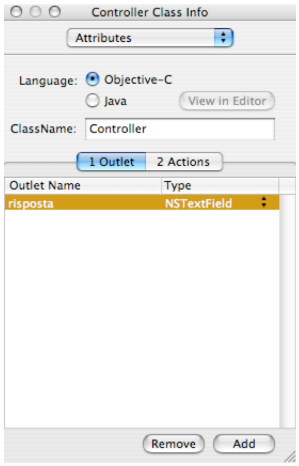
Fino ad ora abbiamo bellamente ignorato che il file *MainMenu.nib* che andavamo a modificare nei nostri programmi fosse contenuto in una cartella *English.lproj* e come tale si aspettasse un'interfaccia grafica in inglese. Ignorare questo fatto va bene all'inizio, quando si prende dimestichezza con Cocoa, ma poi la cosa non può durare. Soprattutto non è ammissibile se volete distribuire la vostra applicazione, ad esempio tramite Internet. È fondamentale che ogni localizzazione che fornite sia nella lingua corretta. Per vedere come si possa fare ciò, qui presentiamo il nostro ultimo esempio, che indovinate un po' si chiamerà *Esempio8*, che sarà in assoluto il programma più stupido che vi capiterà di scrivere. Non fa veramente nulla di utile, se non mostrare alcuni dei "ritocchi finali" che obbligatoriamente dovete apportare alle vostre applicazioni prima di poterle distribuire.

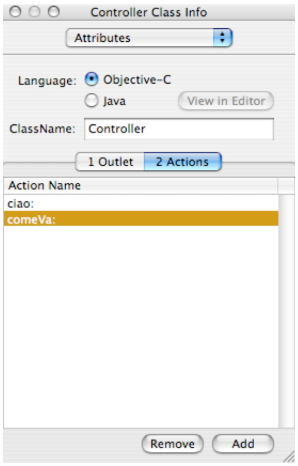
Chiudete come sempre tutti i documenti aperti con XCode e InterfaceBuilder, e create, per l'ultima volta, un nuovo progetto di tipo *Cocoa application*, che chiamerete *Esempio8*. Fate due click sull'icona di *MainMenu.nib* di modo che esso venga aperto in InterfaceBuilder. Tenendo presente che stiamo lavorando sulla localizzazione in inglese, editate la finestra principale del programma in modo che assomigli a quella riportata qua in figura. Come potete vedere già da questo primo scorcio, il programma è la fiera dell'inutile. Due pulsanti permettono di "comunicare" col Mac, salutandolo o chiedendogli come sta. Il Mac risponderà con una frase che verrà scritta nel campo di testo sottostante. Risparmiate i commenti, per favore, è uno schifo e lo so, ma così vediamo bene come si fa a localizzare (che è un processo in due *fasi*, lo sapevate?). Comunque, la finestra di informazioni della finestra principale dell'applicazione è qui riportata in figura, cosicché possiate applicare anche voi le stesse impostazioni.



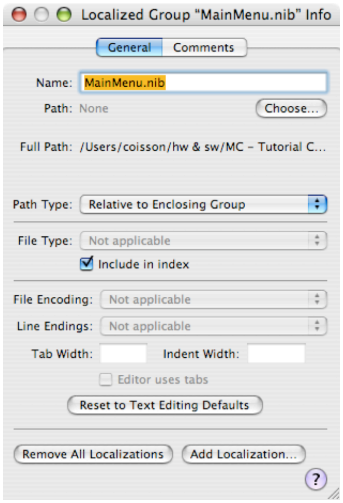


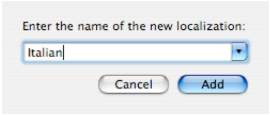
Dal pannello *Classes* della finestra denominata *MainMenu.nib* sottoclassate poi `NSObject` e create la classe `Controller`. Ad essa assegnerete un outlet e due azioni, come mostrato in figura. Create i file per la classe di controllo e istanziatela. Collegherete l'outlet al campo statico di testo, il pulsante di sinistra all'azione *ciao:* e il pulsante di destra all'azione *comeVa:*. Fatto ciò, salvate le modifiche e tornate ad XCode. Qui, manco a dirlo, sposterete i file *Controller.h* e *Controller.m* nella cartellina *Classes* della finestra del progetto.





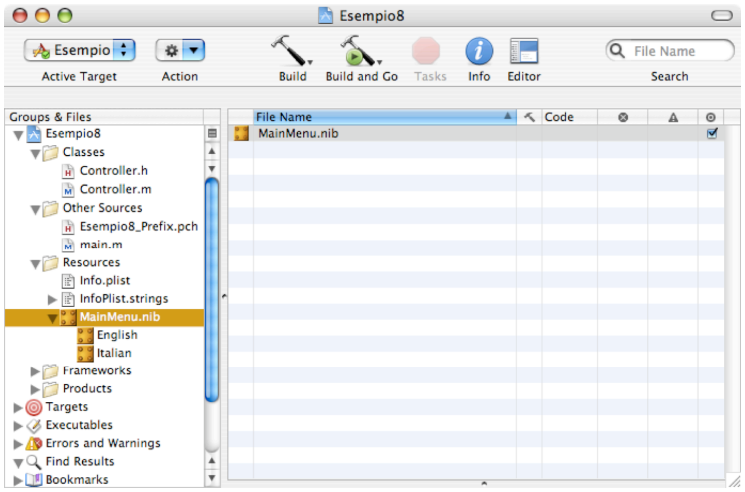
Ora selezionate con un click del mouse l'icona di *MainMenu.nib*, e richiamate la finestra di informazioni digitando mela-I, come mostrato nella figura all'inizio della pagina seguente. Individuate in fondo a destra un pulsante denominato *Add localization…*. Premetelo e nel pannello che compare scrivete *Italian*, come mostrato in figura. Confermate premendo il pulsante *Add*. Dalla finestra del progetto, "aprite" il triangolino posto di fianco a *MainMenu.nib*. Ora vi verranno mostrate tutte le localizzazioni in cui è disponibile il file di risorse dell'applicazione, ovvero *English* e *Italian*, come mostrato in figura. Ovviamente XCode o





InterfaceBuilder non fanno la traduzione per noi dall'inglese all'italiano, ma ci mettono a disposizione un diverso file di risorse per ogni lingua che vorremo supportare. Il sistema operativo sceglierà automaticamente la lingua con cui eseguire l'applicazione in base alle scelte che l'utente ha fatto nelle Preferenze di Sistema (e alle lingue disponibili per l'applicazione).

Prima di implementare il codice, iniziamo allora a tradurre l'interfaccia utente in italiano. Fate due click su *Italian* (all'interno di *MainMenu.nib*), e osservate che in InterfaceBuilder vi si apre una copia del file di risorse che avete appena creato. Cercate di non confondervi su quale sia la copia in inglese e quale quella in italiano, e traducete l'interfaccia utente in modo che la finestra principale dell'applicazione sia come mostrato in figura. Osservate, con divertito stupore, come i collegamenti ad outlet ed azioni si siano perfettamente conservati, e come pertanto sia necessario solo tradurre l'interfaccia utente. Naturalmente, a questo punto potreste anche voler tradurre i menu, e magari togliere quella brutta scritta *NewApplication* che compare in svariati menu e sostituirla col nome dell'applicazione (*Esempio7*, nel nostro caso). Quando vi ritenete soddisfatti dei ritocchi all'interfaccia utente e della localizzazione in italiano, salvate le modifiche e tornate in XCode, perché è giunta l'ora di implementare il codice.



Anzi no. Prima facciamo ancora un commento. Fin qui è stato tutto facile perché abbiamo preso un file di risorse completo e abbiamo aggiunto una localizzazione; in questo modo, l'intero file di risorse è stato duplicato e noi l'abbiamo tradotto in un'altra lingua. Ma se nel corso dello sviluppo dell'applicazione decidete di aggiungere risorse alla stessa (un po' come abbiamo fatto in qualche esempio), ad esempio aggiungete nuovi elementi di interfaccia, ricordate che dovete aggiungerli separatamente *per ogni file di risorse localizzato*: non basterà più aggiungerli alla sola localizzazione in inglese per ritrovarseli poi in tutte le altre localizzazioni, dovrete aggiungerli ad ogni singolo file di risorse localizzato. E alla stessa maniera dovrete effettuare tutti i collegamenti di outlet ed azioni con i nuovi elementi di interfaccia, singolarmente in ogni file localizzato. È quindi una buona idea localizzare l'applicazione il più tardi possibile, quando l'interfaccia (per lo meno della versione 1.0) è ben definita e stabile, così da non dover poi apportare troppe modifiche ad N file di risorse localizzati (con N pari al numero delle lingue che decidete di supportare).

Bene, abbiamo finito con questi deliri. Ora, direte voi, per quale motivo dobbiamo rimbecillire a scrivere il codice di questa applicazione se tanto ormai la localizzazione l'abbiamo fatta? La ragione è che abbiamo fatto la *fase 1* della localizzazione, ovvero quella che riguarda le risorse "statiche". Ma c'è ancora la *fase 2*, quella che riguarda le risorse "dinamiche" (tipicamente stringhe di testo), che non sono "congelate" in un elemento di interfaccia, ma che cambiano in base alle circostanze. Se non avete capito niente, fidatevi, e iniziate a digitare il file di interfaccia *Controller.h*:

```
/* Controller */

#import <Cocoa/Cocoa.h>

#define   kCiao              @"Hello"
#define   kComeStai          @"I'm fine, thanks! And you?"

@interface Controller : NSObject
{
    IBOutlet NSTextField *risposta;
}
- (IBAction)ciao:(id)sender;
- (IBAction)comeVa:(id)sender;
@end
```

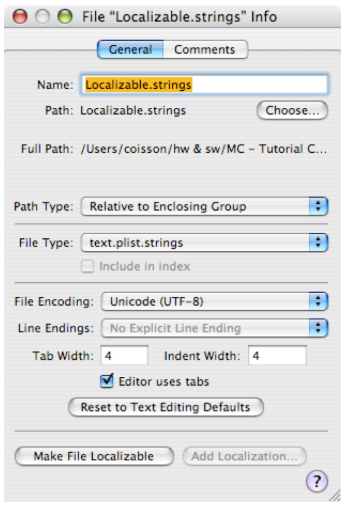
Salvatelo, poi editate il file di implementazione *Controller.m*:

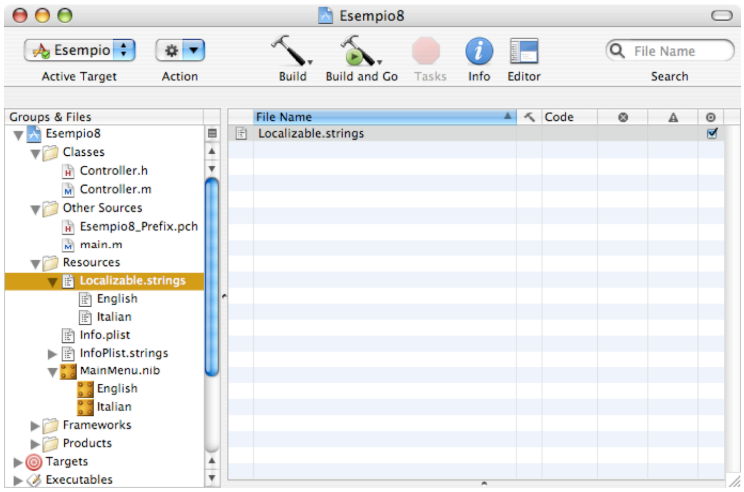
```
#import "Controller.h"

@implementation Controller

- (IBAction)ciao:(id)sender
{
     [risposta setStringValue:
                    [[NSBundle mainBundle] localizedStringForKey:kCiao
                    value:kCiao table:nil]];
}

- (IBAction)comeVa:(id)sender
{
     [risposta setStringValue:
                    [[NSBundle mainBundle] localizedStringForKey:kComeStai
                    value:kComeStai table:nil]];
}

@end
```

Salvatelo. Poi create ancora un nuovo file *vuoto*, e chiamatelo *Localizable.strings* (il nome è *fondamentale* e deve essere *esattamente* questo e non un altro). Spostate (sempre per fare ordine) il file *Localizable.strings* dalla cartellina *Classes* alla cartellina *Resources* nella finestra del progetto di XCode. Editate il file in modo che abbia questo aspetto:

```
"Hello" = "Hello";
"I'm fine, thanks! And you?" = "I'm fine, thanks! And you?";
```

Rispettate rigorosamente la sintassi, in particolare la presenza del punto-e-virgola alla fine di ogni riga. Osservate che le scritte a sinistra del segno "=" sono *esattamente le stesse* che compaiono nel file *Controller.h* per le due costanti `kCiao` e `kComeStai`. Vedremo tra poco perché. A destra del segno "=", invece, dovete mettere, *nella lingua in cui state localizzando*, la medesima espressione che sta a sinistra. Siccome stiamo lavorando in inglese, le due espressioni a sinistra e a destra del segno "=" sono identiche. Salvate le modifiche, poi selezionate il file *Localizable.strings* nella finestra del progetto e, con mela-I, richiamate la finestra di informazioni, mostrata qui in figura. Cliccate sul pulsante *Make file localizable*,

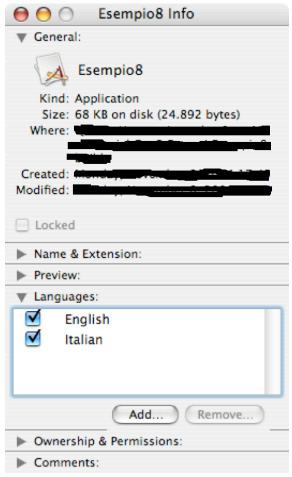




quindi aggiungete la variante italiana esattamente come avete fatto prima per *MainMenu.nib* (cliccate su *Add localization…*, poi inserite *Italian* e cliccate su *Add*). Osservate che il file *Localizable.strings* ha ora due varianti, una in inglese e una in italiano, come mostrato in figura. Fate un doppio click sulla variante *Italian* e modificate il file in modo che abbia il contenuto qui di seguito mostrato:

```
"Hello" = "Ciao";
"I'm fine, thanks! And you?" = "Io sto bene, grazie. E tu?";
```

Osservate che la parte a sinistra del segno "=" è sempre esattamente la stessa, mentre la parte a destra è la sua traduzione in italiano.



Ora potete salvare tutti i file che ancora non avete salvato, compilare ed eseguire il programma. Divertitevi a cliccare sui suoi due pulsanti e a vedere le risposte che vi dà il Mac. Se volete provare il programma in un'altra lingua, andate nel Finder, trovate la cartella in cui avete salvato l'*Esempio8*, aprite la cartella *build* e lì dentro individuate il file denominato *Esempio8*. Selezionatelo e richiamate la finestra di informazioni (di nuovo mela-I), che sarà una cosa simile a quella qui mostrata in figura. Divertitevi a disabilitare una lingua per volta e a lanciare l'applicazione direttamente dal Finder, col normale doppio-click. Visto che roba? Potete averla sia in italiano che in inglese, perfettamente localizzata!

Ora esaminiamo il codice. In realtà non c'è niente di terribile, e come avrete capito tutto il trucco sta nell'aver localizzato il file di risorse *MainMenu.nib* (e a caricare quello nella lingua giusta ci pensa automaticamente il sistema operativo) e in quell'espressione un po' strana che compare nelle due azioni, quella al posto dell'espressione ben più semplice che avremmo usato fino al capitolo precedente (dovre avremmo passato come argomento di `setStringValue:` direttamente la stringa desiderata): `[[NSBundle mainBundle] localizedStringForKey:value:table:]`. Di che cosa si tratta? `[NSBundle mainBundle]` è il modo che Cocoa ci offre per avere accesso a tutte le risorse della nostra applicazione. Già sapete, e se non lo sapete lo imparate adesso, che le applicazioni per MacOS X non sono dei semplici file, ma sono in realtà delle *cartelle speciali*, dette *bundle*. Provate, da Finder, a selezionare ancora l'icona dell'applicazione *Esempio8* e fate control-click su di essa (o click col tasto destro del mouse, se il vostro mouse ha almeno due tasti). Dal menu contestuale che appare selezionate la voce *Mostra contenuto pacchetto*. Si aprirà una nuova finestra, che all'interno mostra tutte le meraviglie di cui è composta l'applicazione, compresi i file di risorse, i file eseguibili e ogni altra cosa che avrete deciso di includere (risorse sonore, risorse grafiche, file di documentazione per l'help in linea, qualunque cosa). Orbene, `[NSBundle mainBundle]`

vi dà accesso a tutto questo. Noi abbiamo deciso di usare il metodo `localizedStringForKey:value:table:` che va alla ricerca del file *Localizable.strings* specifico per la lingua che è stata scelta per eseguire l'applicazione (ecco perché è *fondamentale* che il file *Localizable.strings* si chiami così e non in un'altra maniera, se no questo metodo non lo può trovare). Rintracciato il file, il metodo cerca, tra tutte le righe di cui è costituito, quella in cui alla sinistra del segno "=" sia riportata *l'esatta espressione* passata come primo argomento. Ecco perché era *fondamentale* che il valore delle costanti `kCiao` e `kComeStai` fosse *identico* a quanto scritto a sinistra del segno "=" in *entrambe* le localizzazioni del file *Localizable.strings*. Rintracciata la riga opportuna, il metodo restituisce, sotto forma di una `NSString`, l'espressione che trova a *destra* del segno "=", ovvero l'espressione tradotta nella lingua scelta per l'applicazione. Nel caso non venga trovata la riga cercata, o il file *Localizable.strings* non esista per la lingua scelta, viene restituita la stringa passata come secondo argomento del metodo. Ecco perché è *fondamentale* che le costanti `kCiao` e `kComeStai` siano espressioni compiute che, alla bisogna, possano essere usate nell'interfaccia utente (anche se non nella lingua opportuna). L'ultimo argomento, se diverso da `nil`, specifica il file (localizzato) nel quale avete memorizzato la traduzione di queste stringhe. Se passate `nil`, verrà usato il file *Localizable.strings*. In questa maniera, in applicazioni particolarmente complesse, potete suddividere le stringhe da localizzare in più file, raggruppandole ad esempio per argomenti o per funzionalità.

Un'ultima nota: se avete solo *Localizable.strings* come file di localizzazione delle stringhe e volete risparmiare un po' di battitura di codice, potete usare una funzione alternativa, `NSLocalizedString()`, a cui passate due argomenti, ovvero la *chiave* (quella a sinistra del segno "=") e l'*alternativa*. Scrivere pertanto

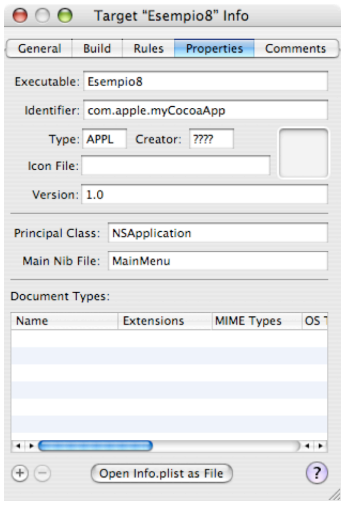
```
NSLocalizedString(kCiao,kCiao)
```

è del tutto equivalente a scrivere

```
[[NSBundle mainBundle] localizedStringForKey:kCiao value:kCiao table:nil]
```

Icona e Versione

Ogni applicazione che si rispetti, quindi anche le vostre, non può mancare di un'icona. Disegnatela come volete e col programma che volete, ma fatela di 128x128 pixel, e possibilmente gestite le trasparenze con un bel canale alpha. Poi, aprite l'applicazione Icon Composer che si trova in /Developer/Applications/Utilities, e incollate l'icona che avete generato nel riquadro più grande, quello da 128x128 pixel per l'appunto. Salvate il file col nome che volete (ad esempio *appIcon*), l'estensione sarà obbligatoriamente *.icns*. Ora dovete aggiungere l'icona alle risorse dell'applicazione. Da XCode, selezionate *Add to Project…* dal menu *Project*, e scegliete il file *appIcon.icns* (o come l'avete chiamato) che avete appena creato. Poi, dalla finestra del progetto in XCode, aprite il triangolino di fianco alla voce *Targets*, e selezionate l'unica voce esistente, ovvero quella col nome della vostra applicazione. Aprite la finestra di informazioni digitando mela-I, e selezionate il pannello *Properties*. Vi comparirà una finestrella simile a quella mostrata qua in figura. Digitate il nome del file contenente l'icona (*appIcon* ad esempio) *senza* l'estensione nel campo di testo denominato *Icon file*. Ora l'icona che avete creato è indissolubilmente legata alla vostra applicazione (per visualizzarla correttamente nel Finder potrebbe essere necessario fare un logout e poi nuovamente un login). Da qui, tra l'altro, potete assegnare alla vostra applicazione il numero di versione che ritenete opportuno (di default è il numero 1.0). Sempre da qui, potete decidere il *nome* del file contenente le preferenze dell'applicazione. La gestione delle preferenze va al di là dei limiti di questo manuale, quindi prendete questa informazione per cultura personale. A meno che non sappiate *molto bene* che cosa state facendo, non modificate il valore degli altri campi, anche di quelli compresi negli altri pannelli della finestra.



Infine, individuate nella finestra del progetto il file

InfoPlist.strings, aggiungetegli la localizzazione in italiano, e poi editate, tanto nella variante inglese quanto in quella italiana, il suo contenuto in modo che in corrispondenza della voce `NSHumanReadableCopyright` vi sia un riferimento a voi e all'eventuale copyright (o copyleft) che copre la vostra applicazione.

Distribuzione

Siamo quasi pronti per distribuire l'applicazione al grande pubblico. Mancano ancora due cose, di cui una *indispensabile* e una *fortemente raccomandata*. Partiamo da quest'ultima. È buona cosa creare un po' di documentazione per l'applicazione, anche solo una paginetta, con qualche screenshot e il vostro recapito (un sito web, un indirizzo e-mail, queste cose qui). È possibile includere questa documentazione (in formato html) all'interno dell'applicazione e renderla visibile tramite l'help in linea di MacOS X, tuttavia anche quest'operazione va al di là degli scopi di questo manuale. Lasciate quindi la documentazione sotto forma di file a sé stanti, e magari create anche una piccola pagina web che descriva le funzionalità della vostra applicazione.

Prima di distribuirla, però, dovete ancora fare una cosa *indispensabile*. Al fine di supportare la compilazione distribuita, XCode linka le applicazioni che create usando un particolare framework denominato ZeroConf. Questo avviene anche se il vostro Mac non è collegato in rete, anche se non avete attivato l'opzione di distribuire la compilazione tra vari Mac presenti in rete, anche se non ve ne può fregare niente di tutto ciò. Non ho idea del perché di tutto questo, ma non dubito che ci siano delle buone ragioni. Dove sta il problema? Sta nel fatto che il framework ZeroConf è presente *solo* sui Mac su cui sia stato installato XCode. Sugli altri Mac semplicemente non c'è. Quindi, se volete che la vostra applicazione funzioni *anche* sui Mac di coloro che non hanno installato XCode, è bene che prendiate provvedimenti. Per fortuna è facile.

Da XCode, cliccate sulla *prima riga* della finestra del progetto, quella riga che riporta il nome della vostra applicazione, quella al di sotto della quale compaiono le cartelline *Classes*, *Other Sources*, *Resources* ecc. Aprite la finestra delle informazioni digitando mela-I, e individuate il pannello *Styles*. Qui, dal menu a comparsa, selezionate la voce *Deployment*. Fate piazza pulita di tutte le compilazioni precedenti selezionando *Clean all Targets* dal menu *Build*, poi ricompilate l'applicazione (potete ricompilare senza lanciare l'applicazione digitando mela-B anziché mela-R, B sta per *build*).

Congratulazioni! Avete concluso. Ora potete prendere l'applicazione, tramite il Finder, dalla cartella *build* all'interno della cartella in cui avete salvato il progetto, e distribuirla (assieme ai suoi file di documentazione e a tutto quello che vi verrà in mente di accludere) sotto forma di un file compresso (.zip o .tar.gz) o di una comoda immagine disco (.dmg). Magari caricatela su un vostro sito web, fatevi un po' di pubblicità, e vedrete che nel giro di breve tempo frotte di clienti danarosi si daranno un gran da fare per rimpinguare cospicuamente il vostro conto in banca. Diventerete presto schifosamente ricchi. A quel punto, ricordatevi di me!

10. Andare oltre (Bibliografia)

Le cose più urgenti

Abbiamo appena iniziato ad esplorare il vasto mondo di Cocoa. È quindi comprensibile che ci siano un sacco di argomenti che abbiamo trascurato, ma la speranza è che questo manuale possa aiutarvi ad affrontarli con maggiore serenità. Tra le classi con cui potreste avere a che fare abbastanza presto, e che qui non sono state esaminate, segnalerei `NSFileManager`, `NSURL`, `NSColor`, `NSDrawer`, `NSImage`, `NSMatrix`, `NSSound`, `NSTextView`, `NSView`, `NSWindow`. Ce ne sono molte altre, ma probabilmente queste saranno le prime che avrete bisogno di utilizzare. Con la calma dovuta, quindi, iniziate a pensare di leggervi, saltuariamente, un po' di documentazione relativa ad esse. Naturalmente, prima di fare ciò, rileggetevi per bene la documentazione relativa a tutte le classi che abbiamo trattato fin qua, perché contengono molte più cose di quelle che abbiamo avuto tempo di esaminare.

Inoltre, potreste essere interessati a sapere come funzionano, e come si possano implementare, le cosiddette *Cocoa document-based applications*, col loro corredo delle classi `NSWindowController` e `NSDocumentController`. È un argomento vasto che non ha potuto trovare spazio in questo manuale. Vi rimando alla bibliografia per informazioni ulteriori su questo tema.

Per non parlare dell'affascinante argomento delle *preferenze*, ovvero del modo che Cocoa e MacOS X mettono a disposizione del programmatore per gestire, in maniera semplice ed efficace, preferenze personalizzate per ogni utente. Anche in questo caso andiamo al di là dei limiti di questo manuale, vi rimando alla bibliografia riportata più sotto per ulteriori informazioni.

Consultare la documentazione di Cocoa

Dicevamo dell'importanza di consultare la documentazione delle classi Cocoa che vi interessano, ivi comprese quelle che abbiamo utilizzato più o meno estensivamente in questo manuale. Un metodo sicuramente efficace è quello di consultare la documentazione in rete disponibile presso il sito di Apple. Tuttavia, in locale, il modo migliore per fare ciò è affidarsi ad acluni ottimi programmi (freeware) che permettono di sfogliare i file di documentazione che i DeveloperTools hanno installato sul vostro hard disk (li potete trovare nella cartella /Developer/Documentation/Cocoa/Reference). I programmini in questione sono strutturati come dei browser, con tanto di funzioni di ricerca e link dinamici (e possibilità di accedere alla documentazione in rete), e si chiamano **AppKiDo** (http://homepage.mac.com/aglee/downloads/) e **CocoaBrowser** (http://homepage2.nifty.com/hoshi-takanori/cocoa-browser/).

Bibliografia

Mi permetto qui di presentarvi alcune risorse, sotto forma di libri e siti web, che potete consultare per ottenere maggiori informazioni su Cocoa.

Partiamo dai libri (purtroppo in inglese). Un ottimo testo introduttivo a Cocoa, scritto molto bene e in maniera molto chiara, è

S. Garfinkel, M.K. Mahoney, *Building Cocoa Applications (a step by step guide)*, O'Reilly.

Un altro testo introduttivo, secondo me meno ben fatto del precedente ma comunque interessante, è

A. Hillegass, *Cocoa Programming for MacOS X*, Addison-Wesley.

Un testo di riferimento, un tomo di oltre 1000 pagine assolutamente non adatto come lettura introduttiva ma su cui potete trovare documentazione dettagliata su qualunque classe Cocoa vi venga in mente, è

S. Anguish, E.M. Buck, D.A. Yacktman, *Cocoa Programming*, SAMS.

Naturalmente esistono svariate risorse anche in rete. Una delle migliori, in inglese, è una sorta di "mailing-list" denominata **Cocoa Mailing List Archive** (http://cocoa.mamasam.com/). Ottimi tutorial si trovano anche su **CocoaDevCentral** (http://www.cocoadevcentral.com/), sempre in inglese. In italiano potete invece trovare risorse molto ben fatte su **macocoa** (http://www.macocoa.omitech.it/) e alcuni tutorial in italiano sulla pagina di programmazione di **Marco Coïsson** (http://homepage.mac.com/marco_coisson/Tutorial/index.html), la stessa da cui avete scaricato questo manuale. Potete inoltre rivolgere domande di carattere tecnico sulla programmazione (per Macintosh) in generale, e quindi anche su Objective-C e Cocoa, sull'ottimo **forum di Tevac** (in italiano), in particolare nella stanza dedicata ad AppleScript e alla Programmazione (http://forum.tevac.com/viewforum.php?f=28=).

Grazie a tutti per i consigli, gli apprezzamenti e il supporto. Grazie per aver avuto la forza e la pazienza di seguirmi fino a qui.