UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Progetto e Sviluppo di un'Applicazione Mobile per il Calcolo dell'H-Index

Daniele Cristofori

Tesi di Laurea

Relatore:
Prof. Riccardo Martoglia

Anno Accademico 2010/2011

RINGRAZIAMENTI

Ringrazio di cuore i miei familiari per il sostegno ricevuto in questi anni, il professor Martoglia per la sua grandissima disponibilità e pazienza e tutti i miei compagni di corso.

PAROLE CHIAVE

H-Index

Objective-C

iPhone

Apple

Publications

Indice

Introduzione **pag. 1**

I – Il Caso di studio **pag. 3**

1 Il sistema operativo iOS **pag. 4**

1.1 Introduzione iOS pag. 4
1.2 Design pattern fondamentali pag. 7
1.3 Oggetti chiave in iOS pag. 9
1.4 Ciclo di vita di un'applicazione pag. 11
1.4.1 La funzione main pag. 12
1.4.2 L'application delegate pag. 13
1.4.3 Stati di un'applicazione iOS pag. 14
1.4.4 Passaggio dal Not Running al Foreground pag. 16
1.4.5 Passaggio dal Foreground al Background pag. 17
1.4.6 Passaggio dal Background al Foreground pag. 18
1.4.7 Gestione delle interruzioni pag. 19
1.4.8 Schema dei passaggi di stato con relativi metodi pag. 20
1.4.9 Chiusura di un'applicazione pag. 22
1.4.10 Notifiche di sistema in Background pag. 22
1.4.11 Rinuncia al Background pag. 24

2 L'Objective-C **pag. 25**

2.1 Da C ad Objective-C pag. 25
2.2 Il primo iPhone e le WebApp pag. 26
2.3 L'iPhone SDK pag. 27
2.4 SDK per il firmware 3.0 pag. 27
2.5 iOS 4.0 e l'iPhone 4 pag. 28
2.6 Cosa offre l'SDK pag. 29
2.7 Ambiente di sviluppo pag. 29
2.8 Oggetti in Objective-C pag. 30
2.9 Dalla realtà al codice pag. 30
2.10 Le classi fondamentali di Cocoa Touch pag. 31
2.11 Gestione della memoria pag. 33
2.12 Le proprietà pag. 34
2.13 Programmazione multithreading pag. 35
2.14 Le categorie pag. 37

3 Standard ed algoritmi utilizzati pag. 39

3.1 Il linguaggio XPath pag. 39
3.1.1 Tipi di nodi pag. 41
3.1.2 Location Step e Location Path pag. 41
3.1.3 L'asse pag. 42
3.1.4 I predicati pag. 43
3.2 I parser HTML/XML disponibili pag. 44
3.3 L'Edit Distance pag. 47

II – Progetto dell'applicazione pag. 50

4 Progetto: tecniche e scelte pag. 51

4.1 Funzionalità applicazione pag. 51
4.2 Interazione tra classi pag. 54
4.3 Scelta del parser da utilizzare pag. 55
4.4 Gestione del database pag. 56
4.5 Problema: confronto dei tag. Soluzione: algoritmo Edit Distance pag. 58
4.6 Estrapolazione intelligente pubblicazioni da homepage pag. 59
4.7 Esempi di ricerca pag. 60

III – Implementazione pag. 62

5 Particolarità d'implementazione pag. 63

5.1 Classe PublicationsParser pag. 63
5.2 Classe DBPublication pag. 64
5.3 Classe FirstViewController pag. 67
5.4 Classe EditDistance pag. 68
5.5 Classe Singleton pag. 68
5.6 Classe TableFromGoogleScholarViewController pag. 69
5.7 Classe TableFromHomepageViewController pag. 72

Conclusioni e sviluppi futuri pag. 74

Elenco delle figure

1.1	Stratificazione iOS	pag. 4
1.2	Model-View-Controller	pag. 7
1.3	Oggetti iOS	pag. 9
1.4	Ciclo di vita applicazione iOS	pag. 12
1.5	Stati applicazione iOS	pag. 14
1.6	Apertura dell'applicazione	pag. 16
1.7	Dal Foreground al Background	pag. 17
1.8	Dal Background al Foreground	pag. 18
1.9	Gestione interruzioni	pag. 19
1.10	Diagramma degli stati con metodi	pag. 20
3.1	Albero dei nodi	pag. 39
4.1	Interazione classi	pag. 54

Introduzione

Il grande interesse personale per gli iPhone ma soprattutto per le applicazioni iOS, è stato determinante nella scelta di questo progetto.
Partendo dai fondamenti principali del sistema operativo iOS e dalle basi del linguaggio di programmazione Objective-C, si è provveduto a creare un'applicazione che calcoli per qualsiasi scienziato l'H-Index, un parametro ideato nel 2005 grazie a Jorge E. Hirsch, che quantifica la prolificità e l'impatto dei lavori dello scienziato di riferimento.
Il calcolo dell'H-Index è stato realizzato prevalentemente grazie ai risultati restituiti dalla piattaforma di Google Scholar, un motore di ricerca che permette di ricercare documenti, libri, pubblicazioni e studi più rilevanti nel campo della ricerca accademica mondiale.
Tenendo presente che una parte dei risultati restituiti da Google Scholar potrebbe essere erronea per via degli omonimi, il valore aggiunto dell'applicazione sta nel fatto di riuscire a considerare, nel calcolo dell'H-Index, solo quelle pubblicazioni che sono state effettivamente pubblicate dallo scienziato in questione. Per fare questo è stato necessario estrarre tutte le pubblicazioni appartenenti all'autore in questione dalla sua homepage.

La struttura di questa tesi è organizzata in cinque capitoli il cui contenuto verrà di seguito illustrato.
Il capitolo 1 comprende gli studi fatti sul sistema operativo iOS puntando l'attenzione sul ciclo di vita di un'applicazione.
Il capitolo 2 invece comprende gli aspetti più significativi del linguaggio di programmazione Objective-C. In particolare vengono descritti quegli aspetti che sono stati poi applicati al progetto vero e proprio.
Il capitolo 3 comprende gli algoritmi e gli standard utilizzati nella fase di progetto.
Il capitolo 4 comprende le tecniche e le scelte adottate durante lo sviluppo dell'applicazione.

Il capitolo 5 comprende particolari dettagli a livello d’implementazione.

Parte I

Il Caso di studio

Capitolo 1

Il sistema operativo iOS

In questo capitolo vengono descritti gli aspetti salienti del sistema operativo iOS: i livelli fondamentali, i design pattern e gli oggetti più importanti, il ciclo di vita di un'applicazione tipo.

1.1 Introduzione iOS

iOS è il sistema operativo sviluppato dalla Apple per i dispositivi iPhone, iPod touch e iPad. iOS, come si può vedere in figura 1.1, è costituito da 4 livelli principali: il **Core OS layer**, il **Core Services layer**, il **Media layer** ed il **Cocoa Touch layer**.

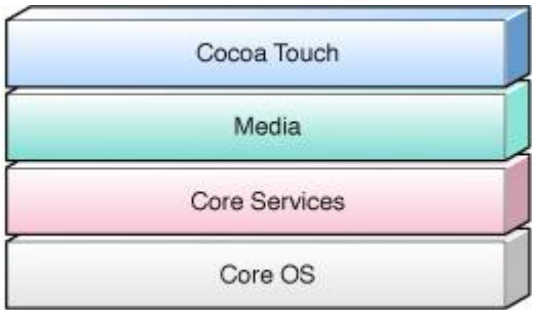


Fig. 1.1 Stratificazione iOS

1. **Core OS** è il livello più basso e fornisce le interfacce per i servizi di più basso livello. Si può accedere alle sue interfacce tramite la libreria *LibSystem*. I servizi appartenenti a questo livello sono: l'allocazione di memoria, informazioni locali, sistema di input e output, il networking, l'accesso ai file di sistema, i drivers, i servizi DNS. Lo strato Core OS comprende il kernel, i driver e l'interfaccia di base del sistema operativo.
2. Il livello **Core Services** fornisce tutti i servizi fondamentali che necessitano a tutte le applicazioni. Tra tali servizi si può elencare la gestione dei contatti (*Address Book*), la gestione dei tipi di dati (*Core Foundation*), i servizi di

localizzazione (*Core Location*), i servizi di rete (*CFNetwork*), la gestione della sicurezza (*Security*), la libreria *SQLite* per la gestione di una base di dati (*SQLite*) e il supporto all'XML (*XML Support*).

2.1. Il framework **Address Book** (*AddressBook.framework*) consente alle applicazioni di accedere ai contatti salvati sul dispositivo. Il framework Addredd Book UI (*AddressBookUI.framework*) fornisce un'interfaccia grafica per accedere ai contatti utente.

2.2. Il framework **Core Foundation** (*CoreFoundation.framework*) è un insieme di interfacce che forniscono la gestione dei dati primitivi. Questo framework fornisce supporto per i tipi di dato Collection (arrays, sets, ecc.), bundle, gestione stringhe, gestione delle date e del tempo, gestione di blocchi di dati grezzi, gestione delle preferenze, manipolazione di stream e di URL, supporto a Thread ed all'avvio del Loop, comunicazione di porte e Socket.

2.3. Il framework **Core Location** (*CoreLocation.framework*) permette di trovare in maniera precisa la posizione geografica dell'utente usando vari sistemi, tra cui il GPS.

2.4. Il framework **CFNetwork** (*CFNetwork.framework*) è un insieme di interfacce che gestiscono i protocolli di networking. Alcuni compiti propri di questo framework sono: uso di Socket BSD, creazione di connessioni protette SSL o TLS, DNS hosts, lavorare con server HTTP e HTTPS, lavorare con server FTP, pubblicare, risolvere, e cercare servizi Bonjour.

2.5. Il framework **Security** (*Security.framework*) è rivolto a quegli sviluppatori che vogliono aumentare la sicurezza dei propri dati nelle applicazioni. Questo framework fornisce interfacce per la gestione dei certificati a chiave pubblica o privata.

2.6. La libreria **SQLite** fornisce agli sviluppatori la possibilità di salvare i propri dati in un database SQL, senza dover accedere in remoto ad un altro database.

2.7. La libreria **libXML2** fornisce il supporto alla manipolazione di pagine XML e permette di analizzare o scrivere pagine XML o HTML.

3. Il livello **Media** fornisce strumenti per la gestione della grafica, dell'audio e dei video.
   3.1. Per quel che concerne la grafica, il framework ***UIKit*** fornisce tutti gli strumenti per gestire l'interfaccia grafica dell'applicazione (*UIButton*, *UIView*, ecc.). Tuttavia questo framework può risultare insufficiente per gli sviluppatori e per questo sono state sviluppate delle nuove librerie per gestire al meglio e con più dettagli ogni aspetto grafico 2D e 3D dell'applicazione: *OpenGLES.framework* che fornisce strumenti per il disegno di contenuti 2D e 3D fornendo un elevato frame rate utile nello sviluppo dei giochi, *QuartzCore.framework* che contiene le interfacce *Core Animation* per lo sviluppo di animazioni avanzate, *CoreGraphics.framework* che contiene le interfacce per il disegno Quartz 2D (disegno base, rendering, gradienti, immagini, colori, ecc.).
   3.2. Le tecnologie audio comprendono la capacità di riprodurre e registrare audio ad alta qualità e di gestire le funzionalità di vibrazione. La principale libreria è **Core Audio** che consente di manipolare, registrare, generare e riprodurre audio e consente anche di gestire la vibrazione del dispositivo.
   3.3. Attraverso il framework **Media Player** (*MediaPlayer.framework*) è possibile riprodurre file video con estensione .mov .mp4 .m4v .3gp.
4. Il livello **Cocoa Touch** è il più importante di tutti. Esso comprende i framework UIKit e Foundation (*UIKit.framework* e *Foundation.framework*) che gestiscono le applicazioni (Application Management), la grafica, gli eventi, l'interfaccia utente, gli oggetti che controllano il sistema, il supporto per testo e contenuti web, l'accelerometro, la fotocamera integrata, la libreria fotografica dell'utente.

1.2 Design pattern fondamentali

I Design Patterns descrivono soluzioni a problemi specifici per quanto riguarda la progettazione ad oggetti di un sistema software e riportano soluzioni già sviluppate, che si sono evolute nel tempo, favorendo quindi il riuso e la flessibilità del codice.

Lo scopo dei design pattern è quindi quello di risolvere un problema che si può presentare in diverse situazioni durante la progettazione e lo sviluppo del software.

I design pattern usati dalle applicazioni iOS sono principalmente 6:

1. **Model-View-Controller**: E' un pattern architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented. Questo pattern, come si può vedere in figura 1.2, è basato sulla separazione del codice in 3 aree funzionali indipendenti tra di loro:

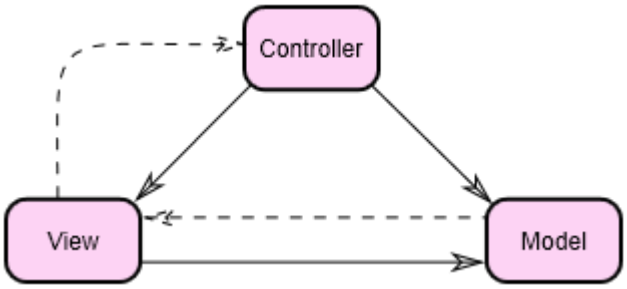


Fig 1.2 Model-View-Controller

   il **model** definisce il motore sottostante ed è responsabile dell'integrità dei dati;

   la **view** visualizza i dati contenuti nel model (definisce l'interfaccia utente) e si occupa dell'interazione con utenti e agenti. Non ha alcuna conoscenza esplicita dell'origine dei dati visualizzati;

   il **controller** funge da ponte tra il model e la view coordinando il passaggio dei dati. In particolare riceve i comandi dall'utente (in genere attraverso la view) e li attua modificando lo stato degli altri due componenti.

   Questo schema implica anche la tradizionale separazione fra l'application logic a carico del controller e del model, e l'interfaccia utente a carico della view.

I dettagli delle interazioni tra questi tre oggetti software dipendono dalle tecnologie usate (linguaggio di programmazione, eventuali librerie, ecc.) e dal tipo di applicazione (per esempio un'applicazione web o un'applicazione desktop).

2. **Block objects**: I block objects sono un modo conveniente per l'incapsulamento del codice e dello stack frame corrente in modo che possa essere eseguito in un successivo momento. Il supporto ai block objects è disponibile dalla versione 4.0 dell'iOS.
3. **Delegation**: L'idea alla base della delega è che un oggetto può avere un singolo delegato che richiamerà quando si verificano determinati eventi. Dal punto di vista del delegato è una sorta di callback o schema di notifica: "Chiamami quando succede questo". Dal punto di vista del delegante è una sorte di scarico di responsabilità: "Non so che cosa deve accadere quando si verifica questo evento, perciò pensaci tu".
4. **Target-action**: I componenti dell'interfaccia grafica usano il design pattern target-action per notificare l'applicazione delle interazioni utente. Quando l'utente interagisce con un componente, come ad esempio la pressione di un bottone, il componente manda un messaggio (the action) ad un oggetto specificato dall'utente (the target). Subito dopo aver ricevuto il messaggio, l'oggetto target può rispondere in maniera appropriata.
5. **Managed memory model**: L'Objective-C usa una strategia detta reference counting per determinare quando rilasciare gli oggetti dalla memoria. Quando viene creato un oggetto, esso assume un reference count pari a 1. Gli oggetti possono inoltre usare i metodi retain, release, o autorelease per incrementare o decrementare il reference count in maniera appropriata. Quando il reference count di un oggetto raggiunge lo 0, l'Objective-C a runtime chiama delle routines di cleanup per deallocare l'oggetto in questione.
6. **Threads and concurrent programming**: Tutte le versione dell'iOS supportano i thread e la programmazione concorrente. Dalla versione 4 dell'iOS, le applicazione possono anche usare il Grand Central Dispatch (GCD) per eseguire tasks in maniera concorrente.

1.3 Oggetti chiave in iOS

Dal momento in cui l'applicazione viene lanciata fino a quando essa viene terminata, il framework *UIKit* controlla quasi tutti i suoi comportamenti. Per esempio un'applicazione iOS riceve continuamente eventi dal sistema e deve sempre rispondere ad essi. Compito dell'oggetto *UIApplication* è ricevere gli eventi, ma rispondere ad essi è compito del codice scritto dallo sviluppatore. La figura 1.3 mostra gli oggetti che più comunemente si trovano nelle applicazioni iOS. Il ruolo di ognuno di questi oggetti viene spiegato di seguito.

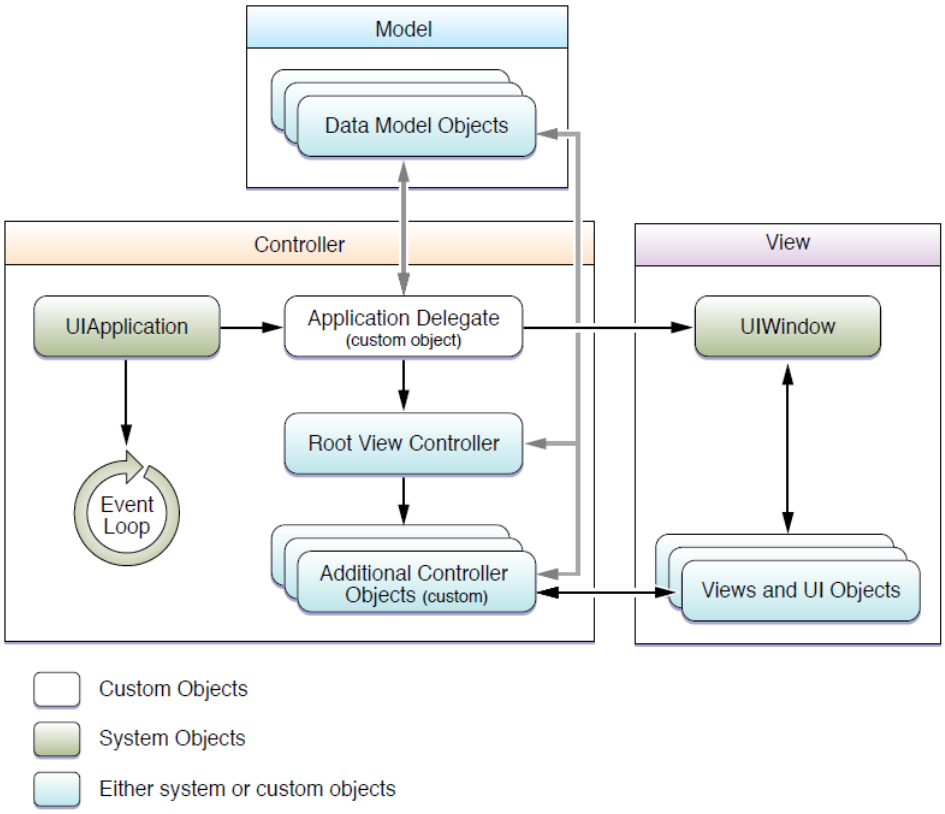


Fig. 1.3 Oggetti iOS

1. ***UIApplication* object**: L'oggetto UIApplication fa partire e coordina l'event loop dell'applicazione altri aspetti di alto livello. Viene usato questo oggetto il più delle volte per configurare diversi aspetti riguardo il lancio dell'applicazione. Il codice appartenente al livello applicativo

risiede nell'oggetto delegato, che lavora in tandem con l'oggetto *UIApplication*.

2. **Application delegate object**: Il delegato è un oggetto tipicamente incluso in un *nib* file. Il lavoro principale di questo oggetto è inizializzare l'applicazione e presentarla sullo schermo. L'oggetto *UIApplication* notifica questo oggetto quando occorre uno specifico evento, come quando l'applicazione deve essere interrotta (per l'arrivo di un messaggio) o deve essere spostata in background (perché l'utente ha schiacciato il pulsante Home).
3. **Data model objects**: Gli oggetti data model salvano il contenuto dell'applicazione. Per esempio un'applicazione bancaria potrebbe dover salvare un database contenente transazioni finanziarie, ecc.
4. **View controller objects**: Gli oggetti view controller gestiscono ogni schermata di un'applicazione iPhone. Tipicamente questo comporta la creazione di viste per presentare la schermata e per gestire l'interazione tra viste e le parti dell'applicazione relative al data model. La classe *UIViewController* è la classe base per tutti gli oggetti view controller. Essa fornisce funzionalità di default per l'animazione e la visualizzazione delle viste, trattando le rotazioni del dispositivo, e molti altri comportamenti di sistema standard. *UIKit* e altri frameworks definiscono anche altre classi view controller per la gestione di interfacce di sistema standard, come l'interfaccia di navigazione o l'image picker.
5. ***UIWindow* object**: L'oggetto *UIWindow* coordina la presentazione di una o più viste sullo schermo del dispositivo o su un display esterno. La maggior parte delle applicazioni hanno una sola window il cui contenuto è provvisto da una o più viste. Un applicazione cambia il contenuto della windows tramite la sostituzione del set corrente di viste (tipicamente con l'aiuto di un oggetto view controller). Le windows sono anche responsabili della consegna degli eventi alle viste e alla gestione dei loro view controllers.

6. **Views, controls, and layers**: Views e controls provvedono alla rappresentazione dell'applicazione sullo schermo dell'iPhone. Una view è un oggetto che disegna il contenuto dell'applicazione in un'area rettangolare. Altro compito della view è rispondere agli eventi generati all'interno di quest'area rettangolare. I controls sono un tipo specializzato di vista responsabile dell'implementazione degli oggetti dell'interfaccia grafica come bottoni e aree di testo. Il framework *UIKit* fornisce viste standard per rappresentare differenti tipi di contenuto. Si possono definire anche viste personalizzate sottoclassando direttamente *UIView* o sue sottoclassi. I layer objects sono oggetti dati che rappresentano il contenuto dell'interfaccia. Le viste usano i layer objects intensivamente per presentare il loro contenuto.

Gli oggetti all'interno dell'applicazione formano un ambiente assai complesso. Come si può vedere in figura 1.3 la maggior parte degli oggetti in un'applicazione sono parzialmente o del tutto personalizzabili.

1.4 Ciclo di vita di un'applicazione

Il ciclo di vita delle applicazioni iOS è costituito dalla sequenza di eventi che occorre tra il loro lancio e la loro terminazione. Nell'iOS l'utente lancia l'applicazione premendo la sua icona sullo schermo. Subito dopo la pressione del tasto, il sistema mostrerà l'interfaccia grafica e procederà con il lancio dell'applicazione chiamando la sua funzione *main*. Da questo punto in avanti, il grosso del lavoro di inizializzazione è delegato al framework *UIKit*, che carica il *nib* file (contenente l'interfaccia grafica) ed entra nell'event loop.
La figura 1.4 rappresenta un ciclo di vita semplificato di un'applicazione iOS. Il diagramma in questione mostra la sequenza di eventi che occorre tra il momento in cui viene lanciata l'applicazione e il momento in cui un'altra applicazione viene lanciata. Nei momenti chiave durante il ciclo di vita dell'applicazione, *UIKit*

manda continuamente messaggi al delegate object per comunicargli a mano a mano cosa sta succedendo. Durante l'event loop, è *UIKit* che si fa carico di mandare eventi agli event handlers (view e view controllers).
Prima della versione 4.0 dell'iOS, quando l'utente chiudeva l'applicazione, essa veniva rimossa dalla memoria. Questo comportava un modello molto semplificato ma d'altro canto un rilancio della stessa applicazione risultava più costoso in termini di tempo. Mantenere l'applicazione in memoria aggiunge sicuramente un certo grado di complessità al ciclo di vita ma al tempo stesso fornisce un significativo beneficio in termini di prestazioni sia per l'applicazione che per l'utente.

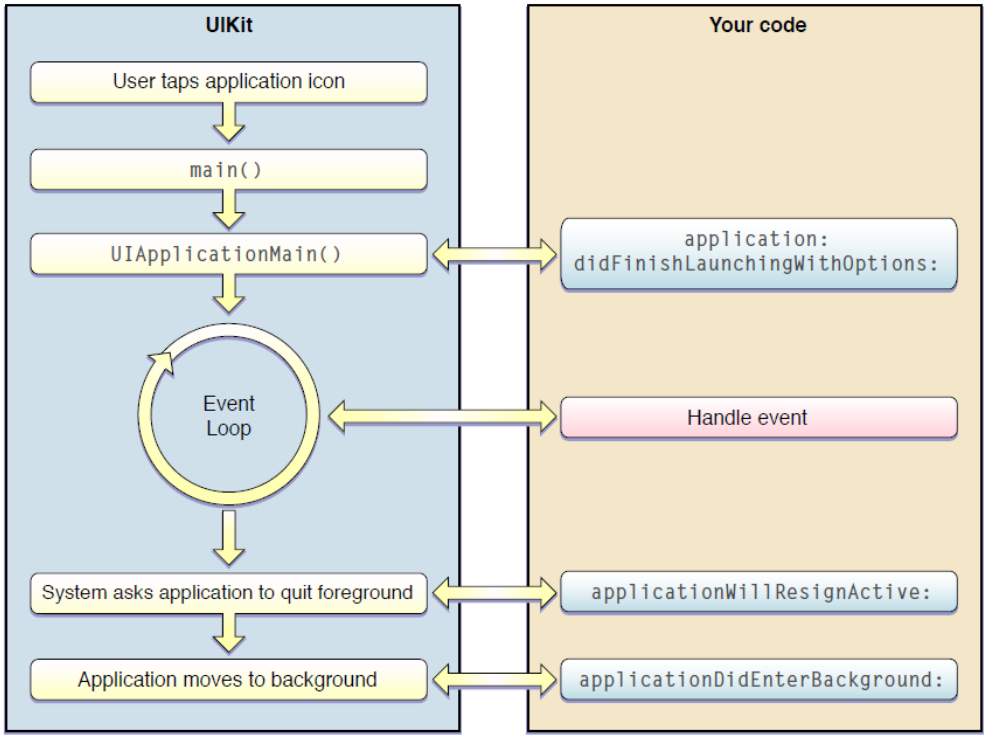


Fig. 1.4 Ciclo di vita applicazione iOS

1.4.1 La funzione main

Come per le applicazioni C-like la prima funzione che viene chiamata al lancio dell'applicazione è la funzione *main*.

Compito del *main* è passare il controllo al framework *UIKit*. Ogni nuovo progetto creato in Xcode presenta la funzione main come di seguito:

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

int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

La signature alla riga 3 è una tipica funzione *main()* del C, progettata per essere richiamata dalla riga di commando con un numero variabile di argomenti.
La riga 4 imposta un pool a rilascio automatico per la gestione della memoria e la riga 6 lo rilascia al termine dell'applicazione.
Alla riga 5 viene richiamata la funzione *UIApplicationMain* per avviare il ciclo di eventi principale e far partire l'applicazione. I primi due argomenti corrispondono agli argomenti della riga di comando (se ve ne sono), mentre il terzo e il quarto indicano la classe principale dell'applicazione e la sua delegata, una classe del progetto che gestisce gli eventi del ciclo di vita dell'applicazione. Se questi ultimi argomenti sono *nil*, come in questo caso, *UIKit* assume di dover caricare l'applicazione da un file *nib*.

1.4.2 L'application delegate

Il concetto di delegato è molto comune ed importante nei linguaggi Object Oriented. Per capire bene il concetto di delegato, supponiamo di avere un oggetto che non compie direttamente una data operazione ma che invece delega tale esecuzione ad un altro oggetto. Quest'ultimo oggetto, quello che di fatto esegue la funzione, è un delegato, una sorta di "aiutante" del primo oggetto.
In Objective-C è presente un delegato molto importante: il cosiddetto "Application delegate" che esegue delle funzioni in vece della classe *UIApplication*.
Ogni applicazione iOS gestisce un'unica istanza della classe *UIApplication*.

Tale istanza è fondamentale poiché è in grado, tra le altre cose, di instradare gli input alle opportune classi controller.
Durante l'esecuzione di un'applicazione UIApplication chiama, in momenti ben precisi, specifici metodi delegati (se c'è un opportuno delegato e se tale delegato implementa tali metodi).
Questa tipologia di delegazione permette di implementare metodi e codice utilizzabili dagli applicativi senza dover creare delle sottoclassi di *UIApplication*.

1.4.3 Stati di un'applicazione iOS

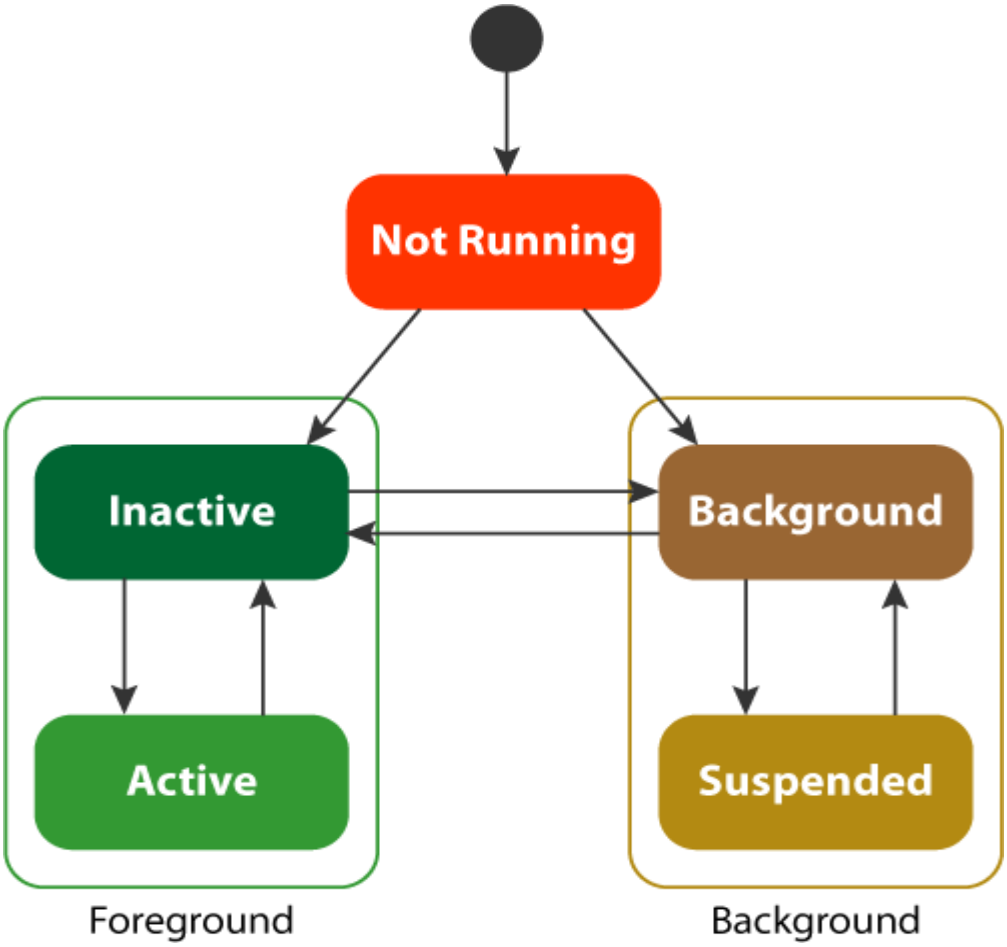


Fig. 1.5 Stati applicazione iOS

Le applicazioni eseguite dalla versione 4.0 dell'iOS, come si può vedere in figura 1.5, possono trovarsi in diversi stati durante il loro ciclo di vita. Per tutte quelle

applicazioni che sono compilate per la versione 3.2 o precedente dell'iOS, non presentano gli stati di Background e di Suspended.

1. **Not running**: l'applicazione non è ancora stata lanciata, oppure è andata in crash o è stata terminata dal sistema operativo.
2. **Inactive**: l'applicazione è in esecuzione in Foreground, ma non sta ricevendo eventi. Potrebbe comunque eseguire del codice. L'applicazione si trova in questo stato solo per poco tempo eccetto quando il dispositivo è in lock o in generale quando compare una finestra in sovrimpressione (overlay window) ad esempio per l'arrivo di una chiamata o un messaggio sms.
3. **Active**: l'applicazione è in esecuzione in Foreground e sta ricevendo eventi. Questo è lo stato "normale" in cui si trova l'applicazione quando ne vediamo l'interfaccia utente.
4. **Background**: l'applicazione è in Background e sta eseguendo del codice, ma non riceve eventi utente. La maggior parte delle applicazioni entra in questo stato per un breve periodo di tempo prima di passare allo stato di Suspended. Non essendo visibile, un'applicazione in Background non deve aggiornare l'interfaccia grafica. Questo stato è disponibile solo a partire da iOS 4.0 sui dispositivi che supportano il multitasking. Se questo stato non è disponibile, l'applicazione viene terminata passando nello stato di Not Running.
5. **Suspended**: l'applicazione è in Background ma non sta eseguendo codice, dunque è sostanzialmente "congelata" e non sta facendo in alcun modo uso della CPU. Se per qualunque motivo il sistema operativo necessita di liberare della memoria, le applicazioni che si trovano in questo stato potrebbero essere terminate (cioè poste nello stato Not Running) senza alcuna notifica e a completa discrezione del sistema. Questo stato è disponibile solo a partire da iOS 4.0 sui dispositivi che supportano il multitasking. Se questo stato non è disponibile, l'applicazione viene terminata passando nello stato di Not Running.

Una ed una sola applicazione alla volta può essere in Foreground, ovvero negli stati Active o Inactive.

1.4.4 Passaggio dal Not Running al Foreground

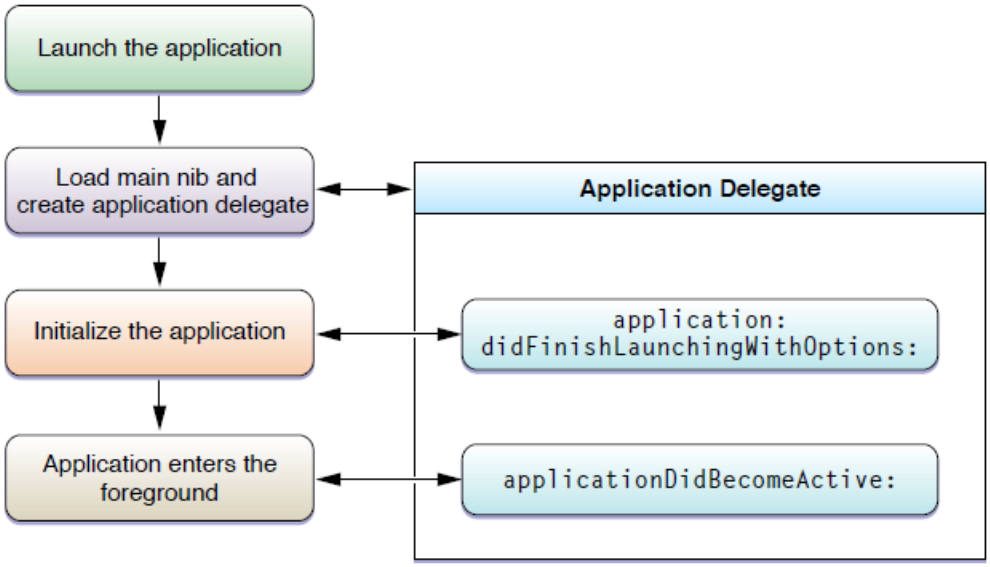
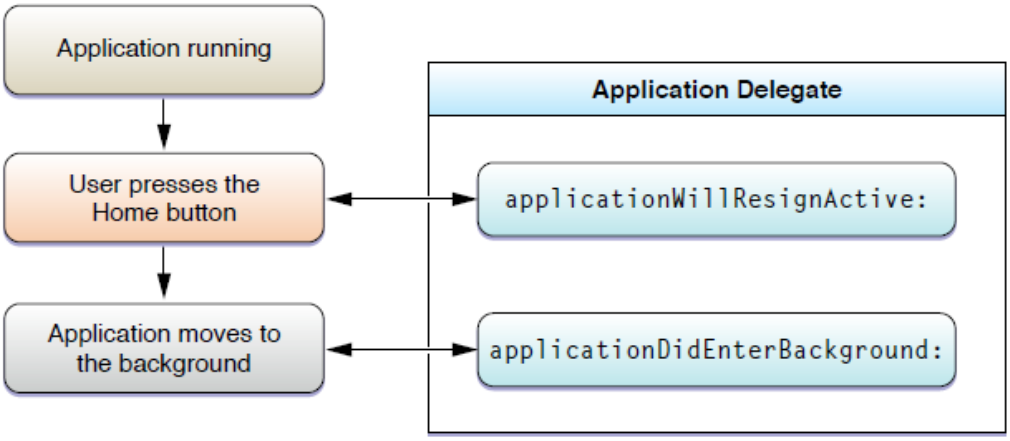


Fig. 1.6 Apertura dell'applicazione

Il passaggio tra i diversi stati è regolato dal sistema in risposta agli eventi utente o a eventi di sistema (come la ricezione di una telefonata o di un SMS).
Supponendo di aver appena fatto il reboot del dispositivo, l'applicazione si trova nello stato Not Running perché non è ancora stata lanciata.
Quando si preme sull'icona dell'applicazione nella Home Screen, come mostrato in figura 1.6, il sistema operativo carica l'immagine di apertura (*Default.png*) e poi lancia la funzione *main()*. All'interno del *main()* viene chiamata la funzione *UIApplicationMain()* che si occupa di creare l'oggetto *UIApplication*, caricare il main *nib* file e instanziare il delegato dell'applicazione.
Terminato il setup iniziale, l'applicazione passa nello stato Inactive con la chiamata del metodo *application:didFinishLaunchingWithOptions:* sul delegato dell'applicazione. Si usa questo metodo per inizializzare l'applicazione e prepararla per l'esecuzione. Questo metodo ha circa 5 secondi per terminare.

Dopo che il metodo *application:didFinishLaunchingWithOptions:* ha terminato la sua esecuzione viene avviato l'event loop, che riceve gli eventi (es: tap dell'utente sullo schermo) e si occupa di inviarli a chi li deve gestire. L'applicazione diventa quindi Active e può ricevere gli eventi e aggiornare l'interfaccia grafica. Nel passaggio dallo stato Inactive ad Active viene chiamato il metodo *applicationDidBecomeActive:* sul delegato dell'applicazione. Si usa questo metodo per riabilitare o abilitare tutto ciò che ha a che fare con la presentazione grafica e l'interazione utente.
L'applicazione è ora in esecuzione e l'utente può interagirvi.

1.4.5 Passaggio dal Foreground al Background



1.7 Dal Foreground al Background

Quando l'utente preme sul tasto Home, a partire da iOS 4.0, come mostrato in figura 1.7, l'applicazione torna nello stato Inactive, chiamando il metodo *applicationWillResignActive:* sul delegato dell'applicazione. Si usa questo metodo per disabilitare tutto ciò che ha a che fare con la presentazione grafica e l'interazione utente.
Dopodiché l'applicazione passa rapidamente nello stato Background. A questo punto viene chiamato sul delegato dell'applicazione il metodo *applicationDidEnterBackground:*, usato principalmente per liberare tutte le

risorse che possono essere ricreate in seguito, salvare lo stato, chiudere le connessioni. In tal modo si riduce la memory footprint dell'applicazione e di conseguenza le possibilità che possa essere terminata dal sistema. Questo metodo ha circa 5 secondi per terminare.
Infine l'applicazione passa allo stato Suspended in cui è sostanzialmente congelata.
Nello stato Suspended molti oggetti, inclusi i view controller, le view e gli altri oggetti creati dall'utente, rimangono in memoria. Tuttavia, il sistema rilascia alcuni degli oggetti usati dietro le quinte, tra cui i riferimenti alle immagini nella cache e alcuni oggetti usati da *Core Animation*.

1.4.6 Passaggio dal Background al Foreground

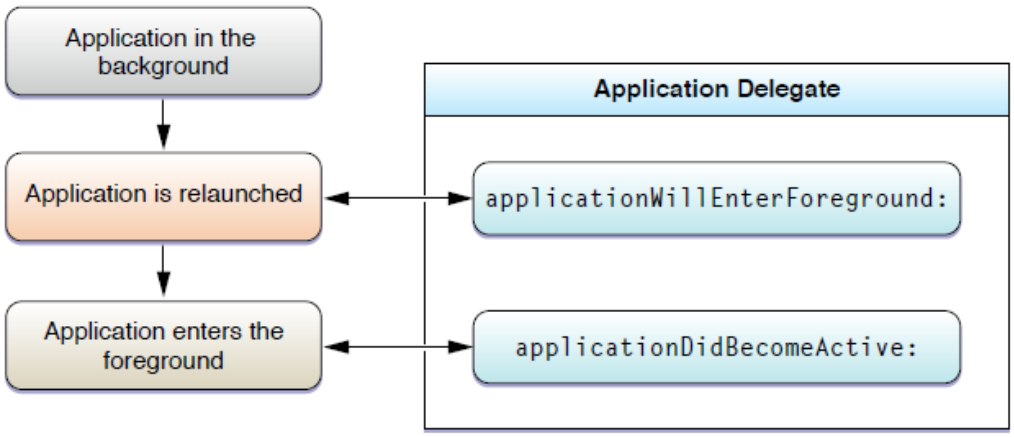


Fig. 1.8 Dal Background al Foreground

Quando l'utente rilancia l'applicazione, ad esempio premendo sulla sua icona nella Home Screen o nella barra del multitasking, come mostrato in figura 1.8, il sistema la sposta nello stato Inactive chiamando il metodo *applicationWillEnterForeground:*. Questo metodo viene usato per ricreare tutto ciò che era stato rilasciato nel metodo *applicationDidEnterBackground:*. Dopodiché lo stato dell'applicazione passa rapidamente nello stato Active chiamando il metodo *applicationDidBecomeActive:* descritto precedentemente.

Questo passaggio, per il fatto che l'applicazione era già congelata, è molto rapido (fast app switching).

1.4.7 Gestione delle interruzioni

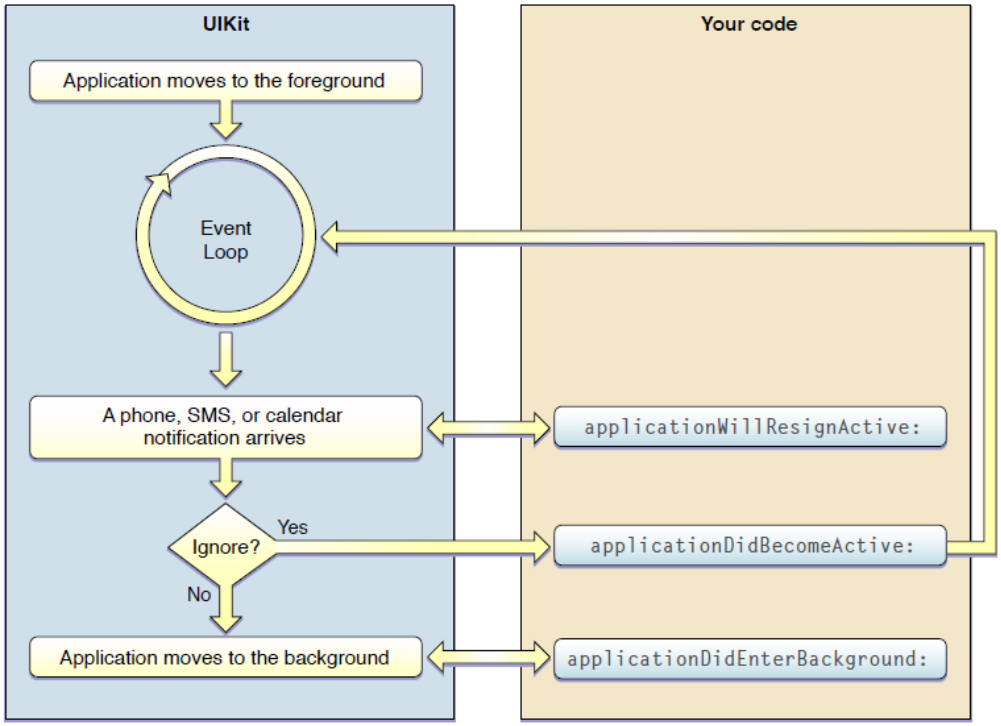


Fig. 1.9 Gestione interruzioni

Quando un'applicazione che si trova nello stato Active viene interrotta a causa della ricezione di una telefonata, di un SMS o di un avviso di calendario, viene, come mostrato in figura 1.9, temporaneamente spostata nello stato Inactive attraverso la chiamata del metodo *applicationWillResignActive:*. Rimarrà in questo stato fino a quando l'utente non decide di ignorare o accettare l'interruzione.

Se l'utente ignora l'interruzione, l'applicazione torna nello stato Active chiamando quindi il metodo *applicationDidBecomeActive:*. Se invece accetta l'interruzione, l'applicazione viene spostata nello stato Background e viene chiamato il metodo *applicationDidEnterBackground:*. Nel caso in cui

l'applicazione sta eseguendo su una versione antecedente alla 4.0 su un dispositivo che non supporta il multitasking, viene chiamato il metodo *applicationWillTerminate:* che termina l'applicazione rilasciandone completamente la memoria.
Terminata l'interruzione, in alcuni casi il sistema rilancia in automatico l'applicazione che si trovava in Foreground. Capita, ad esempio, quando l'utente riceve una chiamata e poi mette giù.

1.4.8 Schema dei passaggi di stato con relativi metodi

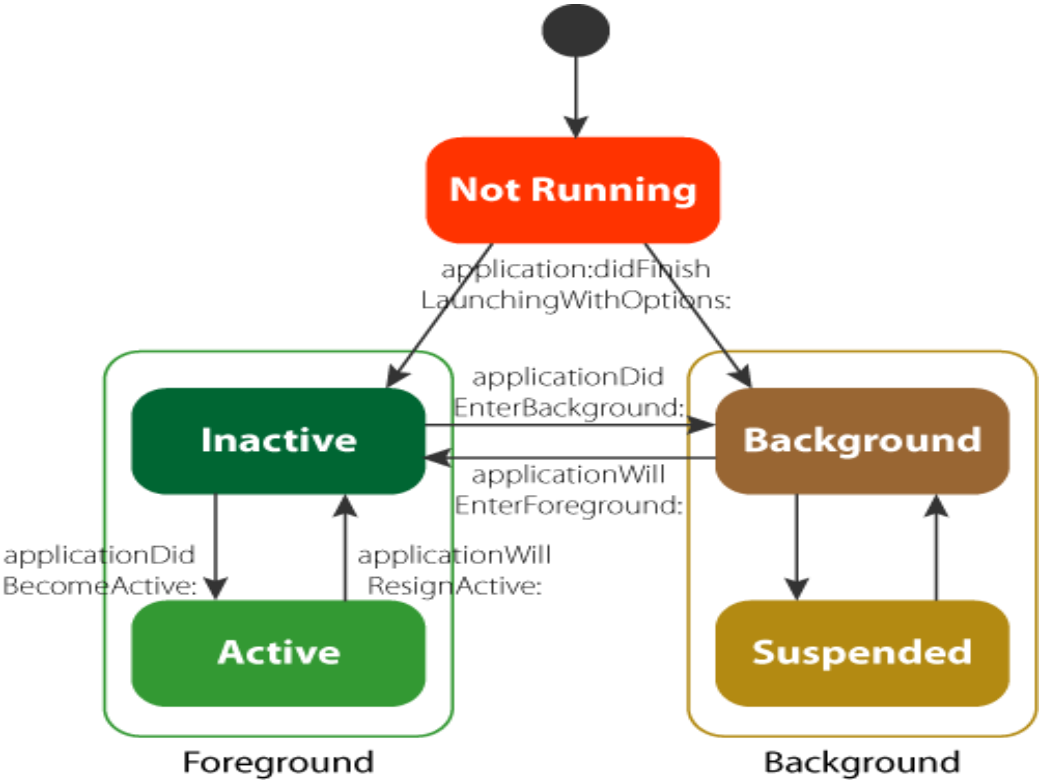


Fig. 1.10 Diagramma degli stati con metodi

Le transizioni tra Background e Suspended, come viene evidenziato in figura 1.10, avvengono senza che venga chiamato alcun metodo sul delegato. Stesso discorso quando l'applicazione è in background e l'utente la chiude dalla barra del multitasking (o quando, per mancanza di memoria, il sistema decide di terminarla). In questi casi il sistema operativo invia semplicemente un segnale

SIGKILL che termina immediatamente il processo. Ecco perché è fondamentale salvare lo stato dell'applicazione nel metodo *applicationDidEnterBackground:*.

Infine si può dire che ad ogni metodo del delegato corrisponde una notifica. Ecco l'elenco:

Metodo del delegato	**Notifica corrispondente**
application:didFinishLaunchingWithOptions:	UIApplicationDidFinishLaunchingNotification
applicationWillResignActive:	UIApplicationWillResignActiveNotification
applicationDidBecomeActive:	UIApplicationDidBecomeActiveNotification
applicationDidEnterBackground:	UIApplicationDidEnterBackgroundNotification
applicationWillEnterForeground:	UIApplicationWillEnterForegroundNotification

L'oggetto *UIApplication* dispone di una proprietà in sola lettura chiamata *applicationState* che restituisce lo stato dell'applicazione (Active, Inactive, etc.) e che permette allo sviluppatore di eseguire determinato codice in base allo stato in cui si trova l'applicazione.

1.4.9 Chiusura di un'applicazione

Benché le applicazioni passano negli stati di Background e Suspended, se è vera una delle seguenti condizioni, l'applicazione viene terminata e la memoria relativa ad essa viene liberata.

1. L'applicazione è compilata per una versione inferiore alla 4.0 dell'iOS.
2. L'applicazione è eseguita su un dispositivo che ha la versione dell'iOS inferiore alla 4.0.
3. Il dispositivo corrente non supporta il multitasking.
4. L'applicazione ha attiva la chiave *UIApplicationExitsOnSuspend* nel file *Info.plist*.

Quando l'applicazione è terminata dal sistema o dall'utente, viene chiamato il metodo *applicationWillTerminate:* appartenente al delegato dell'applicazione.
Si può usare questo metodo per salvare dati o informazioni di stato per poi riutilizzarli al rilancio dell'applicazione. Questo metodo ha circa 5 secondi per terminare. Se esso non ritorna in tempo, l'applicazione viene forzatamente terminata dal sistema e rimossa dalla memoria.
Il metodo *applicationWillTerminate:* non viene chiamato se l'applicazione è sospesa.
L'utente può terminare le applicazioni usando la barra del multitasking.
Inoltre se la memoria occupata diventa grande, il sistema può rimuovere le applicazioni senza alcuna notifica nel caso in cui l'applicazione rimossa si trovava nello stato di Suspended, oppure attraverso la chiamata al metodo *applicationWillTerminate:* in tutti gli altri casi.

1.4.10 Notifiche di sistema in Background

L'applicazione che si trova nello stato di Suspended non riceve eventi dal sistema.

Gli eventi più rilevanti però vengono memorizzati dal sistema e consegnati all'applicazione durante la sua ripresa.
Per evitare un sovraccarico di notifiche alla ripresa, il sistema concentra gli eventi in uno singolo.
La maggior parte degli eventi sono forniti in forma di oggetto notifica. Salvo diversa specifica gli eventi vengono consegnati indipendentemente dal fatto che l'applicazione ritorni in Foreground o in Background.

Evento	**Meccanismo di notifica**
Il codice segnala che una view necessita di essere ridisegnata	Chiamata a setNeedsDisplay o setNeedsDisplayInRect. Le chiamate su una delle viste sono conservate fino a quando l'applicazione non torna in Foreground. Questi eventi non sono consegnati alle applicazioni in background
Un accessorio è connesso o disconnesso	*EAAccessoryDidConnectNotification* *EAAccessoryDidDisconnectNotification*
Modifica dell'orientamento del dispositivo	UIDeviceOrientationDidChangeNotification In aggiunta a questa notifica i view controllers aggiornano le relative interfacce automaticamente
Cambio significativo di tempo	UIApplicationSignificantTimeChangeNotification
Modifica livello o stato di batteria	*UIDeviceBatteryLevelDidChangeNotification* *UIDeviceBatteryStateDidChangeNotification*
Modifica dello stato di prossimità	*UIDeviceProximityStateDidChangeNotification*
Modifica dello stato dei file protetti	*UIApplicationProtectedDataWillBecomeUnavailable* *UIApplicationProtectedDataDidBecomeAvailable*
Schermo esterno connesso o disconnesso	*UIScreenDidConnectNotification* *UIScreenDidDisconnectNotification*
Modifica al modo dello schermo	*UIScreenModeDidChangeNotification*

Modifica preferenza che l'applicazione espone attraverso le impostazioni di applicazione	*NSUserDefaultsDidChangeNotification*
Modifica al corrente linguaggio o luogo	*NSCurrentLocaleDidChangeNotification*

Quando l'applicazione riprende la sua normale attività, tutti gli eventi in coda vengono consegnati attraverso il main run loop. Questi eventi vengono messi immediatamente in coda e quindi consegnati prima di qualsiasi altro evento touch fatto dall'utente.

1.4.11 Rinuncia al Background

Se si desidera che un'applicazione non debba essere messa in background quando si esce basta aggiungere la chiave *UIApplicationExitsOnSuspend* del file *Info.plist* e impostare il valore su *YES*. In questo caso l'applicazione non entrerà mai in Background o in stati di Sospensione. Quando l'utente tocca il tasto home per chiudere l'applicazione, verrà chiamato il metodo *applicationWillTerminate:* e avrà circa 5 secondi per completare tutte le operazioni necessari.
Anche se la rinuncia di esecuzione in Background è fortemente sconsigliato, può essere preferibile in alcune condizioni, come ad esempio se l'applicazione consuma enorme quantità di memoria e non può facilmente rilasciarla senza troppi sforzi, oppure se il codice per il Background richiede uno sforzo di complessità significativo.

Capitolo 2
L'Objective-C

In questo capitolo vengono descritti gli aspetti principali del linguaggio di programmazione Objective-C.
Nella prima parte del capitolo viene descritta un po' di storia mentre nella seconda parte del capitolo vengono analizzati quegli aspetti che sono stati valutati fondamentali per lo sviluppo dell'applicazione.

2.1 Da C a Objective-C

Siamo nel 1972 quando il C vede la luce. A elaborarlo è Dennis Ritchie presso i laboratori di AT&T. Il linguaggio è un'evoluzione del B, creato da Ken Thompson qualche anno prima ed è usato in particolare per scrivere programmi per i sistemi UNIX. Il primo standard del linguaggio C è realizzato nel 1990 dall'American National Standards Institute ed esso prenderà il nome di ANSI C.
Nei primi anni Ottanta, Brad J. Cox e Tim Love alla Stepstone, compagnia software fondata nel 1983, progettano un nuovo linguaggio di programmazione basato sul C, che prenderà poi il nome di Objective-C. La più grande differenza rispetto al C è l'appartenenza ai linguaggi cosiddetti OOP (**O**bject **O**riented **P**rogramming) cioè orientati agli oggetti. Poiché il primo linguaggio orientato agli oggetti è SmallTalk, nato negli anni Settanta, il primo nome dell'Objective-C scelto da Cox e Love è SmallTalk-80.
Nel 1988, NeXT, giovane compagnia fondata da Steve Jobs, acquista il marchio Objective-C da Stepstone, l'azienda di Cox e Love. Nel frattempo, per l'Objective-C arriva il supporto ai tool di sviluppo della Free Software Foundation di GNU: è un passo fondamentale che permette a questo linguaggio di essere usato con maggiore semplicità e sicurezza; inoltre, grazie alla Free Software Foundation, si forma una community relativa a esso, dove trovare assistenza per problemi e dubbi.

La diffusione di questo linguaggio avviene tramite il primo framework creato da NeXT: NeXTSTEP. Successivamente, Apple acquista la NeXT, oltre ai tool di sviluppo per programmi di terze parti e il sistema operativo da esso elaborato, NeXTSTEP, preludio di Mac OS. Da questo momento in avanti l'Objective-C diventa il linguaggio di sviluppo ufficiale per Mac OS X.

2.2 Il primo iPhone e le WebApp

Dopo molti anni, arriviamo al 9 Gennaio 2007, quando Steve Jobs, CEO (Chief Executive Officer) di Apple, annuncia un rivoluzionario telefono, l'iPhone. Il nuovo dispositivo è in vendita dal 29 Giugno del 2007. Per oltre un anno Apple non permette di sviluppare applicazioni native per il suo smartphone/lettore MP3. L'unico modo di estendere le capacità dell'iPhone è usare le WebApp, che sono applicazioni web.

Le WebApp sono ancora disponibili, ma presentano diversi limiti nelle loro funzioni. Le applicazioni web possono essere usate solo tramite browser e solo se si è connessi a Internet. E' quindi esclusa la possibilità di utilizzarle anche quando il dispositivo non è online. I linguaggi utilizzati per queste applicazioni sono solo linguaggi web, quali JavaScript e HTML.

Vi sono poi pesanti limiti anche per quanto riguarda la loro natura: non si possono scrivere giochi complessi o in 3D e non è possibile sfruttare le caratteristiche che rendono unico l'iPhone, come l'accelerometro. Anche gli sviluppatori di WebApp, però, sono supportati da Apple: nel SafariDevCenter Mobile i web-developer possono consultare le linee guida per quanto riguarda l'interfaccia utente e altre caratteristiche delle applicazioni web.

Infine tutti gli sviluppatori possono inviare la propria applicazione a Apple e vederla pubblicata sul sito della Mela in maniera completamente gratuita.

2.3 L'iPhone SDK

Dopo quasi un anno dalla data di commercializzazione, il 6 Marzo 2008 Apple presenta un SDK (Software Development Kit) per iPhone. L'SDK è un insieme di programmi che consentono di sviluppare applicazioni native per iPhone. Le differenze con le WebApp sono notevoli e spingono molti sviluppatori a entrare nel mondo di iPhone. Le applicazioni create da qualunque sviluppatore possono essere messe in vendita nell'AppStore, un negozio virtuale, in cui ogni utente può visionare, scegliere e acquistare applicazioni. Questo nuovo metodo di vendita è accolto con successo straordinario dal pubblico, così da arrivare a oltre un milione di applicazioni scaricate in soli nove mesi e a 50000 presenti nello store. Restano comunque moltissime le limitazioni imposte da Apple:

- Ogni singola applicazione presente in AppStore deve essere controllata e approvata da Apple;
- Non si possono usare librerie esterne;
- Le API fornite non sono contemplate.

Questo sistema è disponibile per tutti gli utenti finali dall'11 Luglio 2008, data in cui è stato rilasciato un nuovo firmware per iPhone (versione 2.0) assieme a un nuovo dispositivo, la versione 3G di iPhone.

2.4 SDK per il firmware 3.0

Dopo circa un anno dalla presentazione della prima versione dell'SDK per iPhone, Apple presenta l'SDK per il firmware 3.0, disponibile dal 17 Giugno 2009. La nuova versione include molte novità, con 1000 nuove API a disposizione degli sviluppatori. Con questa release viene colmato anche un grosso difetto, ovvero la mancanza del Background, aggiungendo il supporto alle notifiche Push, un'alternativa sviluppata da Apple che permette appunto agli sviluppatori di creare una specie di background. Il Push permette al telefono di risparmiare memoria RAM e di rimanere performante, poiché l'applicazione è chiusa e rimane

solo un collegamento costante con il server Apple. Gli svantaggi però non mancano. Per cominciare è richiesta una connessione a Internet permanente per poter usufruire di queste notifiche. Alcune applicazioni, poi, non riescono a beneficiare del Push, per esempio quelle che eseguono un download: nel momento in cui si esce dall'applicazione, con il Push non si può proseguire lo scaricamento in alcun modo. L'altro svantaggio è per gli sviluppatori minori, obbligati ad avere server di grande portata per gestire i ping da inviare al server Apple, i quali successivamente verranno inoltrati al device dell'utente.

2.5 iOS 4.0 e l'iPhone 4

L'8 Aprile 2010, sempre Steve Jobs annuncia una nuova versione del firmware per iPhone: la versione 4.0. In questa release vengono introdotte molte novità, sia per gli sviluppatori, sia per gli utenti. La novità maggiore è il multitasking. Da questa release si possono infatti lasciare in background le applicazioni.
L'approccio è differente da quello che viene considerato normalmente multitasking: l'applicazione non rimane completamente attiva, ma lo sviluppatore può decidere ugualmente di sfruttarne alcune funzioni (come la localizzazione o la riproduzione musicale).
Il firmware 4.0 viene ribattezzato iOS 4.0 e viene rilasciato ufficialmente il 21 Giugno 2010 per iPhone 3G, iPhone 3GS e iPod Touch di seconda e terza generazione. Viene annunciato infine un'altra novità: il nuovo iPhone 4. Quest'ultimo porta notevoli novità relative all'hardware, in particolar modo il nuovo schermo, rinominato Retina Display: la risoluzione del nuovo iPhone è quattro volte la risoluzione dei precedenti device, con un livello di dettaglio incredibile. Questa nuova caratteristica "obbliga" gli sviluppatori a riadattare tutte le applicazioni a una risoluzione maggiore, anche se le vecchie applicazioni funzionano normalmente con questo nuovo telefono.

2.6 Cosa offre l'SDK

Per iniziare a sviluppare per iPhone si deve scaricare l'SDK fornito da Apple che comprende applicazioni per la scrittura di codice, per la realizzazione di interfacce d'uso (GUI) e per monitorare le prestazioni delle applicazioni su iPhone. E' presente anche un software, chiamato iPhone Simulator, che permette di testare le applicazioni create direttamente nel Macintosh. Tramite iPhone Simulator si possono provare applicazioni in tutti i diversi device Apple: iPhone, iPad e iPhone 4. L'insieme delle applicazioni, l'SDK, è completamente gratuito; con questi software si ha tutto il necessario per poter scrivere applicazioni complete da provare su iPhone Simulator. Se lo sviluppatore desidera provarle sul proprio dispositivo o desidera pubblicarle su AppStore, deve registrarsi all'iPhone Developer Program Standard a un costo di 99$.

2.7 Ambiente di sviluppo

Il programma principale per poter sviluppare le proprie applicazioni per iPhone e iPad è XCode. Lo scopo di questo software è permettere al programmatore di scrivere codice e compilarlo.
Un importantissimo tool fornito da Apple insieme ad XCode è Interface Builder, utilizzato per lo sviluppo dell'interfaccia d'uso delle applicazioni. Permette di collocare pulsanti, tabelle, immagini e molto altro ancora nell'applicazione da creare.
Un'altra applicazione utile è Instruments, adoperata per controllare le prestazioni della nostra applicazione, così da verificarne l'utilizzo e correggere eventuali problemi legati all'uso eccessivo di risorse del dispositivo.
L'ultima applicazione di fondamentale importanza è iPhone Simulator, che permette di avere un iPhone (o iPad) virtuale nel desktop. Permette così di testare le applicazioni senza dover ricorrere all'uso di un vero dispositivo.

2.8 Oggetti in Objective-C

Contrariamente ai linguaggi procedurali come il C, i linguaggi ad oggetti trasformano ogni cosa in oggetti.
Per esempio, una determinata automobile può essere definita un oggetto. Ogni automobile ha alcune caratteristiche particolari che la distinguono dalle altre, come per esempio il colore, dov'è stata costruita o l'azienda che la produce. Ha anche una targa che la identifica come unica.
Nella programmazione ad oggetti ogni determinata automobile è un'istanza della classe automobile. La classe automobile contiene e identifica tutte le automobili costruite con determinate proprietà. L'istanza della classe presenta particolari caratteristiche diverse dalle altre. Ogni nuova automobile è una nuova istanza della classe automobile.
Tutte le azioni che si possono svolgere su una determinata automobile (accelerare, frenare, ecc.) sono chiamati metodi. Ogni metodo può essere applicato alla classe o a una singola istanza. Se il metodo è applicato all'istanza è considerato un metodo d'istanza.

2.9 Dalla realtà al codice

Le classi in Objective-C si sviluppano creando un file header e un file di implementazione, che hanno lo stesso nome ma estensioni diverse; per creare una classe Automobile, occorre creare un file *Automobile.h* (header) e un file *Automobile.m* (implementazione). Nel file header si inseriscono le parti pubbliche della classe: nomi e tipi delle variabili e signature dei metodi, che descrivono il tipo del valore di ritorno e i parametri. Nel file di implementazione si implementano i metodi con il codice, dichiarando e utilizzando tutte le variabili locali necessarie che sono relative soltanto all'implementazione e non devono essere visibili ad altre classi. Come in C, occorre anche indicare dove il compilatore può trovare altro codice utilizzato per l'applicazione, importando con

import il file header delle classi a cui si fa riferimento negli header o nell'implementazione.
In Objective-C le chiamate di metodi rappresentano in realtà "messaggi" inviati agli oggetti in memoria da un piccolo motore di runtime. Si tratta di una sottilissima differenza che consente di ottenere un comportamento sorprendentemente dinamico in un linguaggio di programmazione che inizialmente potrebbe sembrare piuttosto formale. Per esempio se si chiama un metodo relativo ad un oggetto *nil* non si verifica un crash né un errore (come il *NullPointerException* di Java): il runtime rileva che l'invio di un messaggio a un oggetto inesistente non avrebbe senso, e perciò non lo invia.
Le tipiche chiamate di metodi Objective-C sono racchiuse da parentesi quadre, iniziando con l'oggetto a cui seguono i nomi dei parametri e i valori forniti da essi. Per esempio se si ha un'istanza *NSString* denominata *myString*, si potrebbe ottenere il suo terzo carattere con una chiamata come la seguente:

```
myChar = [myString characterAtIndex:3];
```

La differenza più significativa rispetto ad altri linguaggi, a parte le parentesi quadre, è che i parametri hanno sempre un nome, e il costrutto parametro e segno di due punti (:) prende il nome di parola chiave o keyword. Occorre digitare più codice, che però risulta molto più leggibile rispetto ad altri linguaggi C-like dove si potrebbero utilizzare una decina di parametri racchiusi in una coppia di parentesi senza alcuna indicazione su ciò che rappresentano.

2.10 Le classi fondamentali di Cocoa Touch

I framework di *Cocoa Touch* contengono la maggior parte delle classi più utilizzate nello sviluppo di un'applicazione. Il nome deriva da *Cocoa*, i framework orientati agli oggetti sviluppati per la programmazione sul Mac OS X (e prima ancora su NeXTSTEP), insieme a classi GUI appositamente progettate per l'utilizzo su un dispositivo mobile e dotato di touchscreen (da qui "Touch").

Il framework *Foundation* di Cocoa contiene le classi dati essenziali. Tutte le classi Cocoa sono eredi di una classe root, *NSObject*, definita in *Foundation*.
Le classi più importanti in *Foundation* sono quelle di gestione dei dati, utilizzate in *Cocoa* al posto delle strutture equivalenti in C procedurale. Per esempio, la stringa del C tradizionale, l'array *char* terminato con *null*, non è quasi mai utilizzato in *Cocoa*; al suo posto si utilizza ***NSString***, che rappresenta non solo i dati carattere, ma anche la relativa codifica. Con l'ampio supporto per Unicode, ***NSString*** facilita la gestione del testo in tutti i numerosi set di caratteri disponibili in iPhone.
Cocoa fornisce anche un esteso insieme di classi *collezione*, che evitano in molti casi di ricorrere ad array C. Esistono tre classi utilizzate per la collezione di oggetti Cocoa: ***NSArray*** per collezioni di oggetti ordinate, ***NSSet*** per collezioni non ordinate e ***NSDictionary*** per associare oggetti chiave a oggetti valore. Queste tre collezioni sono immutabili: una volta inizializzate, non possono essere modificate. Se si desidera aggiungere, rimuovere o modificare il loro contenuto, occorre utilizzare le sottoclassi mutabili ***NSMutableArray***, ***NSMutableSet*** e ***NSMutableDictionary***.
Le collezioni possono contenere soltanto oggetti *NSObject*. Se si hanno delle primitive C, si possono utilizzare le classi wrapper ***NSData*** e ***NSMutableData***, che agiscono su un buffer di byte, e ***NSNumber***, un contenitore per qualsiasi tipo scalare (numerico) del C, come *int*, *float* o *bool*.
Cocoa ha anche alcune classi dati più specifiche, come ***NSURL*** per gli URL (compresi gli URL del tipo file:// che rappresentano elementi sul filesystem locale), e classi per la gestione di date e orari come ***NSDate*** e ***NSTimeZone***.
La parte "Touch" ci Cocoa Touch è largamente rappresentata dal framework *UIKit*, anch'esso importato per default in qualsiasi applicazione iPhone. Questo framework offre modello di disegno, gestione di eventi e ciclo di vita delle applicazioni. Si interagisce con questo framework attraverso le varie classi componenti dell'interfaccia utente: ***UIButton***, ***UITextView***, ***UITableView*** ecc.

2.11 Gestione della memoria

Un aspetto importantissimo delle applicazioni iPhone è la gestione della memoria. Lavorare con linguaggi moderni dotati di garbage collection come il Java, C#, Python, Ruby e così via, non richiedeva alcuno sforzo per quanto riguarda la gestione della memoria, dato che gli oggetti senza più riferimenti vengono rilasciati automaticamente. La necessità di gestire manualmente l'uso della memoria nelle applicazioni iPhone è probabilmente uno dei compiti più ardui per gli sviluppatori. Con una gestione non corretta, l'applicazione crea problemi nell'uso della memoria e alla fine rischia di essere terminata da parte del sistema. Cocoa utilizza un sistema di conteggio dei riferimenti abbastanza semplice: tutti gli oggetti sono allocati con un reference counting di 1. Questo valore può essere manipolato utilizzando due metodi istanza: *retain* incrementa il reference counting di uno, mentre *release* lo decrementa. Quando il valore arriva a zero, l'oggetto verrà rilasciato automaticamente attraverso la chiamata al metodo *dealloc*.

Esiste una regola fondamentale per la gestione della memoria in Cocoa: si è il proprietario di qualsiasi oggetto ogniqualvolta esso viene creato per mezzo di un metodo che utilizza le parole chiave *alloc*, *new* o *copy*. Per qualsiasi oggetto di cui lo sviluppatore è il proprietario, ad un certo punto egli dovrà effettuarne il rilascio. Il rovescio della medaglia è che non si ha la proprietà di alcun oggetto ricevuto per altri mezzi. Quindi non si devono rilasciare questi oggetti, perché è il codice che li ha creati ed è il sistema che avrà la responsabilità di rilasciarli. Tuttavia se si vuole essere proprietari di questi oggetti, l'unico modo che si ha è chiamare il metodo *retain* su di essi.

E' interessante il concetto di oggetti a rilascio automatico. Quando si invia il messaggio *autorelease* a un oggetto, lo si aggiunge ad un pool di oggetti a cui in futuro sarà inviato un metodo *release*, tipicamente quando lo stack di chiamata per la gestione di eventi termina. Il rilascio automatico è utile per metodi che restituiscono oggetti che non sono in grado di gestire; in questo modo il chiamante

ha una possibilità di mantenere l'oggetto restituito, e se non lo fa, alla fine questo verrà rilasciato.

2.12 Le proprietà

Le proprietà sono dei metodi che servono per esporre le variabili di istanza dell'oggetto che le dichiara al mondo esterno (incapsulamento). Possiamo avere proprietà di sola lettera, sola scrittura o lettura e scrittura.
Per definire una proprietà si utilizza un'istruzione nel file header in questa forma:

```
@property (attributes) proptype propname;
```

Nella parte tra parentesi tonda vanno inserite le informazioni necessarie al compilatore per scrivere correttamente i metodi setter e getter della proprietà.
Per creare, realmente, questi metodi, seguendo le specifiche dichiarate nelle property, occorre specificare nel file di implementazione:

```
@synthesize propname;
```

Gli attributi che una property può ricevere in ingresso sono:

1. **Nonatomic**: indica che il metodo che verrà creato avrà la caratteristica di atomicità. Ciò è indispensabile si si usano, nell'applicazioni, i thread per garantire la consistenza dei dati.
2. **Readwrite**: indica che è richiesto sia il metodo get che il metodo set dell'attributo. E' il comportamento di default.
3. **Readonly**: indica che è richiesto solo il metodo get dell'attributo.
4. **Assign**: indica che il metodo set userà semplicemente un assegnamento. Non verrà quindi invocato il metodo *retain* sull'oggetto. Si usa principalmente per attributi scalari come per esempio *int* e *float*:

   ```
   - (void)setNumero:(int)num {
       numero = num;
   }
   ```

5. **Retain**: incrementa di uno il contatore che gestisce i riferimenti all'oggetto.

   ```
   -(void)setNomeVariabile:(NSString *)str {
       if(str != nomeVariabile) {
   ```

```
        [nomeVariabile release];
        nomeVariabile = [str retain];
    }
}
```

6. **Copy**: indica che una copia dell'oggetto viene usata per l'assegnamento.

```
-(void)setNomeVariabile:(NSString *)str {
    if(str != nomeVariabile) {
        [nomeVariabile release];
        nomeVariabile = [str copy];
    }
}
```

E' possibile utilizzare più attributi nelle property concatenandoli semplicemente con una virgola. Alcuni attributi sono, però, mutuamente esclusivi e ciò vuol dire che non possono essere usati entrambi in una dichiarazione di una property. Infatti non è consentito utilizzare insieme l'attributo *readwrite* e *readonly*, come pure non è consentito usare, in una stessa dichiarazione, *assign*, *copy* e *retain*.
Fornendo lo strumento delle property, l'Objective-C semplifica il lavoro dello sviluppatore, riducendo notevolmente il codice da scrivere per quanto riguarda i metodi di accesso agli attributi.

2.13 Programmazione multithreading

La programmazione multithreaded è un aspetto fondamentale nei moderni linguaggi di programmazione poiché grazie ai thread è possibile eseguire funzioni diverse all’interno della stessa applicazione in un preciso istante. Tuttavia i principali svantaggi di questo approccio sono rappresentati dalla grande complessità di coordinare tra di loro i thread, dall’accesso alle risorse condivise e infine dal debugging. L’utilizzo di questo approccio si è diffuso ormai in tutti i tipi sistemi, dai sistemi embedded ai desktop computer.
Cocoa mette a disposizione una classe per realizzare i thread il cui nome è *NSThread*. Con *NSThread* si lancia un thread all'interno dell'applicazione

corrente. Per comunicare tra thread l'approccio più comune è lo scambio di messaggi tra oggetti.
Il problema sorge quando diversi thread tentano di accedere contemporaneamente alle stesse risorse (alla stessa locazione di memoria). Avere thread che non si preoccupano di proteggere gli oggetti condivisi durante una modifica è un metodo molto semplice per creare danni ed inserire errori difficili da trovare. *Cocoa* fornisce meccanismi di protezione, in modo che thread diversi non possano accedere contemporaneamente alla stessa risorsa, ma rimane compito del programmatore scrivere bene il codice.
Per proteggere sezioni di codice dall'essere eseguite da più di un thread alla volta, l'Objective-C fornisce la direttiva *@synchronized()*.
La direttiva *@synchronized()* chiude una sezione di codice in modo tale che vi possa accedere un solo thread alla volta. Gli altri threads sono bloccati finché il thread all'interno della sezione protetta, esce da essa.
La direttiva *@synchronized()* prende come argomento qualsiasi oggetto compreso *self*. Quest'oggetto è conosciuto come mutex.
Un esempio di codice che usa *self* come il mutex per sincronizzare l'accesso a delle risorse è mostrato sotto:

```
- (void)criticalMethod
{
    @synchronized(self) {
        // Critical code.
        ...
    }
}
```

Per quanto riguarda la creazione di un thread esiste un metodo particolarmente semplice da capire appartenente alla classe *NSThread*: *detachNewThreadSelector:toTarget:withObject:*. Un esempio di creazione di un thread è riportato sotto:

```
[NSThread detachNewThreadSelector:@selector(myMethod)
                        toTarget:self
                      withObject:nil];
```

Un esempio di metodo chiamato dal thread è il seguente:

```
- (void)myMethod {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init]
    *** code that should be run in the new thread goes here ***
    [pool release];
}
```

Se si vuole lanciare un thread da dentro un altro thread, bisogna usare il metodo *performSelectorOnMainThread:withObject:waitUntilDone:*. Ecco un esempio:

```
[self performSelectorOnMainThread:@selector(myMethod)
                        withObject:nil
                     waitUntilDone:false];
```

2.14 Le categorie

Una categoria consente di aggiungere metodi ad una classe esistente senza avere a disposizione il codice sorgente. Questa è una caratteristica molto potente che permette di estendere le funzionalità esistenti senza sottoclassi.

E' possibile aggiungere metodi ad una classe attraverso una categoria, dichiarandoli in un file interfaccia sotto un nome di categoria e definendoli in un file implementazione sotto lo stesso nome. Il nome di categoria indica che i metodi sono delle aggiunte ad una classe dichiarata da qualche altra parte, non una nuova classe. Non si può, comunque usare una categoria per aggiungere variabili di istanza ad una classe.

I metodi che la categoria aggiunge entrano a far parte del tipo della classe.

Ad esempio i metodi aggiunti alla classe *NSArray* in una categoria sono tra i metodi che il compilatore si aspetta di trovare nel repertorio dei metodi di un'istanza della classe *NSArray*.

I metodi definiti nella categoria vengono ereditati da tutte le sottoclassi, come accade per gli altri metodi.

La dichiarazione di un'interfaccia di categoria assomiglia molto alla dichiarazione dell'interfaccia di una classe, tranne per il nome di categoria che è racchiuso tra

parentesi dopo il nome di classe. La categoria deve importare il file d'interfaccia della classe che estende:

```
#import "ClassName.h"
@interface ClassName (CategoryName)
    // method declarations
@end
```

Una categoria non può dichiarare variabili di istanza aggiuntive per la classe ma può comprendere solo metodi. Tuttavia, tutte le variabili di istanza entro lo scope della classe sono anche nello scope della categoria incluse quelle dichiarate *@private*.

Non c'è limite al numero di categorie che è possibile aggiungere ad una classe, ma ogni nome di categoria deve essere diverso, ed ognuno dovrebbe dichiarare e definire un insieme diverso di metodi.

Ci sono tanti modi di usare le categorie:

1. Per estendere classi definite da altri programmatori, ad esempio, per aggiungere metodi alle classi definite nel framework *Cocoa*. I metodi aggiunti sono ereditati dalle sottoclassi e sono indistinguibili a runtime dai metodi della classe originale.
2. Come alternativa ad una sottoclasse. Piuttosto che definire una sottoclasse per estendere una classe esistente, attraverso una categoria si possono aggiungere metodi alla classe direttamente. Ad esempio, si possono aggiungere categorie a *NSArray* e altre classi *Cocoa*.
3. Per distribuire l'implementazione di una nuova classe in files sorgenti separati. Ad esempio, si possono raggruppare i metodi di una classe di grandi dimensioni in diverse categorie e mettere ogni categoria in un proprio file.

Capitolo 3
Standard ed algoritmi utilizzati

In questo capitolo vengono descritte le specifiche del linguaggio XPath, il metodo dell'Edit Distance e i tipi di parser presenti in Objective-C. La conoscenza di questi tre aspetti è stata fondamentale per lo sviluppo dell'applicazione.

3.1 Il linguaggio XPath

Grazie al linguaggio XPath è possibile recuperare i contenuti all'interno di un documento XML. La versione 1.0 di XPath è diventato uno standard del W3C (World Wide Web Consortium) il 16 Novembre 1999.
XPath si compone di espressioni che vengono di volta in volta valutate in un determinato contesto restituendo uno o più nodi del documento XML.
Un'espressione XPath è costituita da combinazioni di uno o più chunk di testo separati da caratteri speciali.
La maggior parte delle espressioni XPath individuano contenuti del documento XML preso in esame o parte di questi: tale processo avviene in uno o più passi, detti location step, che messi insieme danno vita ad un percorso completo di localizzazione ovvero una location path. Ad esempio l'espressione */cartella/fascicolo* risulta essere costituita da due passi di localizzazione: il primo passo recupera gli elementi cartella figli dell'elemento root del documento XML,

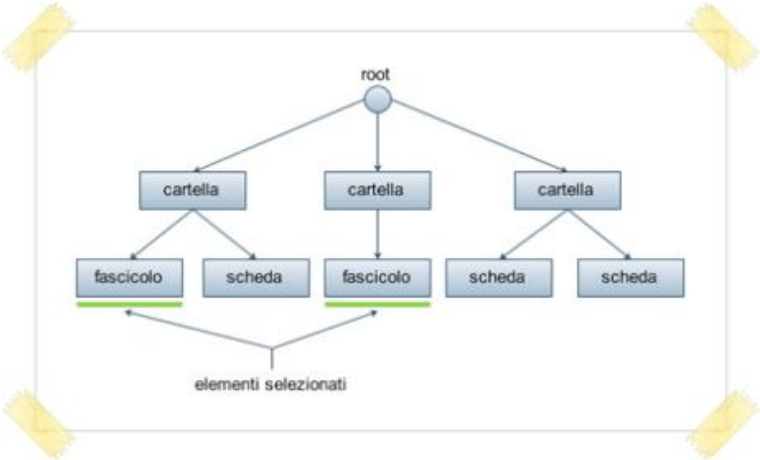


Figura 3.1 Albero dei nodi

il secondo recupera gli elementi fascicolo figli degli elementi cartella recuperati al passo precedente; messi insieme costituiscono una location path completa.
La figura 3.1 mostra il risultato della location path */cartella/fascicolo*. Vengono selezionati soltanto i nodi-elemento fascicolo figli dei nodi-elemento cartella mentre tutti gli altri nodi presenti all'interno del documento vengono trascurati.
Le espressioni XPath risultano essere costituite da token e delimitatori: i token in particolare sono insiemi di caratteri Unicode che assumono il significato di stringa se delimitati da virgolette ed il significato di nodi-elemento se privi di virgolette.
Le specifiche XPath prevedono l'utilizzo del carattere * come wildcard: il suo utilizzo permette di selezionare tutti i nodi del contesto indipendentemente dal loro nome.
Così l'espressione */cartella/** permette di selezionare indipendentemente tutti i nodi figlio degli elementi cartella presenti sotto l'elemento radice del documento xml.
I token di un'espressione XPath sono separati da opportuni delimitatori:

- / separa un passo di localizzazione da un altro.
- [...] permettono di definire un predicato che agisce sul contesto individuato dai location step che precedono il predicato stesso.
- =, !=, <, >, <=, >= vengono usati all'interno di un predicato per verificare il valore *true* o *false* del test applicato sul contesto (assumono rispettivamente il significato di uguale, diverso, minore, maggiore, minore o uguale, maggiore o uguale).
- :: separa l'asse di un'espressione XPath dal nome di uno specifico nodo.
- //, @, ., .. costituiscono delle forme abbreviate per indicare rispettivamente i concetti di "se stesso o discendente", "attributo", "se stesso", "genitore".
- | permette di definire un'espressione XPath composta dall'unione di più espressioni XPath
- (...) permettono di raggruppare sotto espressioni XPath o delimitare gli argomenti di una funzione XPath.
- +, -, *, div, mod agiscono come operatori numerici col significato di somma, sottrazione, moltiplicazione, divisione e modulo.

XPath è in grado di trattare quattro tipi differenti di dati: stringhe, numeri, booleani e nodi.

3.1.1 Tipi di nodi

XPath è in grado di recuperare ogni tipo di contenuto presente all'interno di un documento XML: non solo elementi e attributi ma anche commenti, Processing Instruction, nodi testo ed altro ancora.
Le specifiche XML prevedono che l'intero documento sia contenuto all'interno di un unico elemento detto radice (root): in XPath l'elemento radice è rappresentato dal carattere / e tutte le location path espresse a partire da tale elemento (tutte quelle che iniziano cioè col carattere /) sono considerate path assolute.
Ogni nodo elemento è generalmente composto da un tag di apertura, uno eventuale di chiusura e la porzione di documento xml compreso fra essi che a sua volta è costituito generalmente da altri nodi elemento.

3.1.2 Location Step e Location Path

La struttura tipica di una *location path* prevede dei token separati da delimitatori: */cartella/documento/sezione/paragrafo*.
Essa risulta quindi composta da *location step* disposte in maniera tale che ad ogni passo (ovvero procedendo da sinistra verso destra della location path) vengono esclusi dei nodi dalla selezione e vengono considerati i rimanenti.
La sintassi tipica di un location step prevede l'asse (opzionale), il nodetest (obbligatorio) ed il predicato (opzionale) organizzati secondo la seguente sintassi: *asse::nodetest[predicato]*.
Il nodetest indica il tipo di nodi che vogliamo selezionare.
Esistono due possibili approcci: il primo prevede l'identificazione dei nodi in base al nome degli elementi, il secondo invece prevede l'identificazione dei nodi in base al tipo degli elementi.

Per selezionare gli elementi con un particolare nome è sufficiente indicare come nodetest il nome dell'elemento. Così l'espressione */documento* indica di selezionare tutti gli elementi figli dell'elemento radice che hanno nome pari a "*documento*" (non si è indicato né l'asse né alcun predicato ma solo il nodetest).
Per selezionare tutti gli elementi a prescindere dal loro nome è possibile utilizzare il carattere wildcard *. L'espressione /* seleziona tutti i figli dell'elemento radice.
Per selezionare soltanto i nodi-testo si può utilizzare l'espressione *text()* come nell'esempio seguente: */documento/text()*.
In tal modo si selezionano i nodi-testo figli degli elementi che hanno nome "*documento*" (figli a loro volta dell'elemento radice).
Analogamente si può utilizzare *node()* per selezionare tutti i nodi indipendentemente dal tipo, *comment()* per selezionare i nodi-commento, *processing-instruction()* per selezionare le processing-instruction indipendentemente dal target e *processing-instruction(target)* per selezionare le processing-instruction aventi target indicato.
Attributi e namespace non possono essere selezionati facendo riferimento al solo nodetest ma occorre specificare l'asse.

3.1.3 L'asse

L'asse di un location step permette di indicare la direzione verso la quale si vogliono esaminare i nodi filtrati.
XPath definisce 13 tipi differenti di assi:

- ***child::*** individua il nodo immediatamente discendente dal nodo contesto.
- ***parent::*** individua il nodo genitore del nodo contesto.
- ***descendant::*** individua tutti i nodi discendenti del nodo contesto a qualsiasi profondità (i figli, i figli dei figli, ecc.).
- ***ancestor::*** individua tutti gli antenati del nodo contesto fino all'elemento radice (il padre, il padre del padre ...).
- ***descendant-or-self::*** individua il nodo contesto stesso e tutti i suoi discendenti.

- ***ancestor-or-self::*** individua il nodo contesto stesso e tutti i suoi antenati.
- ***following::*** individua tutti i nodi che all'interno del documento XML seguono il nodo contesto esclusi i discendenti del nodo contesto.
- ***preceding::*** individua tutti i nodi che all'interno del documento XML precedono il nodo contesto esclusi gli antenati del nodo contesto.
- ***following-sibling::*** individua tutti i nodi che seguono il nodo contesto e che condividono con questo lo stesso nodo parent.
- ***preceding-sibling::*** individua tutti i nodi che precedono il nodo contesto e che condividono con questo lo stesso nodo parent.
- ***attribute::*** individua gli attributi del nodo contesto.
- ***namespace::*** individua i nodi namespace.
- ***self::*** individua il nodo contesto stesso.

Per gli assi *child::*, *parent::*, *attribute::*, *descendant-or-self::* e *self::* esistono le espressioni abbreviate "/", "..", "@", "//" e ".".

3.1.4 I predicati

L'utilità dei predicati all'interno delle espressioni XPath sta nella possibilità di poter filtrare ulteriormente i nodi dal contesto correntemente selezionato in base all'impostazione di opportune condizioni.
Generalmente un predicato è racchiuso fra parentesi quadre ed espresso in termini di espressioni booleane del tipo: ***valore1 operatore valore2*** dove *valore1* e *valore2* sono delle espressioni XPath e operatore è un operatore booleano.
Ogni *location path* che appare all'interno del predicato fa riferimento al contesto stabilito dai location step che precedono il predicato stesso.
Per esempio considerato il seguente documento XML:

```
<persona nome="Antonio">
   <fratello nome="Matteo"/>
   <fratello nome="Marco"/>
```

<fratello nome="Luca"/>
</persona>

L'espressione: */persona/fratello[@nome='Matteo']* fa riferimento al contesto determinato dall'expression path */persona/fratello* quindi selezionerà il nodo elemento fratello che ha come attributo nome "*Matteo*".

E' possibile definire all'interno di un predicato altri predicati innestati:
//persona[figlio[@nome='Giovanni']]

La precedente espressione seleziona tutte le persone che hanno dei figli che hanno l'attributo nome pari a Giovanni.

Oppure è possibile comporre più predicati mediante operatori logici *and* e *or*:
//persona[figlio[@nome = 'Giovanni'] or figlio[@nome='Matteo']].

In alcuni casi può sorgere la necessità di impostare dei predicati per verificare o meno l'esistenza di un determinato nodo figlio del contesto attuale.

In questi casi non si ricorre più ad una sintassi del tipo *valore1 operatore valore2* ma viene specificato il solo *valore1*, così l'espressione:
//cartella[descendant::documento] seleziona solo gli elementi cartella che hanno almeno un documento come discendente.

Un caso particolare di predicati è rappresentato da quelli a valori numerici utilizzati per effettuare filtraggi sul contesto in base a quella che è la context-position occupata dai nodi selezionati ad un determinato momento.

Se si vuole selezionare il terzo elemento cartella di tutto il documento ad esempio è possibile utilizzare l'espressione *//cartella[position()=3]* e, in alternativa, la forma contratta *//cartella[3]*.

3.2 I parser HTML/XML disponibili

In molti casi un'applicazione ha bisogno di prendere dati dalla rete, spesso da una pagina web. E' necessario quindi un parser che consenta di prelevare i dati, ad esempio da una pagina web, e trasformarli in oggetti (*NSArray*, *NSDictionary*). Lo scopo del parsing quindi è proprio questo: trasformare codice di markup (XML e HTML, per esempio) in oggetti.

Il requisito fondamentale che deve avere un parser HTML è la error-tolerance, cioè la capacità di effettuare il parsing di contenuti che possano presentare errori di sintassi.
In iPhone SDK è inclusa una classe, *NSXMLParser*, che permette di effettuare il parsing di file XML e HTML.
Sono presenti anche altre librerie di terze parti come per esempio *TBXML*, *TouchXML*, *KissXML*, *Hpple*, ecc.
In generale i parser HTML/XML si dividono in due categorie: **parser SAX** e **parser DOM**.
Il **parser SAX** implementa un approccio *event-driven* ossia basato sulla scansione sequenziale del documento e sulla gestione di alcuni eventi significativi (inizio del documento, rilevazione di un tag, chiusura di un tag, ecc.) da parte di un opportuno delegato. Il punto di forza di questo approccio risiede nel fatto che non necessita di caricare tutto l'albero XML in memoria, il che è un vantaggio nel caso si manipolino documenti di grandi dimensioni.
Il **parser DOM** incapsula la struttura del documento in memoria così da permettere l'effettuazione di interrogazioni con sintassi XPath per recuperare i nodi che rispettano i criteri specificati.
Questo approccio si rivela più intuitivo e soprattutto più efficiente dal punto di vista computazionale per documenti di dimensione modesta.
I parser XML/HTML più popolari sono i seguenti:

1. ***NSXMLParser***: è un parser SAX incluso nell'iPhone SDK. E' scritto in Objective-C, è molto semplice da usare. Esso non permette il salvataggio e l'editing di documenti XML e non permette nemmeno l'XPath. Il grande limite di questo parser è rappresentato dal fatto che accetta in input solo XML o XHTML sintatticamente perfetto. Al primo errore si blocca.
2. ***Libxml2***: è una libreria open source inclusa nell'iPhone SDK. E' scritta in C per cui è più difficile da usare rispetto all'*NSXMLParser*. La libreria supporta sia l'approccio DOM sia l'approccio SAX.
3. ***TBXML***: è un parser XML DOM molto leggero progettato per essere molto veloce e per consumare poca memoria. Esso non permette il

salvataggio e l'editing di documenti XML e non permette nemmeno l'XPath. Il maggior limite di questo parser è rappresentato dal fatto che accetta in input solo XML o XHTML sintatticamente perfetto.

4. ***TouchXML***: è un parser XML DOM molto simile all'*NSXMLParser*. E' di sola lettura e supporta l'XPath. Come per l'*NSXMLParser* e per il *TBXML*, accetta in input solo XML o XHTML sintatticamente perfetto.
5. ***KissXML***: è un parser XML DOM molto simile all'*NSXMLParser*, basato però sul *TouchXML*. La principale differenza tra i due è che *KissXML* supporta anche l'editing e la scrittura di pagine XML. Accetta in input solo XML o XHTML sintatticamente perfetto.
6. **TinyXML**: è un parser XML DOM molto leggero scritto in C. Esso supporta sia la lettura e sia la scrittura di documenti XML, ma non supporta l'XPath. Accetta in input solo XML o XHTML sintatticamente perfetto.
7. **GDataXML**: è un parser XML DOM molto leggero, simile all'*NSXMLParser*, sviluppato da Google. Esso supporta sia la scrittura e sia la lettura di documenti XML e supporta anche l'XPath. Accetta in input solo XML o XHTML sintatticamente perfetto.
8. **Hpple**: è un parser HTML/XML DOM molto leggero. Esso supporta solo la lettura di documenti HTML/XML e l'XPath. Diversamente da tutti gli altri parser, accetta in input qualsiasi tipo di pagina HTML/XML.
9. **HTMLParser**: è un parser HTML/XML SAX molto leggero. Esso supporta la sola lettura del documento HTML/XML consentendo la navigazione dei vari tag. Accetta in input qualsiasi tipo di pagina HTML/XML.
10. **SSJujuParser**: è un parser HTML/XML DOM basato sul parser *Hpple*. A differenza del parser *Hpple*, esso presenta molti meno bug.

3.3 L'Edit Distance

L'Edit Distance, detto anche Levenshtein Distance, è stato inventato dallo scienziato russo Vladimir Levenshtein nel 1965. L'Edit Distance è una misura di similarità di due stringhe cioè serve a determinare quanto due stringhe siano simili.

L'Edit Distance tra due stringhe s_1 e s_2 è il numero minimo di modifiche elementari (sostituzione, cancellazione o inserimento di un carattere) che consentono di trasformare la s_1 nella s_2. Ad esempio, la distanza tra "*gtgcca*" e "*ggcga*" è 2 poiché è necessario inserire una "*t*" nella seconda stringa e sostituire il penultimo carattere "*g*" con una "*c*" per ottenere la prima stringa).

L'Edit Distance rappresenta il costo dell'allineamento ottimo di due stringhe, quando tale costo è misurato attribuendo a due simboli (caratteri) *a* e *b*, uno dei quali può essere un *gap* (cioè un "buco"), il valore seguente:

$$c(a, b) = \begin{cases} 0 & \text{se } a = b \\ 1 & \text{se } a \neq b \text{ o } a = - \text{ o } b = - \end{cases}$$

dove il simbolo − denota un *gap*. Il costo complessivo dell'allineamento è dato dalla somma dei costi di ciascuna posizione. Ad esempio, un allineamento ottimo, di costo 2 (sono presenti due differenze), delle due stringhe precedenti è *gtgcca* e *g-gcga*.

Un allineamento non ottimo potrebbe essere *gtgcc-a* e *gg-c-ga* di costo 4 (sono presenti quattro differenze).

L'algoritmo per calcolare l'Edit Distance tra due stringhe si basa sull'osservazione seguente.

Siano $s_1 = a_1 \ldots a_m$ e $s_2 = b_1 \ldots b_n$ due stringhe. Sia $e(i, j)$ l'Edit Distance tra i due prefissi $p_1 = a_1 \ldots a_i$ e $p_2 = b_1 \ldots b_j$ di s_1 e s_2 rispettivamente. Se $e(i-1, j-1)$, $e(i-1, j)$ ed $e(i, j-1)$ sono noti, allora si può calcolare $e(i, j)$. Ciascuno dei tre valori corrisponde infatti a un allineamento ottimo:

- $e(i-1, j-1)$ è il costo dell'allineamento ottimo di $a_1 \ldots a_{i-1}$ e $b_1 \ldots b_{j-1}$
- $e(i-1, j)$ è il costo dell'allineamento ottimo di $a_1 \ldots a_{i-1}$ e $b_1 \ldots b_j$

- $e(i, j-1)$ è il costo dell'allineamento ottimo di $a_1 \ldots a_i$ e $b_1 \ldots b_{j-1}$

Ognuno dei tre allineamenti può essere esteso a un allineamento tra $a_1 \ldots a_i$ e $b_1 \ldots b_j$:

- allineando a_i con b_j
- allineando a_i con un *gap*
- allineando un *gap* con b_j

Per mantenere l'ottimalità, bisognerà scegliere l'operazione che consente di ottenere l'allineamento di costo minimo. Si può notare che il generico valore $e(i, j)$ dipende solamente da $e(i-1, j-1)$, $e(i, j-1)$ ed $e(i-1, j)$, e può essere calcolato usando la seguente equazione:

$$e(i, j) = min \begin{cases} e(i-1, j-1) + c(a_i, b_j); \\ e(i, j-1) + 1; \\ e(i-1, j) + 1 \end{cases}$$

Questi valori si possono disporre in una matrice $(m+1) \times (n+1)$ (dove ε denota la stringa vuota):

	ε	b_1	b_2	$\cdots$	b_n
ε	$e(0,0)$	$e(0,1)$	$e(0,2)$	$\cdots$	$e(0,n)$
a_1	$e(1,0)$	$e(1,1)$	$e(1,2)$	$\cdots$	$e(1,n)$
a_2	$e(2,0)$	$e(2,1)$	$e(2,2)$	$\cdots$	$e(2,n)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_m	$e(m,0)$	$e(m,1)$	$e(m,2)$	$\cdots$	$e(m,n)$

È evidente che l'allineamento di una stringa vuota con una stringa $x_1 \ldots x_i$ ha costo i (bisogna inserire i caratteri nella stringa vuota per ottenere $x_1 \ldots x_i$):

$e(0, i) = e(i, 0) = i$.

In particolare, l'allineamento tra due stringhe vuote ha costo zero:

$e(0, 0) = 0$.

La matrice pertanto può essere riscritta così:

	ε	b_1	b_2	$\cdots$	b_n
ε	0	1	2	$\cdots$	n
a_1	1	$e(1,1)$	$e(1,2)$	$\cdots$	$e(1,n)$
a_2	2	$e(2,1)$	$e(2,2)$	$\cdots$	$e(2,n)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_m	m	$e(m,1)$	$e(m,2)$	$\cdots$	$e(m,n)$

Sulla base delle osservazioni precedenti, la matrice può essere costruita *riga per riga*, ossia da sinistra a destra e dall'alto in basso.

Il valore $e(m, n)$ rappresenta l'Edit Distance tra s_1 e s_2.

Parte II

Progetto dell’applicazione

Capitolo 4
Progetto: tecniche e scelte

In questo capitolo vengono descritte le varie tecniche e le varie scelte che sono state utilizzate nello sviluppo dell'applicazione.

4.1 Funzionalità applicazione

L'applicazione creata deve poter calcolare l'H-Index di un generico scienziato facendo riferimento a Google Scholar ed eventualmente all'homepage dello scienziato di riferimento in cui sono presenti tutte e solo le sue pubblicazioni.
L'applicazione è composta principalmente da tre viste: la prima è la vista di ricerca, la seconda è la vista in cui sono presenti tutte le pubblicazioni trovate dall'homepage, la terza è la vista in cui sono presenti l'H-Index dello scienziato e le pubblicazioni trovate da Google Scholar.
L'utente potrà spostarsi da una vista all'altra grazie ad una *UITabBar*.

All'apertura dell'applicazione, verrà visualizzata la prima vista nella quale sono presenti due *UITextField*, uno *UISwitch*, due *UISlider* e un *UIButton*.
La prima *UITextField* partendo dall'alto è il campo di testo in cui l'utente deve obbligatoriamente inserire il nome e il cognome (non è importante l'ordine di inserimento) dello scienziato per il quale si vuole trovare l'H-Index, appoggiandosi solo ed esclusivamente a Google Scholar. A questo punto l'utente potrà iniziare la ricerca premendo semplicemente il *UIButton* "SEARCH".
Lo *UISwitch* presente esattamente sotto la prima *UITextField* serve per attivare/disattivare la ricerca avanzata delle pubblicazioni, oltre che da Google Scholar, anche dall'homepage in cui sono presenti tutte le pubblicazioni dello scienziato di riferimento.
Se lo *UISwitch* è attivo verranno abilitati i componenti immediatamente sotto di questo: una *UITextField* che dovrà essere riempita dall'utente con l'URL della

pagina web in cui si trovano le pubblicazioni; due *UISlider* che rappresentano il grado di similarità dei titoli (la prima *UISlider* partendo dall'alto) e degli autori (la seconda *UISlider)* con i titoli e gli autori estrapolati da Google Scholar. Facendo un esempio, se l'utente imposta il grado di similarità dei titoli al 90% e il grado di similarità degli autori all'80%, l'applicazione, quando andrà alla ricerca dei titoli, durante l'algoritmo dell'estrapolazione intelligente spiegato più avanti, scarterà tutte quelle stringhe che sono simili all'89% o meno alle stringhe rappresentanti i titoli estrapolate da Google Scholar. Stessa cosa avviene per gli autori, solo che in questo caso verranno scartate tutte quelle stringhe che sono simili al 79% o meno alle stringhe rappresentanti gli autori estrapolate da Google Scholar.
Nella parte inferiore della vista di ricerca è presente un *UIButton* con la scritta "SEARCH" che l'utente potrà premere quando avrà finito di modificare i vari componenti presenti. Alla pressione del *UIButton* "SEARCH" verrà visualizzata subito una *UIActivityIndicatorView*, un componente di caricamento, una *UILabel* che farà vedere all'utente cosa sta facendo in un certo istante l'applicazione e un *UIButton* "Stop" che potrà essere premuto dall'utente per terminare la ricerca dei titoli o degli autori dall'homepage.
Se la ricerca non va a buon fine, verrà visualizzata una *UIActionSheet*, un popup che spiegherà il perché la ricerca è fallita.

Se invece la ricerca va a buon fine, nel caso in cui l'utente abbia tenuto disattivato o disattivato lo *UISwitch* della prima vista, verrà visualizzata la terza vista contenente il nome dello scienziato, il relativo H-Index, il numero delle pubblicazioni trovate e la tabella contenente in dettaglio ogni pubblicazione trovata (titolo, autori, numero di citazioni e anno) da Google Scholar. L'utente potrà attivare/disattivare le pubblicazioni che riterrà opportuno semplicemente schiacciando con il dito sopra la pubblicazione che vuole attivare/disattivare. La disattivazione di una pubblicazione porta essa ad escluderla dal calcolo dell'H-Index. Al contrario, l'attivazione di una pubblicazione porta essa ad includerla dal calcolo dell'H-Index. I colori delle celle contenenti le pubblicazioni possono essere verde, grigio o rosso: il verde indica che la pubblicazione in

questione concorre all'H-Index; il grigio indica che la pubblicazione in questione può concorrere all'H-Index (sono presenti *n* pubblicazioni colorate di grigio con un uguale numero di citazioni ma solo *m* pubblicazioni, con $m < n$, concorrono all'H-Index); il rosso indica che la pubblicazione in questione non concorre all'H-Index.

Nel caso in cui l'utente abbia attivato lo *UISwitch* della prima vista, dopo il caricamento delle pubblicazioni da Google Scholar e dall'homepage dello scienziato in questione verrà visualizzata una nuova vista, la seconda, contenente il nome dello scienziato, il numero delle pubblicazioni trovate dall'homepage e la tabella contenente in dettaglio ogni pubblicazione trovata (titolo ed eventuali autori). I colori delle celle contenenti le pubblicazioni possono essere verde o rosso: il verde indica che la pubblicazione in questione è presente in Google Scholar; il rosso indica che la pubblicazione in questione non è presente in Google Scholar. L'utente in questa nuova vista potrà solo scorrere la tabella per vedere le pubblicazioni trovate.

Il vantaggio che si ha nella ricerca delle pubblicazioni dall'homepage di un generico scienziato è la seguente: l'H-Index verrà calcolato solo in base alle pubblicazioni che sono state effettivamente pubblicate dallo scienziato in questione. Vengono quindi eliminate tutte le pubblicazioni da Google Scholar che non sono presenti nell'homepage.
Purtroppo molte volte Google Scholar restituisce delle pubblicazioni che non appartengono all'autore considerato e quindi l'H-Index calcolato risulta erroneo. Può capitare anche il caso in cui Google Scholar non restituisce tutte le pubblicazioni che appartengono allo scienziato in questione. Anche in questo caso l'H-Index risulta erroneo ma il problema non è risolvibile.

4.2 Interazione tra classi

Il seguente class diagram mostra come sono interconnesse tra di loro le classi implementate dell'applicazione.

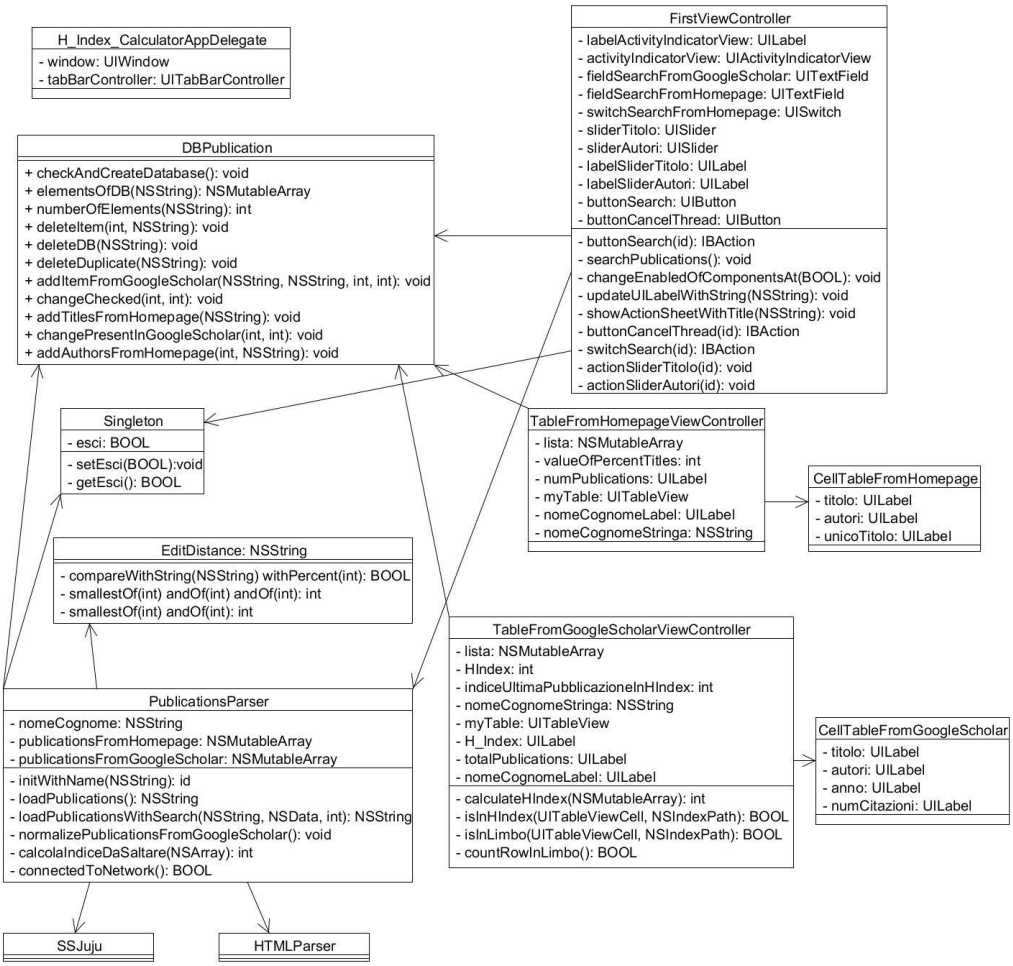


Figura 4.1 Interazione classi

La classe *H_Index_CalculateAppDelegate* è la classe delegata dell'applicazione e perciò eseguirà le funzioni in vece della classe *UIApplication* come spiegato nel precedente capitolo.

Le classi view controller sono quelle che gestiscono l'interfaccia grafica. In particolare la classe *FirstViewController* gestisce la vista di ricerca, la classe *TableFromHomepageViewController* gestisce la vista in cui è inclusa la tabella con le pubblicazioni estrapolate dall'homepage di un generico scienziato, la classe

TableFromGoogleScholarViewController gestisce la vista in cui è inclusa la tabella con le pubblicazioni estrapolate da Google Scholar.
Le classi *CellTableFromHomepage* e *CellTableFromGoogleScholar* rappresentano due generiche celle personalizzate rispettivamente delle tabelle rappresentanti le pubblicazioni estrapolate dall'homepage e le pubblicazioni estrapolate da Google Scholar.
La classe *PublicationsParser* è il cuore dell'applicazione poiché è quella deputata alla ricerca delle pubblicazioni da Google Scholar (con l'utilizzo del parser *SSJuju*) e dall'homepage dello scienziato di riferimento (con l'utilizzo dei parser *SSJuju* e *HTMLParser*); inoltre è deputata all'inserimento nel database delle pubblicazioni trovate.
La classe *DBPublication* è quella che gestisce ogni aspetto riguardante il database.
La classe *Singleton* è quella deputata alla modifica di una variabile condivisa tra la classe *FirstViewController* e *PublicationParser* per la gestione dell'azione riguardante il bottone "*Stop*" che ferma la ricerca delle pubblicazioni dall'homepage.
La classe *EditDistance* è in realtà una categoria che aggiunge principalmente un nuovo metodo alla classe *NSString*. Il metodo in questione applica a due stringhe l'algoritmo dell'Edit Distance spiegato nel capitolo 3.

4.3 Scelta del parser da utilizzare

La scelta di un buon parser è uno dei punti cruciali da affrontare durante lo sviluppo di un'applicazione.
Visto che l'applicazione in questione deve poter parsare documenti HTML da Google Scholar, che presentano tanti errori di formattazione, e deve poter navigare i tag dall'homepage di uno scienziato, è erroneo pensare di poter usare un solo parser.
Per l'applicazione corrente si sono usati quindi due diversi parser: HTMLParser per poter navigare i tag dall'homepage e *SSJuju* per poter usare query in formato XPath.

4.4 Gestione del database

L'SDK fornito da Apple integra in modo nativo il supporto a *SQLite*, una libreria software scritta in C che implementa un DBMS SQL, permettendo di creare un database all'interno di un unico file. SQLite è molto veloce e leggero, adatto per semplici applicazioni per dispositivi mobili, ma non è completo come un DBMS poiché manca di alcune caratteristiche avanzate.
All'interno dell'applicazione in questione è stato creato un database con *SQLite* contenente due tabelle: la prima, *publicationsFromGoogleScholar*, contenente tutte le pubblicazioni che sono state estrapolate da Google Scholar; la seconda, *publicationsFromHomepage*, contenente tutte le pubblicazioni che sono state estrapolate dall'URL dello scienziato in cui sono presenti tutte le sue pubblicazioni.
Il database *SQLite* è stato creato direttamente dal terminale del Mac OS X. Tramite il seguente comando: "*sqlite3 publications.sqlite*" si è avviato il programma *SQLite*, facendo partire la modalità interattiva, e si è creato un database con nome "*publications.sqlite*".
Le due tabelle sono state create con i seguenti comandi:

1. "*CREATE TABLE publicationsFromGoogleScholar (pk INTEGER PRIMARY KEY, numCitazioni INTEGER, titolo VARCHAR, autori VARCHAR, anno INTEGER, isChecked INTEGER);*"
2. "*CREATE TABLE publicationsFromHomepage (pk INTEGER PRIMARY KEY, titolo VARCHAR, autori VARCHAR, isPresentInGoogleScholar INTEGER);*"

Con il primo comando si dice al database di creare una tabella di nome "*publicationsFromGoogleScholar*", con l'elemento *pk*, un intero che si incrementa automaticamente per ogni *insert*, con i quattro elementi *numCitazioni*, *titolo*, *autori* e *anno* che rappresentano rispettivamente un intero contenente il numero delle citazioni della pubblicazione, una stringa contenente il titolo della pubblicazione, una stringa contenente gli autori della pubblicazione e un intero contenente l'anno in cui è stata finita la pubblicazione. L'ultimo elemento,

isChecked, è un intero legato prettamente all'interfaccia grafica e più specificatamente alla tabella in cui sono presenti tutte le pubblicazioni da Google Scholar. Questo elemento rappresenta lo stato in cui si trova la riga contenente una determinata pubblicazione e può assumere solo due valori: 0 significa che quella pubblicazione non concorre alla determinazione dell'H-Index (è stata deselezionata dall'utente); 1 significa che quella pubblicazione concorre alla determinazione dell'H-Index (è stata selezionata dall'utente). Di default, dopo la ricerca delle pubblicazioni di un determinato scienziato, questo elemento vale 1 per tutte le pubblicazioni. E' stato necessario inserire quest'ultimo elemento poiché ogni singola cella non prevede il salvataggio dello stato del checkmaker (il segno di spunta, abilitato o meno).

Con il secondo comando si dice al database di creare una tabella di nome "*publicationsFromHomepage*", con l'elemento *pk*, un intero che si incrementa automaticamente per ogni *insert*, con i due elementi *titolo*, *autori* che rappresentano rispettivamente una stringa contenente il titolo della pubblicazione e una stringa contenente gli autori della pubblicazione. L'ultimo elemento, *isPresentInGoogleScholar*, è un intero che rappresenta l'esistenza o meno di quella determinata pubblicazione in Google Scholar e può assumere solo due valori: 0 significa che quella pubblicazione non è presente in Google Scholar (la cella verrà colorata di rosso); 1 significa che quella pubblicazione è presente in Google Scholar (la cella verrà colorata di verde). E' stato necessario inserire quest'ultimo elemento per questioni di prestazioni: meglio calcolare una sola volta, alla visualizzazione della tabella, la presenza o meno di tutte le pubblicazioni in Google Scholar piuttosto che effettuare questo calcolo di continuo anche più di una volta per ogni pubblicazione durante lo scorrimento della tabella da parte dell'utente.

4.5 Problema: confronto dei tag. Soluzione: algoritmo Edit Distance

Durante lo sviluppo dell'applicazione, uno dei principali problemi che è stato affrontato è quello relativo al confronto tra tutti i tag relativi all'homepage di un certo scienziato con i titoli e gli autori estrapolati da Google Scholar per consentire di capire, grazie ad un algoritmo, dove fossero localizzati tutti i titoli e i relativi autori delle pubblicazioni all'interno dell'homepage.
Molte volte infatti una pubblicazione estrapolata da Google Scholar viene presentata con qualche piccola differenza di caratteri (uno spazio in più o in meno, un "!" in più o in meno, ecc.) rispetto alla stessa pubblicazione all'interno dell'homepage.
Per fare questi confronti quindi non è stato usato il metodo standard *isEqualToString*: appartenente alla classe *NSString*. Grazie a questo metodo infatti è possibile soltanto confrontare due stringhe e, se esse sono uguali, il metodo ritorna True, e se, al contrario, le due stringhe non sono uguali, il metodo ritorna False.
La soluzione a questo problema è stata la conoscenza dell'algoritmo Edit Distance, spiegato nel precedente capitolo.
Nell'applicazione il metodo che calcola l'Edit Distance tra due stringhe è stato aggiunto all'interno della classe *NSString* tramite una categoria.
In particolare il metodo creato riceve in input due parametri: una stringa e una percentuale impostata dall'utente rappresentante la percentuale di similarità desiderata.
Il metodo ritorna un valore booleano: *TRUE* se la percentuale di similarità tra le due stringhe è più grande di quella data in input; *FALSE* nel caso contrario.
Ad esempio se si chiama il metodo sulla stringa "*aaaab*" e si dà in input la stringa "*aaaaa*" e la percentuale di similarità al 75%, il metodo ritorna *TRUE* poiché le due stringhe sono simili all'80%.

4.6 Estrapolazione intelligente pubblicazioni da homepage

Durante lo sviluppo dell'applicazione uno dei maggiori problemi che è stato affrontato è stato l'estrapolazione delle pubblicazioni (titolo e autori) da una qualunque homepage di un qualsiasi scienziato.
Per l'occasione è stato ideato e alla fine implementato un nuovo algoritmo di ricerca di queste pubblicazioni.
L'algoritmo è composto principalmente da tre fasi.
La **fase uno** prevede, tramite il parser *HTMLParser*, l'estrapolazione di tutti i tag dall'homepage. I tag trovati vengono confrontati, utilizzando l'algoritmo dell'Edit Distance spiegato precedentemente, con tutti i titoli delle pubblicazioni trovati con Google Scholar. Se la ricerca non va a buon fine o se viene trovato un solo titolo, si avvisa l'utente che la ricerca non ha avuto esito positivo e quindi si esce. Se, al contrario, vengono trovati due titoli aventi lo stesso nome di tag, si passa alla fase successiva.
Si è scelto di trovare due titoli e non uno perché nella fase due dell'algoritmo si ha la necessità di risalire l'albero dei tag fino a quando non si riesce a trovare un tag comune ai due titoli in modo tale da costruire correttamente la query XPath per la ricerca.
La **fase due** prevede, sempre tramite il parser HTMLParser, l'individuazione precisa dei due titoli trovati all'interno della pagina HTML. Questa fase si può suddividere in più sotto fasi e prevede la memorizzazione di una stringa *str*, inizialmente vuota, contenente il path per il raggiungimento di tutti i titoli, la memorizzazione del tag in *node1* contenente un titolo e la memorizzazione del tag in *node2* contenente l'altro titolo:

1. Vengono estrapolati i nomi dei due tag (*node1* e *node2*).
2. Se i nomi dei tag sono diversi si esce.
3. Vengono estrapolati tutti gli attributi comuni dei due tag.
4. Il nome del tag (comune ai due titoli) e gli attributi trovati nelle sotto fasi uno e tre vengono aggiunti all'interno della stringa *str* seguendo la formattazione dell'XPath.

5. Se i due tag in *node1* e in *node2* sono effettivamente lo stesso tag si esce.
6. Vengono estrapolati i nodi padri dei due tag e memorizzati in *node1* e *node2*.
7. Ritorna nella sotto fase uno.

La fase due prevede quindi un ciclo iterativo che ha come unico obiettivo quello di trovare una query XPath per la ricerca di tutti i titoli presenti all'interno dell'homepage.
La **fase tre** prevede, tramite il parser *SSJuju*, la ricerca di tutti i titoli nell'homepage grazie all'utilizzo della stringa *str* costruita nella fase due.

La ricerca degli autori nell'homepage segue le stesse fasi della ricerca dei titoli tranne per il fatto che alla fine viene confrontato il numero dei risultati trovati con il numero dei titoli trovati precedentemente: se sono uguali si procede con la memorizzazione nel database; se non sono uguali si avvisa l'utente che la ricerca degli autori non è andata a buon fine.

Questo algoritmo funziona molto bene per tutte quelle homepage che presentano i titoli e gli autori in modo omogeneo (tutti i titoli hanno la stessa formattazione e stessa cosa vale per gli autori). Purtroppo nella realtà ci sono frequenti casi in cui le homepage non sono ben formattate per cui capita spesso di trovare soltanto i titoli o una parte dei titoli delle pubblicazione senza riuscire a trovare gli autori.

4.7 Esempi di ricerca

In questo paragrafo verranno elencati alcuni esempi di ricerca dell'H-Index tenendo conto anche delle pubblicazioni inserite nelle varie homepage.

E' stata fatta una ricerca dell'H-Index di Andrew Zisserman, un importantissimo informatico e professore dell'università di Oxford, celebre per le sue ricerche nel campo della visione artificiale. E' stata l'unica persona nel mondo che ha

conseguito per tre volte di seguito il Marr Prize, un prestigioso premio nel campo della visione artificiale.
Se si inserisce nel campo dedicato al nome e al cognome “Andrew Zisserman” e nel campo dedicato all’URL dell’homepage “www.robots.ox.ac.uk/~vgg/publications/” e settando le percentuali di significatività dei titoli e degli autori rispettivamente dell’80% e del 68%, dopo circa 100 secondi di caricamento verranno mostrate tutte le pubblicazioni con i relativi autori estrapolati dall’homepage. Se ci si sposta, con l’uso della *TabBar*, nella tabella con le pubblicazioni trovate da Google Scholar, salta subito all’occhio lo spaventoso H-Index che ha questo importantissimo informatico inglese: 79.

Un altro esempio è stato effettuato con la ricerca dell’H-Index del famosissimo informatico americano Richard M. Karp conosciuto per aver fatto importanti studi sulla teoria degli algoritmi. Richard M. Karp è anche professore presso l’università di Berkeley, California, ha vinto il premio Turing nel 1985 e il Premio Kyoto per la tecnologia nel 2008.
Se si inserisce nel campo dedicato al nome e al cognome “Richard Karp” e nel campo dedicato all’URL dell’homepage “http://www.cs.berkeley.edu/~karp/publications.html” e settando le percentuali di significatività dei titoli e degli autori rispettivamente dell’80% e dello 0%, dopo circa 60 secondi di caricamento verranno mostrati tutti e solo i titoli delle pubblicazioni estrapolati dall’homepage. Se ci si sposta, con l’uso della TabBar, nella tabella con le pubblicazioni trovate da Google Scholar, si nota subito l’elevato H-Index di questo eccellente informatico: 78.

Parte III

Implementazione

Capitolo 5
Particolarità d'implementazione

In questo capitolo vengono analizzati in dettaglio alcuni aspetti riguardo l'implementazione vera e propria dell'applicazione.

5.1 Classe PublicationsParser

La classe PublicationsParser è il cuore dell'applicazione.
Attraverso il metodo privato *loadPublications* va alla ricerca di tutte le pubblicazioni da Google Scholar tramite query in formato XPath e, se la ricerca va a buon fine, le inserisce nel database. Di seguito vengono riportate le tre query XPath che vengono eseguite per la ricerca dei titoli, autori/anno e numero citazioni.

```
// VADO ALLA RICERCA DEI TITOLI
[temp setString:[NSString
stringWithFormat:@"//div[@class='gs_r'][./font/span[@class='gs_a'][./b='%@'][../sp
an[@class='gs_fl']]]/div[@class='gs_rt']/h3/a |
//div[@class='gs_r'][./font/span[@class='gs_a'][./b='%@']]/div[@class='gs_rt']/h3[
span='[CITAZIONE]']", cognomeStringa, cognomeStringa]];

...

// VADO ALLA RICERCA DEGLI AUTORI E DELL'ANNO
[temp setString:[NSString
stringWithFormat:@"//span[@class='gs_a'][./b='%@'][../span[@class='gs_fl']] |
//span[@class='gs_a'][./b='%@'][../../div[@class='gs_rt']/h3[span='[CITAZIONE]']]"
, cognomeStringa]];

...

// VADO ALLA RICERCA DEL NUMERO DI CITAZIONI
[temp setString:[NSString
stringWithFormat:@"//div[@class='gs_r'][./font/span[@class='gs_a'][./b='%@']]/font
/span[@class='gs_fl']/a[1] |
//div[@class='gs_r'][./font/span[@class='gs_a'][./b='%@']][./div[@class='gs_rt']/h
3[span='[CITAZIONE]']][not(./font/span[@class='gs_fl'])]/font", cognomeStringa,
cognomeStringa]];
```

La stringa *cognomeStringa* contiene il cognome dello scienziato in questione.
Le tre query formate verranno date in input al parser *SSJuju* che restituirà un *NSArray* per ogni query con all'interno i risultati. E' stato necessario inoltre pulire in qualche modo i risultati ottenuti tramite espressioni regolari.
Il metodo ritorna "*OK*" se la ricerca è andata a buon fine, "*Disconnesso*" se l'iPhone non è collegato ad alcuna rete, "*Pubblicazioni non trovate*" se la ricerca non ha prodotto alcun risultato o "*Ricerca fallita*" se la ricerca è andata a buon fine ma ci sono delle incongruità di numero di titoli, autori/anno o numero di citazioni trovati.

Attraverso il metodo privato *loadPublicationsWithSearch:withData:withPercent:* vengono estrapolati tutti i titoli prima ed eventualmente gli autori poi dall'homepage di un generico scienziato. La ricerca dei titoli/autori viene determinata dal primo parametro in input, una *NSString*, che può quindi essere "*titoli*" o "*autori*". Il secondo parametro rappresenta la pagina HTML dell'homepage mentre il terzo parametro indica la percentuale di similarità dei titoli/autori immessa dall'utente. All'interno di questo metodo è implementato l'algoritmo intelligente dell'estrapolazione delle pubblicazioni spiegato nel capitolo precedente.

5.2 Classe DBPublication

La classe *DBPublication* è la classe che gestisce tutte le interazioni con il database *sqlite*.
Il metodo *checkAndCreateDatabase* verifica la presenza del database e, se non lo trova, crea materialmente il file e lo rende disponibile all'uso.
Il metodo *elementsOfDB:* restituisce le pubblicazioni da Google Scholar/homepage presenti nel database. L'unico parametro in input, una *NSString*, serve a distinguere quali pubblicazioni restituire:
"*publicationsFromGoogleScholar*" restituisce le pubblicazioni da Google Scholar;

"*publicationsFromHomepage*" restituisce le pubblicazioni dall'homepage. Le due query che vengono eseguite per restituire le pubblicazioni da Google Scholar e dall'homepage sono le seguenti:

```
if ([table isEqualToString:@"publicationsFromHomepage"]) {
      sqlStatement = "SELECT * FROM publicationsFromHomepage";
}
else {
      sqlStatement = "SELECT * FROM publicationsFromGoogleScholar ORDER BY
                       numCitazioni DESC";
}
```

Il metodo *numberOfElements:* restituisce il numero di pubblicazioni presenti nel database. L'unico parametro in input serve a distinguere su quali pubblicazioni il metodo deve lavorare. La query che viene eseguita da *sqlite* è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"SELECT COUNT (*) FROM %@",
table];
```

La variabile *table* è una stringa rappresentante o la tabella da Google Scholar o la tabella dell'homepage.

Il metodo *deleteItem:* è quello delegato alla cancellazione di una pubblicazione con indice *pk*, primo parametro in ingresso, da Google Scholar/homepage a seconda del secondo parametro *table* in ingresso. La query è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"DELETE FROM %@ WHERE pk=%i",
                    table, number];
```

Il metodo *deleteDB:* elimina tutte le pubblicazioni presenti da Google Scholar/homepage a seconda dell'unico parametro in ingresso *table*. La query eseguita è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"DELETE FROM %@", table];
```

Il metodo *deleteDuplicate:* è quello delegato alla cancellazione delle pubblicazioni duplicate cioè elimina quelle pubblicazioni che sono presenti più di una volta. Questo metodo, seppur possa essere usato per eliminare le pubblicazioni duplicate sia da Google Scholar e sia dall'homepage, nell'applicazione viene usato soltanto per eliminare le pubblicazioni duplicate dall'homepage. La query eseguita è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"DELETE FROM %@ WHERE pk NOT IN
                    (SELECT MIN(pk) FROM %@ GROUP BY titolo)", table, table];
```

Il metodo *addItemFromGoogleScholar:* aggiunge una pubblicazione estrapolata da Google Scholar, avente titolo, autori, anno e numero di citazioni, nel database. La query eseguita è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"INSERT INTO
  publicationsFromGoogleScholar (numCitazioni,titolo,autori,anno,isChecked)
  VALUES (%i, '%@', '%@', %i, 1)", numCitazioni, titolo, autori, anno];
```

Il metodo *changeChecked:* è quello deputato al cambio del valore *isChecked* di una pubblicazione da Google Scholar da "*0*" a "*1*" o viceversa rispettivamente nell'atto in cui l'utente abilita (la pubblicazione concorre all'H-Index) o disabilita (la pubblicazione non concorre più all'H-Index) quella pubblicazione. La query eseguita è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"UPDATE
  publicationsFromGoogleScholar SET isChecked = %i WHERE pk = %i", isChecked, pk];
```

Il metodo *addTitlesFromHomepage:* aggiunge una pubblicazione avente solo il titolo estrapolata dall'homepage nel database. La query eseguita è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"INSERT INTO
  publicationsFromHomepage (titolo,autori,isPresentInGoogleScholar) VALUES ('%@',
  '', 0)", titolo];
```

Il metodo *changePresentInGoogleScholar:* è quello deputato alla modifica del valore *isPresentInGoogleScholar* di una pubblicazione dall'homepage da "*0*" a "*1*" quando quella determinata pubblicazione è presente anche in Google Scholar. La query eseguita è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"UPDATE publicationsFromHomepage
  SET isPresentInGoogleScholar = %i WHERE pk = %i", isPresentInGoogleScholar, pk];
```

Il metodo *addAuthorsFromHomepage:* aggiunge ad una pubblicazione già presente nel database i relativi autori. La query eseguita è la seguente:

```
NSString *stringa = [NSString stringWithFormat:@"UPDATE publicationsFromHomepage
  SET autori = '%@' WHERE pk = %i", authors, pk];
```

5.3 Classe FirstViewController

La classe FirstViewController gestisce tutti i componenti grafici e le relative azioni della vista di ricerca, quella che compare all'apertura dell'applicazione. Di seguito verranno descritti i metodi più significativi della classe.

Il metodo *buttonSearch:* implementa l'azione relativa alla pressione del bottone "*SEARCH*". In particolare fa partire un thread che si preoccuperà della ricerca delle pubblicazioni da Google Scholar ed eventualmente anche dall'homepage dello scienziato di riferimento. Il codice relativo a questo metodo è il seguente:

```
//IMPLEMENTO L'AZIONE DEL PULSANTE SEARCH
- (void)buttonSearch:(id)sender {
      [Singleton setEsci:FALSE];
      [activityIndicatorView setHidden:NO];
      [activityIndicatorView startAnimating];
      [buttonCancelThread setHidden:NO];

      // AVVIO IL THREAD CHE EFFETTUERA' LA RICERCA DELLE PUBBLICAZIONI
      [NSThread detachNewThreadSelector:@selector(searchPublications)
                                 toTarget:self
                               withObject:nil];
}
```

Il metodo *searchPublications* è in realtà il thread che viene invocato dal metodo *buttonSearch:* che fa partire la ricerca delle pubblicazioni.

Il metodo *updateUILabelWithString:* è in realtà un thread che si preoccupa di aggiornare la *UILabel* rappresentante le informazioni di stato dell'applicazione. Il codice del thread è il seguente:

```
// THREAD CHE AGGIORNA LA LABEL CON IL TESTO "text"
- (void)updateUILabelWithString:(NSString *)text {
      @synchronized(self) {
        labelActivityIndicatorView.text = [NSString stringWithString:text];
      }
}
```

Il metodo *showActionSheetWithTitle:* mostra, a seguito di un errore, un popup con tutte le informazioni necessarie per risolvere quel determinato errore.

Il metodo *buttonCancelThread:* è in realtà un thread che viene chiamato quando l'utente spinge il bottone "*Cancel*" per terminare la ricerca delle pubblicazioni dall'homepage. Il thread modifica lo stato di una variabile condivisa che verrà poi letta, dalla classe *PublicationsParser*. La scrittura e la lettura di questa variabile

viene fatta tramite la classe *Singleton* che verrà spiegata più avanti. Il codice relativo a questo thread è il seguente:

```
// THREAD CHE TERMINA LA RICERCA DELLE PUBBLICAZIONI DALL'HOMEPAGE
- (IBAction)buttonCancelThread:(id)sender {
      [Singleton setEsci:TRUE];
      [self performSelectorOnMainThread:@selector(updateUILabelWithString:)
                                 withObject:@"Terminating..."
                              waitUntilDone:NO];
}
```

5.4 Classe EditDistance

La classe EditDistance è in realtà una categoria che estende le funzionalità della *NSString* grazie al metodo *compareWithString:withPercent:*. Questo metodo implementa l'algoritmo Edit Distance spiegato nel primo capitolo tra la stringa sulla quale è chiamato il metodo e un'altra stringa specificata nel primo parametro. Il secondo parametro in input rappresenta la percentuale di similarità con la quale si vogliono confrontare le due stringhe. Il metodo ritorna un booleano: *TRUE* se le due stringhe sono simili con una percentuale superiore a quella data in input; *FALSE* al contrario.

5.5 Classe Singleton

La classe Singleton implementa uno dei più importanti design pattern della programmazione ad oggetti. In particolare la classe Singleton implementa un design pattern di tipo creazionale. Il suo scopo è quello di garantire che, durante il ciclo di vita dell'applicazione, esista una ed una sola istanza di questa classe ma anche quello di fornire un punto di accesso globale a tale istanza.
In questo caso, tramite i metodi pubblici *setEsci:* e *getEsci* è possibile rispettivamente settare il valore di una variabile booleana condivisa e leggerne il valore. L'accesso alla variabile condivisa viene fatta tramite l'operatore *@synchronized* spiegato nel capitolo 1 che di fatto "protegge" la variabile da accessi indesiderati durante una lettura/scrittura.

La variabile condivisa viene settata dall'utente quando preme il bottone "*Cancel*" per fermare la ricerca delle pubblicazioni dall'homepage e viene letta dalla classe *PublicationsParser*. Il codice relativo a questi due metodi è il seguente:

```
// SET DELLA VARIABILE ESCI
+ (void) setEsci:(BOOL)valore {
      @synchronized(self) {
        esci = valore;
      }
}


// GET DELLA VARIABILE ESCI
+ (BOOL) getEsci {
      @synchronized(self) {
        return esci;
      }
}
```

5.6 Classe TableFromGoogleScholarViewController

La classe TableFromGoogleScholarViewController gestisce tutti i componenti grafici e le relative azioni della vista in cui è presente la tabella con tutte le pubblicazioni estrapolate da Google Scholar.

Il principale metodo di questa classe è *calculateHIndex:* che calcola l'H-Index dello scienziato in questione. Il metodo prende in input un *NSArray* con tutte le pubblicazioni da Google Scholar e restituisce il relativo H-Index. Il codice del metodo sopra citato viene riportato di seguito:

```
// METODO CHE CALCOLA L'H-INDEX DELLE PUBBLICAZIONI DA GOOGLE SCHOLAR
- (int)calculateHIndex:(NSMutableArray *)elementsOfDB {
      int hIndex = 0;
      int numPubblicazioniEsaminate = 1;
      int i;
      for (i = 0; i < [elementsOfDB count] && [[[elementsOfDB objectAtIndex:i]
       objectForKey:@"numCitazioni"] intValue] >= numPubblicazioniEsaminate; i++)
      {
        if ([[[lista objectAtIndex:i] objectForKey:@"isChecked"] intValue] == 1) {
                hIndex++;
                numPubblicazioniEsaminate++;
        }
      }
      indiceUltimaPubblicazioneInHIndex = i - 1;
      HIndex = hIndex;

      return hIndex;
}
```

La classe implementa anche i due protocolli relativi alla *UITableView*: *UITableViewDelegate* e *UITableViewDataSource*. Un metodo molto importante

relativo al *UITableViewDelegate* è *tableView:didSelectRowAtIndexPath:* che consente di sapere quando viene cliccata la cella. Nel caso dell'applicazione corrente, questo metodo cambia il checkmaker (segno di spunta) e ricalcola l'H-Index. Di seguito viene riportato il codice di questo metodo:

```
// METODO CHE VIENE CHIAMATO QUANDO VIENE SELEZIONATA O DESELEZIONATA UNA
CHECKMARK
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {

    [tableView deselectRowAtIndexPath:[tableView indexPathForSelectedRow]
        animated:NO];

    if ([[tableView cellForRowAtIndexPath:indexPath] accessoryType] ==
      UITableViewCellAccessoryCheckmark) {
        [[tableView cellForRowAtIndexPath:indexPath]
          setAccessoryType:UITableViewCellAccessoryNone];
        [DBPublication changeChecked:[[[lista objectAtIndex:indexPath.row]
          objectForKey:@"id"] intValue] :0];
        [[lista objectAtIndex:indexPath.row] setObject:@"0" forKey:@"isChecked"];
    }
    else {
        [[tableView cellForRowAtIndexPath:indexPath]
          setAccessoryType:UITableViewCellAccessoryCheckmark];
        [DBPublication changeChecked:[[[lista objectAtIndex:indexPath.row]
          objectForKey:@"id"] intValue] :1];
        [[lista objectAtIndex:indexPath.row] setObject:@"1" forKey:@"isChecked"];
    }
    [H_Index setText:[NSString stringWithFormat:@"H-INDEX: %i", [self
      calculateHIndex:lista]]];
    [myTable reloadData];
}
```

Inoltre per visualizzare il contenuto di una determinata pubblicazione all'interno di una cella è stato usato un metodo appartenente al protocollo *UITableViewDataSource*: *tableView:cellForRowAtIndexPath:*. Questo metodo viene invocato ogniqualvolta viene visualizzata una cella. Il codice relativo a questo metodo viene riportato di seguito.

```
// METODO CHE VIENE CHIAMATO QUANDO VIENE VISUALIZZATA UNA CELLA
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    // IMPORTO LA CELLA CREATA NELLA CLASSE "CellTableFromGoogleScholar"
    CellTableFromGoogleScholar *cell = (CellTableFromGoogleScholar *) [tableView
     dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        NSArray *topLevelObjects = [[NSBundle mainBundle]
          loadNibNamed:@"CellTableFromGoogleScholar" owner:self options:nil];
        for (id currentObject in topLevelObjects) {
                if ([currentObject isKindOfClass:[UITableViewCell class]]) {
                        cell = (CellTableFromGoogleScholar *) currentObject;
                        break;
                }
        }
    }
```

```
    // SCRIVO SULLA CELLA: TITOLO, AUTORI, ANNO E NUMERO DI CITAZIONI
    [[cell autori] setText:[[lista objectAtIndex:[indexPath row]]
      objectForKey:@"titolo"]];
    [[cell titolo] setText:[[lista objectAtIndex:[indexPath row]]
      objectForKey:@"autori"]];
    [[cell anno] setText:[[[lista objectAtIndex:[indexPath row]]
      objectForKey:@"anno"] isEqualToString:@"0"] ? @"" : [[lista
      objectAtIndex:[indexPath row]] objectForKey:@"anno"]];
    [[cell numCitazioni] setText:[NSString stringWithFormat:@"Quoted: %@", [[lista
      objectAtIndex:[indexPath row]] objectForKey:@"numCitazioni"]]];

    // IMPOSTO O NO IL CHECKMARK
    if ([[[lista objectAtIndex:[indexPath row]] objectForKey:@"isChecked"]
      intValue] == 1) {
       cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }
    else {
        cell.accessoryType = UITableViewCellAccessoryNone;
    }

    return cell;
}
```

L'ultimo metodo che viene presentato è *tableView:WillDisplayCell:forRowAtIndexPath:*, appartenente al protocollo *UITableViewDataSource*, che viene invocato dopo il metodo *tableView:cellForRowAtIndexPath:* spiegato precedentemente. Questo metodo ha il compito di colorare la cella contenente una determinata pubblicazione a seconda che concorra o meno all'H-Index. Di seguito è riportato il codice di questo metodo:

```
// METODO CHE VIENE CHIAMATO QUANDO VIENE VISUALIZZATA UNA CELLA
- (void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell
forRowAtIndexPath:(NSIndexPath *)indexPath {
      if ([self isInLimbo:cell :indexPath]) {
        cell.backgroundColor = [UIColor lightGrayColor]; // grigio
      }
      else if ([self isInHIndex:cell :indexPath]) {
        cell.backgroundColor = UIColorFromRGB(0x99FF66); // verde
      }
      else {
        cell.backgroundColor = UIColorFromRGB(0xFF9999); // rosso
      }
}
```

Il metodo descritto si serve di altri metodi per colorare una determinata cella: *isInLimbo:* che ritorna *TRUE* se la pubblicazione deve essere colorata di grigio; *isInHIndex* che ritorna *TRUE* se la pubblicazione concorre all'H-Index (deve essere colorata di verde).

5.7 Classe TableFromHomepageViewController

La classe TableFromHomepageViewController gestisce la tabella con tutte le pubblicazioni estrapolate dall'homepage di un certo scienziato. In particolare l'unico compito che ha questa classe è quello di visualizzare le pubblicazioni estrapolate dall'homepage nella tabella e colorarle di verde o di rosso a seconda se sono presenti o meno in Google Scholar.

Come per la classe *TableFromGoogleScholarViewController*, questa classe implementa i due protocolli relativi alla *UITableView*: *UITableViewDelegate* e *UITableViewDataSource*.

Il metodo che visualizza le pubblicazioni nelle varie celle è, come la precedente classe, il metodo *tableView:cellForRowAtIndexPath:* appartenente al protocollo *UITableViewDataSource*. Di seguito è riportato il codice:

```
// METODO CHE VIENE CHIAMATO QUANDO VIENE VISUALIZZATA UNA CELLA
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

      // IMPORTO LA CELLA CREATA NELLA CLASSE "CellTableFromHomepage"
    CellTableFromHomepage *cell = (CellTableFromHomepage *) [tableView
      dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        NSArray *topLevelObjects = [[NSBundle mainBundle]
          loadNibNamed:@"CellTableFromHomepage" owner:self options:nil];
        for (id currentObject in topLevelObjects) {
                if ([currentObject isKindOfClass:[UITableViewCell class]]) {
                        cell = (CellTableFromHomepage *) currentObject;
                        break;
                }
        }
    }

    // SCRIVO SULLA CELLA TITOLO ED EVENTUALI AUTORI
    if ([[[lista objectAtIndex:[indexPath row]] objectForKey:@"autori"]
      isEqualToString:@""]) {
        [[cell unicoTitolo] setText:[[lista objectAtIndex:[indexPath row]]
          objectForKey:@"titolo"]];
        [[cell titolo] setText:@""];
        [[cell autori] setText:@""];
      }
      else {
        [[cell unicoTitolo] setText:@""];
        [[cell titolo] setText:[[lista objectAtIndex:[indexPath row]]
          objectForKey:@"titolo"]];
        [[cell autori] setText:[[lista objectAtIndex:[indexPath row]]
          objectForKey:@"autori"]];
      }

    return cell;
}
```

Il metodo che ha il compito di colorare una determinata cella di verde o di rosso, è, come la precedente classe, il metodo *tableView:WillDisplayCell:forRowAtIndexPath:* appartenente al protocollo *UITableViewDataSource*. Di seguito è riportato il codice di questo metodo:

```
// METODO CHE VIENE CHIAMATO QUANDO VIENE VISUALIZZATA UNA CELLA
- (void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell
forRowAtIndexPath:(NSIndexPath *)indexPath {
      if ([[[lista objectAtIndex:[indexPath row]]
       objectForKey:@"isPresentInGoogleScholar"] intValue] == 1) {
         cell.backgroundColor = UIColorFromRGB(0x99FF66); // verde
      }
      else {
         cell.backgroundColor = UIColorFromRGB(0xFF9999); // rosso
      }
}
```

Conclusione e sviluppi futuri

Durante lo sviluppo di questa tesi, si è giunti alla creazione di un'applicazione che calcoli in maniera quasi ottimale l'H-Index di un qualsiasi scienziato appoggiandosi alla piattaforma di Google Scholar.
Il punto di forza dell'applicazione sta nel prendere in considerazione solo quelle pubblicazioni che sono state effettivamente pubblicate dallo scienziato in questione, scartando quindi tutte quelle pubblicazioni definite "sporche", comunque restituite da Google Scholar. Questa pulizia avviene grazie all'aiuto delle pubblicazioni prelevate dalla pagina web dello scienziato in questione.

Per quanto riguarda gli sviluppi futuri, si potrebbe aggiungere la funzionalità di modifica delle pubblicazioni estrapolate dall'homepage dello scienziato di riferimento e convogliare tali modifiche alle pubblicazioni da Google Scholar.
Un'altra funzionalità che è possibile aggiungere, potrebbe essere quella di estrapolare tutte le pubblicazioni presenti all'interno di una homepage che è mal formattata, cioè che presenta i titoli/autori in maniera non omogenea (ad esempio una parte dei titoli/autori è sottolineata e l'altra no, una parte dei titoli/autori è in grassetto e l'altra no, ecc.).
Un'ultima miglioria che è possibile aggiungere potrebbe essere quella di modificare i parser utilizzati affinché possano parsare qualunque tipo di pagina HTML/XML.

Bibliografia

[1] Ricercatori o tennisti: H-Index come ATP Ranking?: http://unipiblog.wordpress.com/2011/02/09/ricercatori-o-tennisti-h-index-come-atp-ranking/

[2] Indice H: http://it.wikipedia.org/wiki/Indice_H

[3] h-index: http://en.wikipedia.org/wiki/H-index

[4] An index to quantify an individual's scientific research output: http://www.pnas.org/content/102/46/16569.full

[5] iOS (Apple): http://it.wikipedia.org/wiki/IOS_(Apple)

[6] Teoria: Il sistema operativo iPhone OS e la sua tecnologia: http://www.iphoneandgo.it/2008/09/01/teoria-il-sistema-operativo-iphone-os-e-la-sua-tecnologia-core-os-core-service-media-e-cocoa-touch-prima-parte-n5/

[7] Design pattern: http://it.wikipedia.org/wiki/Design_pattern

[8] Model-View-Controller: http://it.wikipedia.org/wiki/Model-View-Controller

[9] Design Patterns: implementiamo il Singleton in Objective-C: http://www.iosdeveloper.it/2011/06/05/design-patterns-implementiamo-il-singleton-in-objective-c/

[9] L#017 – Il ciclo di vita delle applicazioni iOS:
http://www.devapp.it/wordpress/l017-il-ciclo-di-vita-delle-applicazioni-ios.html#more-7032

[10] Le proprietà:
http://programmazione.html.it/guide/lezione/4617/le-proprieta/

[11] L#018 – Objective-c: Le *@property* e altre amenità:
http://www.devapp.it/wordpress/l018-objective-c-le-property-e-altre-amenita.html

[12] Objective C: property e synthesize:
http://www.mrwebmaster.it/mobile/guide/objective-property-synthesize_1115.html

[13] Guida Objective-C in Italiano - Parte 7 - Categorie ed Estensioni:
http://emdief.blogspot.com/2010/04/guida-objective-c-in-italiano-parte-7.html

[14] How To Choose The Best XML Parser for Your iPhone Project:
http://www.raywenderlich.com/553/how-to-chose-the-best-xml-parser-for-your-iphone-project

[15] iPhone programming: come fare HTML scraping senza (troppa) fatica:
http://www.lorenzoridi.net/2009/09/iphone-programming-come-fare-html-scraping-senza-troppa-fatica/

[16] Il linguaggio XPath:
http://www.giuseppesicari.it/articoli/xpath/

[17] Edit Distance o Levenshtein Distance: un algoritmo di similarità tra

stringhe:
http://www.appuntisoftware.it/edit-distance-o-levenshtein-distance-un-algoritmo-di-similarita-tra-stringhe/

[18] Programmare applicazioni per iPhone e iPad – Francesco Novelli

[19] Sviluppare applicazioni con iPhone SDK – Bill Dudney, Chris Adamson

[20] iOS Application Programming Guide