ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA

Seconda Facoltà di Ingegneria con sede a Cesena
Corso di Laurea in Ingegneria Informatica

PROGETTAZIONE E SVILUPPO DI APPLICAZIONI MOBILE SU iOS: ANALISI DEGLI ASPETTI FONDAMENTALI

Elaborata nel corso di: Sistemi Operativi

Relatore:
Prof. ALESSANDRO RICCI

Tesi di Laurea di:
PIETRO BRUNETTI

ANNO ACCADEMICO 2010-2011
SESSIONE III

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Uno smartphone, in italiano "cellulare intelligente", è un dispositivo portatile che abbina funzionalità proprie di un telefono cellulare a quelle di gestore di dati personali per migliorare la produttività personale ed aziendale; costruito su una piattaforma di mobile computing, è caratterizzato da una elevate capacità computazionale e connettività rispetto ai normali telefoni cellulari.

Uno degli aspetti peculiari per gli smartphone è la possibilità di installarvi ulteriori applicazioni, aggiungendo così nuove funzionalità. Tali dispositivi si differenziano inoltre per l'adozione di APIs (application programming interfaces) avanzate, permettendo alle applicazioni di terze parti una migliore integrazione con il sistema operativo del dispositivo.

In questi ultimi anni si è notato un rapido avanzamento nel campo dell'innovazione dei sistemi operativi e delle piattaforme applicative orientate ai dispositivi mobile, dato il costante e rapido incremento della capacità computazionale dei processori, reso possibile grazie alle fiorenti tecnologie disponibili. Tutto questo ha permesso di definire un livello ancora più alto di "intelligenza" nei dispositivi, dove la geo–localizzazione, basata su differenti tecnologie come GPS, Wi–Fi, GSM, si presenta ora come una funzionalità indispensabile; i sistemi operativi associati a questa fascia di prodotti offrono tutti un *market place* o uno *store* dove poter scaricare applicazioni di ogni tipo e per qualsiasi utilizzo, cambiando radicalmente la percezione che l'utente finale ha di tali dispositivi: da semplici cellulari si giunge a considerarli veri e propri computer in miniatura.

Un passo immediatamente successivo sarà caratterizzato dalla presenze di un livello di consapevolezza riferita al contesto applicativo, maturata all'interno dei dispositivi per poter non solo tener traccia dei dati personali dell'utente, ma soprattutto tracciare e riconoscere il comportamento individuale, riuscendo quindi ad anticipare le intenzioni decisionali dell'u-

tilizzatore. Queste ultime potenzialità descritte vanno a costituire la parte veramente "smart" dei dispositivi; in questo senso questi ultimi assumono anche le sembianze di sensori a tutti gli effetti.

In un ambiente in continua evoluzione come quello elettronico–informatico, dove le potenzialità sembrano essere infinite, sono emerse le prime idee, all'interno delle maggiori aziende produttrici di software, relative alle enormi possibilità offerte da piattaforme software consolidate, applicate a dispositivi con tecnologia pari ai ben noti computer desktop, iniziando ad investire nella realizzazione di sistemi operativi mobile come supporto ai moderni smartphones; tra i più comuni SO si ricordano Google's Android, Apple's iOS, Microsoft's Windows Phone, Nokia's Symbian, RIM's BlackBerry OS e distribuzioni Linux integrate come Maemo e MeeGo.

Il successo di tali investimenti è reso evidente dal numero sempre maggiore di smartphone connessi ad Internet rispetto ai computer; in effetti è in atto un rilevante cambiamento riguardo al modo in cui, sempre più spesso, vengono utilizzate le applicazioni desktop più frequenti, dove la parte di comunicazione assume un ruolo in primo piano, a scapito di azioni e *tasks* legati all'ambiente desktop tradizionale, come la creazione di documenti. Questa tendenza si è osservata sia in ambito lavorativo che nella vita di tutti giorni, facilitando molto l'effettiva traduzione delle applicazioni desktop in applicazioni mobile, nelle quali il fattore "comunicazione" rappresenta uno dei concetti più importanti, specialmente dal punto di vista progettuale.

Lentamente ci si sta avvicinando ad una situazione in cui l'adozione di tali dispositivi mobili "intelligenti" divenga la norma per i normali internauti, dato che molte delle azioni di routine che ora sono caratteristiche dei computer desktop e laptop diverranno molto presto possibili per tutti gli smartphone, e le nuove applicazioni sviluppate per questi ultimi incontrano perfettamente le esigenze degli utenti che non usano computer. In questa ottica l'utilizzo dei computer desktop potrebbero essere facilmente confinato ai soli professionisti e ad una ristretta élite di specialisti, tanto quanto lo sono oggi i mini–computer e super–computer.

Come precedentemente accennato, le applicazioni mobile esistevano già negli anni novanta, anche se con una struttura minimale e sviluppavando applicazioni indipendenti che potevano essere installate su dispositivi specifici, ma il processo stesso era ancora macchinoso e questo aspetto scoraggiava gli utenti finali ad aggiungere applicazioni ai loro dispositivi mobili; quello che mancava era un componente software aggiuntivo, chiamato *market*

place o *application store*, atto a rendere l'acquisto e l'installazione di applicazioni di terze parti il più semplice possibile, tramite un'interfaccia utente comprensibile e di facile utilizzo. In questi ultimi anni alcuni di tali luoghi virtuali sono giunti a contenere centinaia di migliaia di applicazioni; Apple, in particolare, annunciò nel lontano 2009 il raggiungimento di due bilioni di download dall'App Store, decretando il pieno successo di iOS e dando inizio ad una nuova era mobile.

Capitolo 1

Caratteristiche principali di iOS

iOS rappresenta la versione mobile del sistema desktop Mac OS X appartenente alla ben nota azienda internazionale produttrice di hardware–software; è utilizzato in tutti i dispositivi mobili della medesima marca di fascia alta (iPhone, iPod touch, iPad) e mette a disposizione le tecnologie necessarie per implementare applicazioni web e/o native, oltre che dare la possibilità agli utenti di usufruire di un'innumerevole quantità di applicazioni gratuite o a pagamento pronte per essere installate.

Da subito rivoluzionò il modo di pensare e utilizzare tali dispositivi, dando inizio ad una vera e propria "corsa allo smartphone"; giunto alla versione 5.0.1, rappresenta tuttora uno dei sistema di punta in ambiente mobile hi–tech. Il suo successo è stato determinato dalla facilità d'uso e semplicità, due dei maggiori punti di forza, che da sempre hanno reso possibile un utilizzo immediato, rapido, anche grazie ad un'ottima fluidità del sistema; inoltre i modi in cui si è stata ideata l'interfaccia grafica e si sono gestite le interazioni con i componenti hardware per supportare il touchscreen, hanno aumentato il grado di intuitività nei gesti eseguiti in un utilizzo giornaliero. È opportuno ricordare che tali caratteristiche generali non consentono un'alta personalizzazione e manipolazione del dispositivo, limitando la libertà dell'utente finale. Però, così facendo, tale piattaforma mobile risulta essere molto sicura e robusta, rispecchiando una scelta progettuale, non ché una linea di percorrenza dell'azienda produttrice ben precisa.

Come già accennato in precedenza è possibile progettare e sviluppare

applicazioni native mediante interfacce, tools e risorse, dove queste ultime sono molto vincolanti e stringenti; l'aspetto più interessante è la possibilità di giungere ad un livello di performance e ottimizzazione formidabile tramite l'interfacciamento con strutture e tecnologie di basso livello, perseguendo metodi più rischiosi e difficoltosi, oppure affidarsi a metodi "preconfezionati" (la norma, nei maggiori ambienti di sviluppo) che risultano sicuramente più rapidi, ma meno personalizzabili.

Per quanto riguarda gli aspetti architetturali si può notare una grande affinità presente fra l'ambiente desktop e quello mobile, mettendo in luce un buon grado di analisi e organizzazione, ma soprattutto percependo la volontà dell'azienda produttrice di procedere, nel corso degli anni, verso un'unificazione dei due ambienti, al fine di pervenire ad una eterogenea gamma di dispositivi, dal punto di vista dell'equipaggiamento hardware ed obbiettivi specifici, nei quali è in esecuzione il medesimo sistema software sovrastante.

1.1 Struttura Architetturale

L'architettura di iOS è simile a quella basica in Mac OS X e come essa è strutturata in una serie di quattro livelli di astrazione, o *layers*, ognuno dei quali implementa funzionalità ben specifiche, per rendere semplice la scrittura di applicazioni che funzionano in modo coerente su dispositivi con differenti capacità hardware.

Nel livello più alto, il sistema operativo agisce come intermediario fra l'hardware sottostante e l'interfaccia grafica, mentre le varie applicazioni installate comunicano con i livelli sottostanti attraverso un ben determinato set di interfacce, aumentando considerevolmente la protezione delle applicazioni da eventuali modifiche hardware. Il livello più basso del sistema ospita i servizi e le tecnologie fondamentali da cui tutte le applicazioni dipendono.

La maggior parte delle interfacce di sistema sono rese disponibili in pacchetti speciali chiamati *framework*. Un framework consiste in un direttorio contenente una libreria dinamica di funzioni e risorse (header files, immagini etc.) a supporto di essa. In aggiunta ai frameworks, Apple rende disponibili alcune tecnologie nella forma di librerie dinamiche in formato standard; molte di esse sono appartenenti al livello più basso del sistema operativo e derivano da tecnologie Open Source, come naturale conseguenza del fatto

che iOS, similmente al corrispondente sistema desktop di casa Apple, sia basato sulla piattaforma Unix.

1.2 Livelli Architetturali

Procedendo in un'analisi più accurata circa la composizione e la coesione delle stratificazione che vanno a formare l'architettura di iOS, si possono individuare quattro differenti livelli di astrazione, come mostrato in figura, ognuno dei quali verrà trattato in maniera dettagliata nei capitoli seguenti.

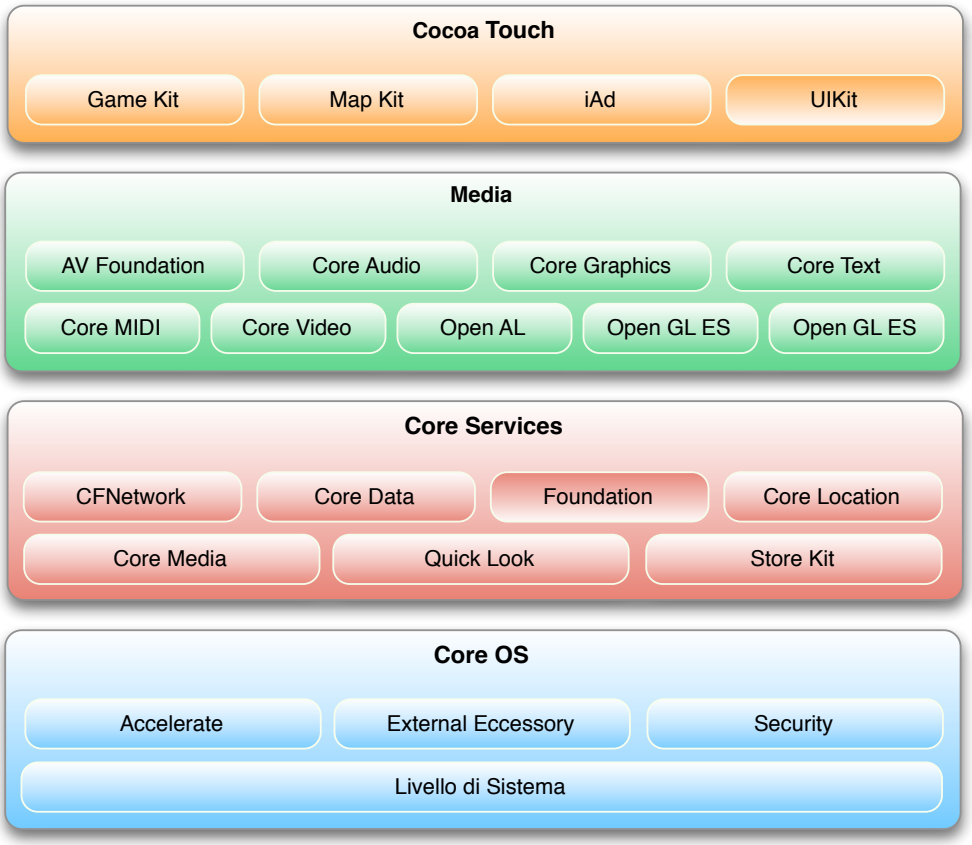

Figura 1.1: Livelli architetturali di iOS

1.2.1 Core OS

È lo strato che permette di lavorare a diretto contatto con l'hardware sottostante ed è considerato il vero cuore del sistema operativo; infatti in esso sono presenti gli elementi considerati fondamentali, utilizzati poi dalle tecnologie di livelli sovrastanti.

Accelerate Framework - contiene le interfacce utilizzate per l'esecuzione di calcoli matematici, DSP ed anche per l'elaborazione di numeri molto grandi. Tale strumento ha il vantaggio di essere ottimizzato per tutte le configurazioni hardware presenti in dispositivi basati su iOS.

External Accessory Framework - offre supporto per la comunicazione con la parte hardware di accessori o componenti collegati a dispositivi basati su iOS.

Security Framework - mette a disposizione interfacce specifiche per gestire certificati, chiavi private o pubbliche e la generazione di numeri crittografati pseudo-casuali; tutto ciò in aggiunta alle caratteristiche di sicurezza già presenti, in modo da garantire un livello di sicurezza personalizzato per i dati delle applicazioni sviluppate.

Livello di Sistema - comprende l'ambiente del kernel, i drivers, e le interfacce Unix di basso livello. la parte centrale, fulcro stesso di Mac OS X e iOS, che include il kernel e la base Unix, è noto come *Darwin*, un sistema operativo open source pubblicato da Apple. Esso non include, come invece è nel caso di Mac OS X, un'interfaccia utente caratteristica, ma fornisce solamente gli strumenti e servizi base relativi all'ambiente dedicato al kernel e dello spazio utente, tipici dei sistemi Unix.

Per quanto concerne il kernel, il sistema operativo Darwin è costituito dal kernel *XNU*; quest'ultimo è costituito da un'architettura a livelli che conta tre componenti principali, come è riportato in figura. L'anello più interno del kernel, se così si può definire, si riferisce al livelli Mach, derivante dal kernel omonimo alla versione 3.0, sviluppato presso la *Carnegie Mellon University*.

Il kernel Mach fu definito come microkernel, pensato come un sottile strato con lo scopo di fornire solamene i servizi fondamentali, come la gestione

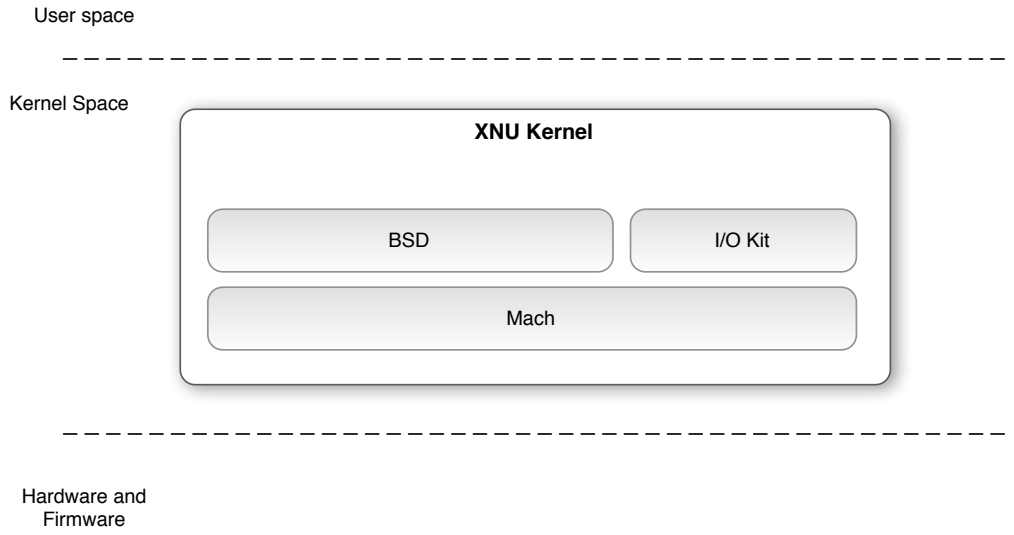

Figura 1.2: Architettura del kernel XNU

dei processori, dei thread e la schedulazione dei tasks, per i due componenti sovrastanti che completano l'XNU, rappresentati dal livello BSD e dall'I/O Kit.

La parte BSD si interpone tra il microkernel Mach e le applicazioni utente, implementando molte funzioni chiave del sistema operativo, quali la gestione dei processi, delle chiamate di sistema, del file system e del collegamento in rete. Tale layer si riferisce ad una parte del kernel che deriva dal sistema operativo FreeBSD 5, del quale ne rappresenta una porzione di codice e non un sistema completo in sé.

Il layer chiamato I/O Kit è in realtà un framework Object-Oriented necessario alla scrittura di drivers per i dispositivi ed altre estensioni del kernel; in particolare prevede un'astrazione dei sistemi hardware con classi base predefinite per supportarne diversi tipi, rendendo più semplice l'implementazione di nuovi drivers.

I drivers presenti nel livello di sistema consentono di fornire un'interfaccia interposta fra la specifica parte hardware e i frameworks di sistema anche se, per motivi di sicurezza, l'accesso al kernel e ai suddetti drivers è ristretto ad un set limitato di framework ed applicazioni. In fine, le interfacce Unix in questione sono basate sul linguaggio C e forniscono supporto per il Threading (threads POSIX), la gestione della rete (sockets BSD), l'accesso

al filesystem, la gestione dell'alimentazione, informazioni di localizzazione, gestione della memoria e relativa allocazione.

1.2.2 Core Services

Come suggerisce il nome assegnato, questo livello contiene i servizi di sistema fondamentali, utilizzati da tutte le applicazioni, spesso considerate utility. Le tecnologie chiave presenti si possono riassumere nelle aree di interesse che comprendono la programmazione concorrente, il commercio elettronico, la gestione e memorizzazione dei dati all'interno di database e presentazione/manipolazione delle informazione ricevute o trasmesse.

Grand Central Dispatch (GCD) - introdotto nella versone 4.0 di iOS, rappresenta una tecnologia molto utilizzata per gestire l'esecuzione delle varie attività (task) all'interno delle singole applicazioni. Tale tecnologia verrà trattata in maniera esaustiva nella seconda parte del documento; ad ogni modo offre un'alternativa più conveniente ed efficiente rispetto al concetto di threading.

Acquisti in–App - implementato tramite lo Store Kit Framework, il quale mette a disposizione le infrastrutture necessarie per i processi che coinvolgono le transazione finanziarie, utilizzate all'interno dell'iTunes Store. Esso offre la possibilità di vendere contenuti e servizi all'interno delle singole applicazioni, senza dover ricorrere obbligatoriamente ad una gestione centralizzata dei contenuti commerciali.

SQLite - costituita da una libreria dedicata, permette di incorporare una versione "lite" del database SQL nelle applicazioni, senza bisogno di porre in esecuzione separatamente un database remoto all'interno di un processo server; in particolare è possibile gestire tabelle e tuple attraverso un database locale contenuto in appositi files. Inoltre, anche se tale libreria è stata ideata per un uso generico, può essere ottimizzata al fine di innalzare il livello di performance.

Supporto XML - tramite il Foundation Framework, mette a disposizione una classe che incapsula il concetto di parser per riuscire a trattare il testo all'interno del linguaggio di markup che rappresenta e descrive l'intera struttura di un documento. Per default si possono utilizzare solo parser di tipo "event-driven" per ragioni legate alla performance

delle operazioni interessate e all'utilizzo di memoria; in questi casi si richiama il costrutto di un documento XML in maniera sequenziale, tramite un meccanismo che si basa sulla ricezione ed invio di eventi.

Nelle tecnologie sopra descritte è spesso indispensabile l'utilizzo dei frameworks presenti in questo livello, alcuni di essi riportati di seguito.

CFNetwork Framework - basato su socket BSD, è costituito da un insieme interfacce in linguaggio C molto performanti, utilizzando paradigmi orientati agli oggetti per lavorare con svariati protocolli di rete; la caratteristica di rilievo è rappresentata dalla possibilità di ottenere un controllo dettagliato dei vari stack di protocolli e rendere molto semplice il loro utilizzo grazie alle astrazione introdotte.

Core Data Framework - costituisce una tecnologia per la gestione del modello dei dati in applicazioni caratterizzate dall'utilizzo del pattern MVC. In linea di principio è destinato ad un utilizzo nello sviluppo di applicazioni in cui il modello dei dati previsto è altamente strutturato.

Foundation Framework - insieme ad UIKit è uno dei framework essenziali per la piattaforma mobile iOS, mentre tutti gli altri sono secondari e non indispensabili; ne sono un esempio le applicazioni a linea di comando, le quali utilizzano solo tali due framework.

Le funzioni svolte ed i ruoli assunti dal framework Foundation si possono riassumere nella definizione di un livello base di classi che possono essere utilizzate per ogni tipo di programma Cocoa, (il cui significato verrà chiarito ampiamente nei capitoli successivi) punto di incontro fra i due sistemi operativi Apple.

Per quanto riguarda gli obbiettivi, esso si preoccupa di definire il comportamento base degli oggetti utilizzati ed introduce per essi delle convenzioni coerenti quali la gestioni della memoria, il sistema di notifiche e il supporto alla loro mutevolezza, persistenza e distribuzione. Oltre a ciò, da supporto per il sistema di localizzazione e fornisce alcune misure per rendere maggiormente portabile il sistema operativo stesso. Ad esso è strettamente legato il *Core Foundation Framework*, costituito da un insieme di interfacce basate sul linguaggio C, indirizzato alla gestione dei dati e servizi in applicazioni iOS in modo più specifico rispetto al Foundation Framework.

Core Location Framework - permette di accedere alle informazioni di localizzazione e posizionamento all'interno delle applicazioni; in particolare fa uso della tecnologia GPS, (se è compreso nella dotazione hardware) o delle onde Wi-Fi, per ottenere la longitudine e latitudine correnti. Un esempio di utilizzo è dato da una ricerca eseguita anche per prossimità rispetto alla posizione geografica realtiva del dispositivo.

Core Media Framework - include in se gli strumenti e i tipi di comunicazione audio/video di basso livello, utilizzati nel livello architetturale sovrastante, per esempio dal *AV Foundation Framework*; quest'ultimo è uno dei tanto framework da utilizzare per riprodurre e creare mezzi di comunicazione audio-visiva, anche in tempo reale. Mentre l'uso di AV Foundation è raccomandato, tipicamente lo sviluppo di applicazioni non necessita dell'utilizzo del Core Media Framework; infatti sono pochi gli sviluppatori che ricorrono ad esso per ottenere un controllo preciso in caso di creazione e presentazione di contenuti audio/video.

Quick Look Framework - fornisce un interfaccia diretta per la visualizzazione di un'anteprima dei files non supportati direttamente; tale strumento risulta utile in applicazioni che scaricano contenti dalla rete o sono portati a lavorare con files la cui fonte è sconosciuta.

Store Kit Framework - offre supporto nella compravendita di contenuti e servizi che avvengono all'interno delle singole applicazioni; in particolare tale framework si focalizza sull'aspetto finanziario della transazione, assicurando che sia avvenuta correttamente e nel modo più sicuro possibile.

1.2.3 Media

Rappresenta lo strato che contiene tutte le funzionalità e le librerie per la gestione di video e audio. Mediante le tecnologie presenti, si è orientati verso la creazione della migliore esperienza multimediale raggiungibile su un dispositivo mobile. Inoltre, è necessario l'inserimento di una nota per quanto riguarda una recente tecnologia sviluppata chiamata *AirPlay*, la quale permette lo streaming audio verso la Apple TV oppure verso altoparlanti AirPlay di terze parti. Il supporto AirPlay è integrato in AV Foundation

e Core Audio Framework; questo implica che qualunque contenuto audio riprodotto usando una dei due framework sopra citati è automaticamente reso idoneo per la distribuzione tramite AirPlay.

AV Foundation Framework - utilizzato per la riproduzione e manipolazione di contenuti audio; con esso è possibile avere un ampio controllo su vari aspetti dei suoni riprodotti, come la possibilità di registrare audio e gestire le informazioni sulle sessioni sonore acquisite.

Core Audio - offre supporto nativo per le operazioni sull'audio; in particolare supporta la manipolazione di audio a qualità stereo, la generazione, registrazione e mix nonché riproduzione dell'audio risultante.

Core Graphics, MIDI, Text, Video Frameworks - come tutti i frameworks presenti in questo livello architetturale, permettono di avere un ampio controllo e possibilità di espressione nelle rispettive aree di interesse.

OpenAL (AudioLibrary) Framework - rappresenta uno standard inter–piattaforma per posizionare fonti audio in modo tridimensionale, tenendo in considerazione i parametri di disturbo che potrebbero influire. Viene utilizzato per implementare giochi o altre applicazioni che richiedano un audio posizionale in output, caratterizzate da alte prestazioni e soprattutto con un audio di altissima qualità.

OpenGL (Graphical Library) ES Framework - è una potente libreria grafica, pensata per interfacciarsi direttamente con l'hardware e fornire al singolo programma una serie di primitive, più o meno essenziali, per lo sviluppo di applicazioni nel campo del rendering 2D e 3D. Le primitive considerate comprendono funzioni per la manipolazione dei pixel, per la proiezione di poligoni in 3D e per la gestione del movimento degli stessi, per la rappresentazione di luci etc.

Quartz Core Framework - contiene le interfacce del *Core Animation*, una tecnologia avanzata per l'animazione e composizione utilizzata come via per il rendering ottimizzato al fine di implementare complesse animazioni ed effetti visivi.

1.2.4 Cocoa Touch

Rappresenta lo strato più vicino all'applicazione utente e i frameworks di questo livello supportano direttamente le applicazioni basate su iOS. Esso si occupa della gestione del touch e multi-touch, interpretando i differenti gesti (gestures) compiuti dall'utente finale mediante i *gesture recognizers*, oggetti collegati alle *view* (a loro volta definite come schermate visibili sul video), utilizzati per rilevare i tipi più comuni di gestures, come lo zoom–in o la rotazione di elementi; non appena collegati si può stabilire quale comportamento associare ad essa.

Oltre alla gestione del touch, molti dei frameworks del livello Cocoa Touch contengono specifiche classi genericamente denominate *View Controller*, per poter visualizzare interfacce standard di sistema. Tali componenti rappresentano particolari tipi di *controller* molto utilizzati per presentare e gestire un insieme di view; giocano un ruolo importante nella progettazione ed implementazione di applicazioni iOS perché forniscono un'infrastruttura per gestire i contenuti correlati alle view e coordinare la comparsa/scomparsa di queste ultime.

Un'ulteriore caratteristica di questo livello è data dal supporto a due differenti modalità in cui utilizzare le notifiche, le quali danno la possibilità di avvisare l'utente di nuove informazioni mediate un segnale sonoro o visivo. Il primo prende il nome di *Apple Push Notification Center*; utilizza un processo server per generare e distribuire la notifica inviatagli dal client, mentre Il secondo, *Notifiche Locali*, completa il meccanismo di notifiche sopra descritto dando la possibilità, in fase di progettazione, di generarle in locale, senza fare affidamento su un server esterno.

In fine, molto importante per gli argomenti correlati alle prestazioni di sistema, è il pieno supporto dato al Multitasking anche ad alto livello architetturale, per poter coordinare ed impostare azioni tipiche del livello applicativo, come la ricezione di notifiche, il passaggio da uno stato attivo ad uno passivo.

Come in tutti gli altri livelli in cui è struttura iOS, anche in Cocoa Touch si utilizzano innumerevoli frameworks per giungere agli obbiettivo appena descritti, dei quali se ne riportano i più importanti.

Game Kit Framework - fornisce varie funzionalità separate, principalmente implementabili nelle applicazioni di tipo ludico; una di queste è la possibilità di attivare una connettività con altri dispositivi basati

su iOS di tipo peer–to–peer tramite una connessione diretta (Bluetooth o wireless), con la quale scambiarsi qualsiasi tipo di dato. Inoltre è possibile usufruire di una comunicazione vocale, sempre attraverso l'attivazione di una rete personale e diretta fra i due interlocutori che utilizzano prodotti equipaggati con iOS. Per garantire la massima integrazione con le applicazioni ludiche, tale framework mette a disposizione uno strumento chiamato *Game Center*, che presenta innumerevoli funzioni tra cui la possibilità di essere in contatto con una lista di amici quando si gioca, mantenere online una classifica e risultati personali aggiornati, il supporto al multiplayer etc.

Map Kit Framework - fornisce un'interfaccia per integrare le mappe direttamente nella finestre e view delle applicazioni, supportando anche l'inserimento di annotazioni, la possibilità di introdurre sovrapposizioni fra differenti tipologie di mappe e la ricerca tramite un'operazione di reverse–geolocation al fine di determinare le informazioni per un determinato segnaposto, una volta note le coordinate geografiche. È importante ricordare che si utilizza largamente il servizio di Google per poter visualizzare mappe dati.

iAd Framework - consente all'applicazione di ottenere un introito mediante la visualizzazione di annunci pubblicitari nella forma di piccoli banner. Gli annunci sono incorporati all'interno delle view in formato standard e possono essere visualizzate in ogni momento.

UIKit Framework - come già accennato nel livello Core Services, è uno dei due framework portanti nella struttura architetturale di iOS; in questo caso il suo compito principale è mettere a disposizione tutte le classi atte alla costruzione e gestione del'interfaccia utente nell'applicazione. Inoltre incorpora il supporto per caratteristiche specifiche di alcuni dispositivi, come l'accellerometro, la libreria fotografica dell'utente, informazioni sullo stato della batteria, sui sensori di prossimità etc.

1.3 Dietro le quinte dei framework: Cocoa

Cocoa costituisce un ambiente applicativo per entrambi i sistemi operativi targati Apple; esso consiste in una suite di librerie software Object-Oriented,

un sistema runtime e un ambiente di sviluppo integrato, Xcode, per entrambi le librerie; più precisamente Cocoa è l'insieme di APIs (Application Programming Interfaces) native ed Object-Oriented che forniscono un ambiente designato per il supporto delle applicazioni in esecuzione su Mac OS X e iOS, organizzate in frameworks, alcuni dei quali descritti o elencati nei capitoli precedenti.

Rappresenta l'ambiente applicativo preminente per Mac OS X, mentre risulta essere l'unico per iOS ed è costituito da due "facce", o più tecnicamente da due aspetti. Quello di runtime si interessa di presentare l'interfaccia utente e per questo strettamente integrato con gli altri componenti visibili del sistema operativo. L'aspetto di sviluppo permette di creare rapidamente robuste applicazioni con ogni sorta di possibili funzionalità, utilizzando una suite di componenti software, classi (sempre orientati agli oggetti) che costituiscono dei "blocchi" software indipendenti e riusabili.

Nello sviluppo di applicazioni Cocoa è possibile utilizzare diversi linguaggi di programmazione, ma fondamentalmente è richiesto il linguaggio *Objective-C*, che è un superset dell'*ANSI C*, esteso attraverso alcune funzionalità e caratteristiche a livello semantico e sintattico (le quali a loro volta derivano da *Smalltalk*) per supportare programmazione Object-Oriented.

Tra i tanti frameworks in Cocoa, tre sono considerati i più importanti, rispettivamente Foundation e AppKit per Mac OS X, Foundation e UIKit per iOS. L'abbinamento del framework Foundation con AppKit o UIKit rispecchia la ripartizione delle interfacce di programmazione Cocoa in classi legate alla GUI (AppKit e UIKit), da quelle che non lo sono, com'è nel caso di Foundation. I framework sopra elencati divengono essenziali nelle rispettive piattaforme in ogni progetto Cocoa che includa lo sviluppo di un'applicazione perché tipicamente vi è sempre un metodo o funzione appartenente ad uno di questi che permette di accedere alle potenzialità di determinate tecnologie situate a livelli inferiori rispetto a quello applicativo; ovviamente se si necessita di specifiche funzionalità, oppure di un controllo accurato su ciò che accade all'interno dell'applicazione, è sempre possibile riferirsi in modo diretto ad un framework sottostante.

A titolo di esempio si può considerare il caso in cui UIKit utilizzi alcuni metodi di un altro framework specifico, *WebKit*, per manipolare del testo destinato a comparire su pagine web con alcune tecniche principali, mentre per poter avere a disposizione tutti gli strumenti occorre riferirsi al *Core Text framework*, posizionato nel livello sottostante, per la gestio-

ne di generico testo in tutte le applicazioni che lo richiedono, del quale si integrano alcune funzionalità più utilizzate in frameworks come UIKit.

Anche se basati sullo stesso kernel centrale ed entrambi composti da Foundation, uno dei frameworks fondanti, vi sono piccole, ma significanti differenze circa la composizione e la funzione dei frameworks di cui sono composti Mac OS X e iOS. Alcune di esse sono pressoché lampanti, come per esempio il differente tipo di interfaccia grafica utilizzata per interagire con l'utente e tutto ciò che comporta l'organizzazione di questo aspetto; altre invece riguardano aspetti di comunicazione e di produttività. Ad ogni modo è importante evidenziare come, generalmente, le librerie e frameworks definiti in iOS siano un sottoinsieme, o subset, di quelle presenti in Mac OS X; in questo senso il sistema desktop è considerato sotto molti punti di vista un'estensione della piattaforma mobile, della quale se ne approfondirà ulteriormente la trattazione nel capitolo successivo.

Capitolo 2

Sviluppo di applicazioni Cocoa in iOS: principali meccanismi

Come riportato più volte all'interno di "Costruire Sistemi Software" [6],

> *costruire software significa realizzare un prodotto, sulla base di un progetto, adottando uno specifico processo di sviluppo.*

È importante adottare un processo che renda possibile, in primo luogo, la realizzazione di un prodotto di qualità per rimediare ad errori o introdurre modifiche con relativa facilità, il quale potrebbe essere riapplicato nella costruzione di altri sistemi, senza costringere ad una organizzazione dal punto di partenza.

Al fine di concludere la fase di progettazione ottenendo un prodotto di qualità, è opportuna non solo la conoscenza, ma bensì la padronanza di concetti strutturali e stilistici, comuni a tutte le applicazioni Cocoa indirizzate alla piattaforma mobile iOS, che comprendono:

- la gestione della memoria all'interno del sistema e delle applicazioni;
- la programmazione multithreading e come questa possa essere utilizzata per l'ottimizzazione delle risorse utilizzate;
- i principali metodi di comunicazioni fra i diversi tipi di oggetti interessati nella comunicazione con l'utente, sia in locale, sia tramite l'ausilio della rete, elemento divenuto oramai indispensabile;

- i design pattern e come essi siano stati adattati rispetto alle forme standard.

Inoltre è importante specificare ulteriormente che tale studio è rivolto ad applicazioni per dispositivi mobile che supportino il sistema operativo iOS, quindi con capacità computazionali, risorse hardware, ma soprattutto dimensioni ridotte, dove l'utente può interagire solo mediante lo schermo touchscreen.

In particolare ci si riferirà ad *iPhone*, lo smartphone di casa Apple, come modello target, per escludere alcuni aspetti legati alla presentazione dell'interfaccia utente, e per la rilevanza del dispositivo che ha contribuito a rivoluzionare il modo di concepire ed utilizzare gli smartphone; ad ogni modo i contenuti seguenti si considerano validi anche per i rimanenti dispositivi mobili disponibili sul mercato alla data di scrittura ed equipaggiati con iOS, differenziandosi per le dotazioni hardware e il fattore di forma che ne caratterizza il funzionamento.

Per quanto riguarda la versione del sistema operativo, si è scelto di riferirsi ad *iOS 5.0.1*, corrispondente all'ultimo aggiornamento della piattaforma mobile, rilasciato a Gennaio 2012, in cui sono state aggiunte funzionalità ad alcuni framework che contribuiscono a rendere i dispositivi odierni ancora più "smart"; ai fini della trattazione corrente, è sufficiente prendere in considerazione versioni non inferiori alla 4.0, per ovvi motivi legati all'introduzione del multitasking per le applicazioni di terze parti.

2.1 Dal Sistema Operativo alle Applicazioni

Ogni odierna applicazione progettata per la piattaforma iOS è realizzata utilizzando il framework UIKit e presenta essenzialmente la medesima architettura; infatti UIKit rende disponibile l'infrastruttura elementare, costituita da "oggetti–chiave" necessari al funzionamento dell'applicazione, al coordinamento per la gestione e l'invio degli input dell'utente, alla visualizzazione del contenuto sullo schermo, mentre ciò che rende un'applicazione differente da ogni altra è il risultato della configurazione di tali oggetti di default e dell'introduzione di ulteriori oggetti personalizzati o specificatamente creati, al fine di apportare modifiche al comportamento e all'interfaccia utente, in relazione agli obbiettivi da perseguire.

Nello specifico, l'infrastruttura di UIKit appena citata rende possibile la gestione automatica della maggior parte delle principali interazioni fra il sistema operativo e le applicazioni in esecuzione; inoltre gestisce i comportamenti applicativi che rientrano nella categoria "core" tramite un oggetto dedicato chiamato *UIApplication*, con il compito di gestire il ciclo continuo di eventi riferiti alla specifica applicazione e coordinando i restanti comportamenti di alto livello; la responsabilità per le risposte a tali eventi ed azione relative ricade completamente sulla bontà del codice personalizzato redatto su misura.

Il concetto molto importante alla base di tutto ciò è che relazioni similari sussistono anche in molte altre parti dell'applicazione, dove gli oggetti di sistema gestiscono i processi nel loro insieme, mentre il codice personalizzato si focalizza sull'implementazione del comportamento specifico desiderato. La figura sottostante riassume in breve quanto detto riguardo al ruolo che ciascun oggetto ricopre. Come si può notare le applicazioni iOS sono

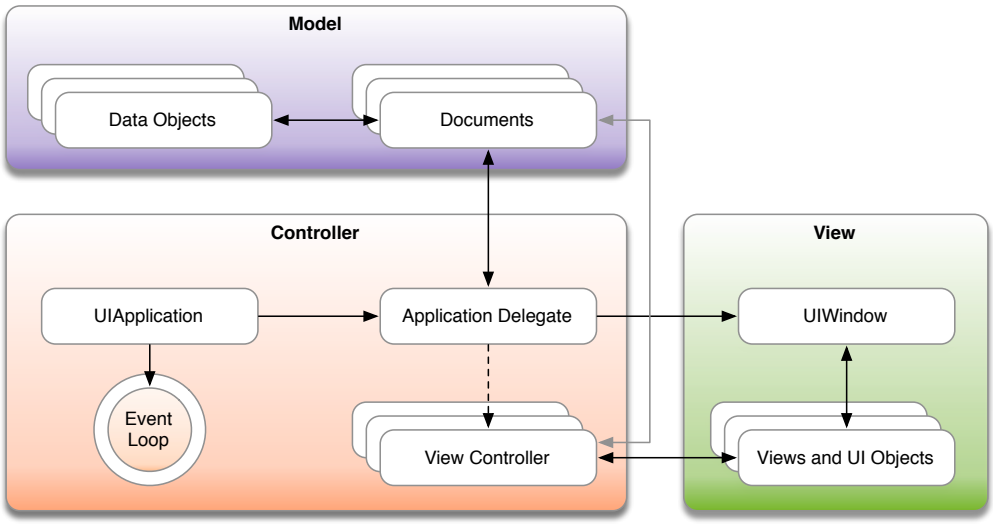

Figura 2.1: Oggetti fondamentali all'interno delle applicazioni

organizzate secondo il *model–view–controller* design pattern, consentendo sostanzialmente una separazione degli oggetti a seconda del ruolo ricoperto e promuovendo la riusabilità del codice, necessaria per la creazione di applicazioni universali, di cui sarà ampiamente trattano nei capitoli seguenti; un altro elemento importante è costituito dalla presenza di un oggetto

delegato, *Application Delegate*, affiancato ad *UIApplication*, il quale rappresenta il concetto stesso di delegazione (uno dei più importanti design pattern introdotti e personalizzato da Apple), e che gestisce le transizioni di stato all'interno dell'applicazione, ma che in generale costituisce il punto di riferimento per la manipolazione e smistamento di tutti gli specifici eventi. Quest'ultimo elemento costituisce la principale classe di oggetti che interagiscono con le varie rappresentazioni delle informazioni ed anche con i *View Controller*, oggetti che gestiscono la presentazione del contenuto della specifica applicazione sullo schermo tramite l'inserimento di viste (*UIView*), agganciate a runtime ogni qual volta sia necessario.

2.2 Gestione della Memoria

La conoscenza dei passi fondamentali nel processo di gestione della memoria "lato utente" è un aspetto rilevante, per poter avere una maggior consapevolezza sull'impatto che può produrre una buona progettazione dei componenti, senza però tenere in considerazione il carico delle risorse non distribuito in modo ottimale rispetto al metodo di gestione degli oggetti in memoria. In particolare questa sezione risulta necessaria in tutta la fase di sviluppo di applicazioni per la piattaforma iOS, come verrà chiarito in seguito.

Si definisce *Gestione della Memoria* a livello applicativo, il processo di allocazione di memoria durante l'esecuzione di un programma, utilizzandola e rilasciandola quando non è più necessaria. Tipicamente è buona norma che un programma utilizzi un quantitativo di memoria che corrisponda al minimo necessario, come naturale risultato di una gestione efficiente delle risorse; questo è tanto più vero quanto più le risorse a disposizione sono limitate, come è nel caso corrente, infatti l'ambiente a runtime di cui è costituito iOS prevede e definisce uno specifico meccanismo di gestione delle risorse in memoria, al quale sono affiancate una serie di politiche al riguardo, chiamate *Politiche di Proprietà esercitata sugli oggetti*, (*Ownership Policy*), applicabili in tre modalità ben precise, alle quali corrispondono altrettanti livelli di consapevolezza o anche responsabilità:

- *MRC (Manual Reference Counting)*
- *ARC (Automatic Reference Counting)*

- *Garbage Collector*

l'ultimo dei quali non verra preso in considerazione perché non supportato in iOS, a causa di una quantità di risorse richieste troppo elevata; questa metodologia prevede infatti l'introduzione di un'ulteriore componente software, e quindi dell'aumento del carico computazionale, ritenuto non particolarmente adatto per l'ambiente mobile, considerando l'idea dell'azienda stessa di realizzare un sistema operativo il più possibile fluido e performante, adatto ad un utilizzo rapido, frammentario.

2.2.1 Politiche di gestione per gli oggetti in memoria

L'organizzazione dei framework per la gestione della memoria prevede che Foundation fornisca al livello sovrastante un set di istruzioni compatto in Objective-C, mentre nel framework Core Foundation è associata l'implementazione estesa delle funzioni sovrastanti, ma descritte in linguaggio C, più adatto a comunicare con l'hardware sottostante.

Il set di istruzioni fornito dal framework Foundation costituisce lo strumento mediante il quale applicare il modello definito come "Politiche di Proprietà esercitata sugli oggetti" (d'ora in avanti si farà riferimento ad esso con il nome *Object Ownership Policy*), considerando ora la gestione di memoria non al livello di oggetti individuali, ma come un metodo per la distribuzione della proprietà, esercitata su un piccolo insieme di risorse in memoria. In particolare la consistenza di tale modello si basa su un prerequisito fondamentale, ovvero la possibilità per ogni oggetto di avere uno o più proprietari; fintanto che l'oggetto mantiene almeno un proprietario continua ad esistere, altrimenti il sistema di runtime si preoccupare di rimuoverlo dalla sua posizione in memoria.

L'insieme di politiche fissate per mantenere la validità del concetto di proprietà appena espresso sono implementate attraverso la nozione di *Reference Counting* (*conteggio delle istanze*, ovvero i riferimenti agli oggetti), un meccanismo o procedura che associa ad ogni oggetto Cocoa un intero rappresentante il numero di "proprietari" (in questo caso altri oggetti o frammenti di codice procedurale) interessati alla sua persistenza in memoria. Questo intero è riferito al conteggio delle occorrenze nelle quali si è inserito un riferimento a tale oggetto. Il Reference Counting è attuato tramite l'utilizzo di un set ristretto di istruzioni (messaggi) inviati agli oggetti

da manipolare, con una politica sovrastante che fissa alcune semplici regole per ottenere un risultato consistente, riassunte di seguito.

- Si detiene la proprietà su un oggetto che si è creato mediante le istruzioni che iniziano con *alloc*, *new*, *copy* o *mutableCopy*;
- In un qualsiasi momento è possibile richiedere la proprietà su un oggetto chiamando il metodo *retain*;
- È necessario rilasciare la proprietà che si detiene sull'oggetto inviando ad esso un messaggio di *release* o *autorelease*, dal momento in cui si sono concluse le operazioni con tale oggetto e non deve essere più utilizzare per i propri scopi;
- Non si deve assolutamente rilasciare la proprietà riferita ad oggetti che non si detengono.

Un esempio di tali meccanismo è dato dal diagramma sottostante.

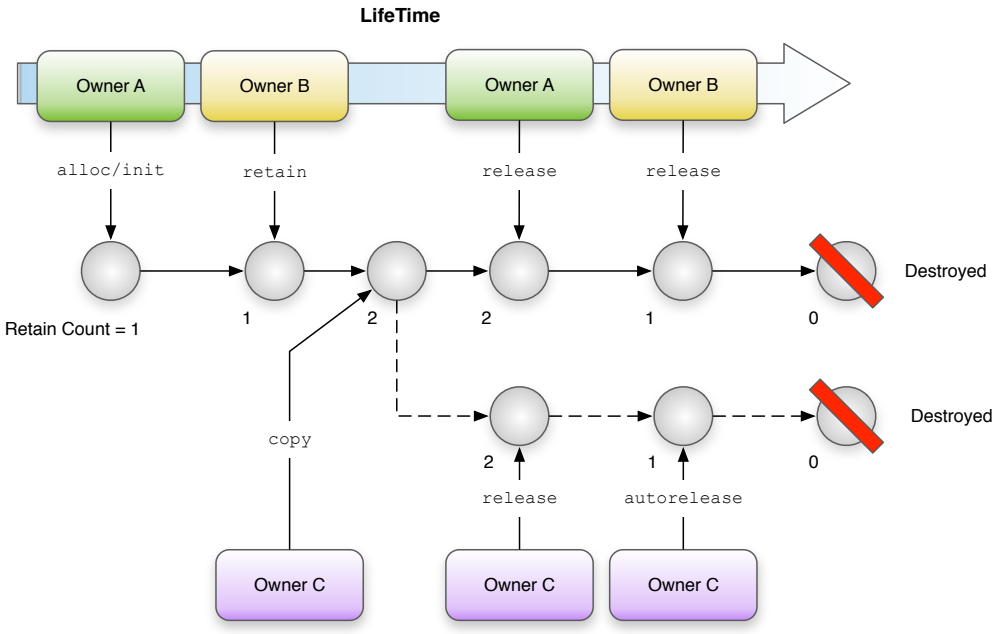

Figura 2.2: Il Retain Count in azione

2.2.2 MRR

L'approccio denominato *Manual Reference Counting* (*MRC*), conosciuto anche con il nome di *Manual Retain-Release* (*MRR*), consente di gestire in maniera esplicita la memoria del livello applicativo, tenendo traccia di ciascun oggetto che si detiene. In questo caso è lo stesso sviluppatore a preoccuparsi di quali messaggi inviare per soddisfare correttamente la Object Ownership Policy, sempre utilizzando il modello di Reference Counting definito in precedenza.

Per ottimizzare l'utilizzo delle risorse o in casi in cui la performance applicativa risulti di primaria importanza è consigliato, se non necessario, prendere in considerazione l'utilizzo del metodo MRC per la gestione della memoria, perché è l'unico che consenta un'ampia gamma di opzioni e possibilità per ottenere una completa gestione delle risorse. Ovviamente, le grandi potenzialità offerte si traducono in un rischio e possibilità maggiore di compromettere il funzionamento dell'intera applicazione; per questo motivo spesso si preferisce affidarsi al metodo duale ARC.

2.2.3 ARC

ARC è l'acronimo di *Automatic Reference Counting*; utilizza la stessa procedura in MRC per implementare la Object Ownership Policy anche se, rispetto al metodo manuale accoppiato, inserisce automaticamente l'appropriato metodo di gestione della memoria a tempo di compilazione, rimuovendo la relativa implementazione dal codice sorgente.

Sostanzialmente ARC non introduce altre funzionalità rispetto ad MRC, ma aggiunge solamente una certa quantità di codice a tempo di compilazione per assicurare la persistenza degli oggetti in memoria solo per il tempo necessario, non oltre. Per questo motivo l'esecuzione di applicazioni dove viene utilizzato il metodo ARC sono pressoché identiche a quelle dove si utilizza correttamente il metodo MRC e le differenze nel comportamento dell'intera applicazione sono spesso trascurabili dato che l'ordinamento delle operazioni e la performance in entrambi i casi è molto simile.

Anche se in molti casi l'utilizzo del metodo ARC semplifica l'approccio al lato gestionale delle risorse in memoria, rimane sempre uno strumento che agisce in modo automatizzato, il quale presenta alcune problematiche in scenari particolarmente complessi.

2.3 Programmazione Concorrente

Una delle maggiori sfide per gli odierni sviluppatori è quella di scrivere software che possa eseguire azioni più o meno complesse in risposta all'input dell'utente, rimanendo comunque reattivo ad ogni altra richiesta esterna, in modo da non creare attese mentre il processore è impegnato nello svolgimento di un determinato *task*, termine utilizzato per fare riferimento al concetto astratto di lavoro che deve essere eseguito.

Questa sfida accompagna il mondo dell'informatica da molto tempo; il problema continua ad esistere, anche se vi è stato un avanzamento delle tecnologie ed accompagnato sempre da un incremento della velocità nelle CPU, perché parte di tale problematica è da ricercare nel modo in cui è scritto tipicamente il software, ovvero come una sequenza di eventi da elaborare seguendo un certo ordine. Operando in questo modo il software è scalabile fino alla velocità di aumento della CPU, ma solo fino ad un certo punto; infatti non appena il programma viene bloccato in attesa di una risorsa esterna, l'intera sequenza di eventi è effettivamente mantenuta in attesa.

Tale argomento risulta essere molto significativo specialmente in ambiente mobile, dove una grande importanza riveste il fattore di reattività di un'applicazione, addirittura in primo piano rispetto al livello di *throughput*, il quale si mantiene sempre in aumento, dato l'enorme sviluppo dei componenti hardware che costituiscono gli smartphones.

Già da diversi anni si è adottato un approccio che costituisce tutt'ora una valida soluzione alla problematica sopra riportata, basato sulla programmazione concorrente.

Innanzi tutto, con il termine *concorrenza*, si intende l'esecuzione di un insieme di processi computazionali nello stesso istante, riuscendo in parte ad ottimizzare le performance applicative; infatti, se per molto tempo le prestazione degli elaboratori furono largamente limitate dalla velocità alla quale potevano giungere i singoli microprocessori, con l'avvicinarsi sempre più al loro limite fisico, i produttori di chip adottarono soluzioni multicore, dando la possibilità agli elaboratori di eseguire un insieme di tasks simultaneamente ed incrementando la potenza totale dei chip.

Mac OS X e iOS, come gran parte dei sistemi operativi moderni, si sono avvalsi da subito di questi core aggiuntivi, quando possibile, per eseguire task relativi a livello di sistema operativo, mentre all'interno delle applicazioni si è sfruttata tale potenzialità offerta introducendo la nozione di *Thread*, in

una programmazione definita *multithreading*.

I dispositivi mobili, che fino ad un anno fa adottavano processori con un singolo core, utilizzano principalmente soluzioni multithreaded per suddividere la parte interessata alla presentazione dei contenuti da quelle concentrate nell'esecuzione di operazioni specifiche; in questi casi i threads utilizzati vengono schedulati, forzando un avvio ed interruzione dell'esecuzione quando necessario, in modo *prelazionale* (*preemptive*). Così facendo, nelle fasi di sospensione di alcuni thread, altri possono entrare in esecuzione, dando vita ad una sorta di switch fra essi, per simulare situazioni di esecuzione simultanea, che risulta ovviamente improponibile per dotazioni hardware che contano di un solo core.

Un primo approccio, seguito fin dagli albori della programmazione concorrente, è definito utilizzando "direttamente" la nozione di Thread, ma ciò ha introdotto, oltre che un certo grado di utilità e beneficio, anche potenziali problematiche e costi, in termini di utilizzo delle risorse, come verrà chiarito successivamente; per tali ragioni Apple presenta alcune alternative, elencate di seguito.

Notifiche a tempo inattivo (*idle-time notifications*) - utili per tasks con una durata relativamente limitata e soprattutto caratterizzata da un livello minimo di priorità. Permettono di eseguire tasks nei brevi istanti di tempo in cui l'applicazione non è occupata nello svolgimento delle sue normali mansioni. Le APIs messe a disposizione utilizzano demoni e processi per portare a termine i compiti assegnati.

Timer - utilizzati in special modo all'interno del Thread principale di ogni applicazione, per eseguire tasks periodici che richiedono determinate risorse ad intervalli regolari, ma tali da non esigere l'utilizzo dei Threads.

Operazioni - costituiscono oggetti *wrapper* per il concetto di task. Sono state introdotte nella prima versione pubblica dell'SDK di iOS e in Mac OS X 10.2.

Grand Central Dispatch (GCD) - facendo il suo debutto in Mac OS X 10.6 e nella versione 4.0 di iOS, è considerata, ad oggi, la soluzione più rivoluzionaria in questo campo. Utilizza strutture di controllo molto simili a quelle presenti nelle Operazioni e si incarica dell'intera gestione dei Threads.

Al fine di un'analisi critica, si considereranno solamente le due soluzioni più utilizzate e consigliate nelle linea guida ufficiali, dove si specifica, inoltre, che pur continuando a supportare l'idea di gestione diretta dei Threads, tale pratica è apertamente sconsigliata, in particolare per applicazioni di media–alta complessità e per dispositivi mobili con ridotte capacità computazionali o risorse disponibili.

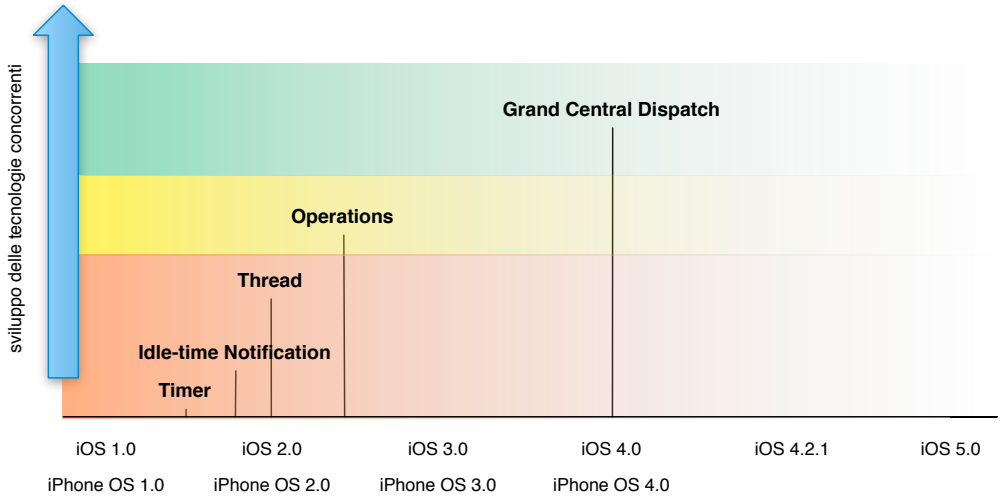

Figura 2.3: Tecnologie utilizzate in iOS nel corso degli anni

2.3.1 Threading in iOS

Da un punto di vista prettamente tecnico si definisce *Thread* una combinazione di strutture dati, sia al livello di kernel che a quello applicativo, necessarie per gestire l'esecuzione di codice all'interno di un processo. A livello di kernel tali strutture coordinano l'invio e la ricezione di eventi relativi al Thread specifico ed effettuano lo *scheduling prelazionale* o *preemptive* di quest'ultimo in uno dei cores disponibili, mentre a livello applicativo le strutture utilizzate sono costitute dallo stack, dove sono mantenute in memoria le chiamate a funzioni, e alcune variabili necessarie per gestire e manipolare gli attributi e lo stato del Thread.

Utilizzando un'intero set di framework orientati agli oggetti, nelle piattaforme Cocoa è supportata la modellazione del concetto di Thread come og-

getto *NSThread*, incluso nel framework Foundation, anche se, data la natura Unix dei sistemi Apple, è presente la tipica *POSIX API*, adatta per i *POSIX Threads*, la quale definisce un'interfaccia basata sul linguaggio C, relativamente semplice e flessibile, per l'utilizzo e configurazione dei propri Thread, molto conveniente quando si tratta di scrivere software multi–piattaforma.

In applicazioni non concorrenti, vi è un solo Thread in esecuzione, il quale inizia e termina insieme alla routine principale (caratterizzata dalla funzione *main*); l'esecuzione dei metodi o funzioni che caratterizzano il comportamento dell'intera applicazione vengono considerati uno ad uno. In maniera molto simile, nelle applicazioni dove si sceglie un approccio concorrente, vi è sempre la presenza di un Thread principale associato al ciclo di vita dell'applicazione, ma è possibile aggiungere innumerevoli flussi separati di esecuzione, ognuno associato ad un task specifico. Questo modo di suddividere il carico computazionale porta a due importanti potenziali benefici:

- miglioramento della reattività percepita;
- valorizzazione delle performance per applicazioni real–time, ancora più accentuata se fanno uso di soluzioni hardware con architettura multicore.

Per quanto riguarda l'aspetto di comunicazione fra i Threads, sono previsti alcuni meccanismi specifici supportati dalle applicazioni Cocoa, la quale descrizione non verrà affrontata perché in secondo piano rispetto all'attuale soluzione adottata in entrambi le piattaforme, desktop e mobile.

Un fattore che riveste un ruolo principale nell'effettivo sfruttamento della concorrenza nelle applicazioni è la sincronizzazione dei Threads utilizzati, dovuta ad una caratteristica fondamentale insita nel concetto stesso del modello di Thrread, ovvero la condivisione delle risorse all'interno del medesimo processo. Per far fronte all'esigenza di un accesso sincronizzato, si ripropongono in questo caso concetti classici nel campo della concorrenza, in parte rivisitati, la quale trattazione esula dallo scopo dell'analisi intrapresa.

Correlato al fattore di sincronizzazione è il concetto di codice definito *Thread-Safe*, caratteristica di alcune librerie software di essere state organizzate considerando il tema della programmazione concorrente, prendendo misure idonee per garantire la protezione delle sezioni critiche, come è nel caso del framework Foundation, il quale è considerato generalmente Thread-Safe.

Focalizzando l'attenzione sulla piattaforma mobile iOS, il framework dedicato alla gestione dell'interfaccia utente, UIKit, è per la maggior parte di esso considerato *non Thread-Safe*; questo implica che dal momento in cui l'applicazione entra in esecuzione, tutte le chiamate a metodi che interagiscano con oggetti appartenenti ad UIKit debbano necessariamente essere eseguiti all'interno dello stesso thread, che comunemente è conosciuto come *main thread*, a meno che non si ritenga opportuno adottare manualmente misure di sincronizzazione, fortemente sconsigliate in questo caso. Tale necessità rappresenta una scelta progettuale ben precisa di utilizzare il Thread principale per la ricezione di eventi relativi ad input dell'utente ed avviare gli aggiornamenti necessari riguardanti l'interfaccia grafica. In particolare da esso dipende anche l'oggetto *UIApplication*, già menzionato nella parte introduttiva del capitolo secondo, quindi la maggior parte delle interazioni fra il sistema e l'applicazione.

Se si considera solamente l'esecuzione dei Thread accostata al tema della gestioni di eventi, appare evidente la presenza di una problematica insita nelle caratteristiche stesse di Thread: esso non si presta alla modellazione del concetto di ciclicità. Una volta che è terminata l'esecuzione viene rimosso dal sistema; quindi, dato che la creazione e gestione di un Thread è relativamente costosa in termini di risorse in memoria e di tempo, è consigliato attribuirgli un certo quantitativo di lavoro da eseguire. Tutto ciò risulta pesantemente in contrasto con il modello di programmazione *event–driven*, dove sono richiesti lunghi periodi di attività, interrotti da altrettanti istanti di pausa, o meglio di sospensione.

Proprio per cercare di far collidere i due modelli e giungere ad un terzo, *multithreaded event-driven* efficace, Apple ha introdotto il concetto di *Run Loops*.

2.3.1.1 Run Loops : Multithreading event-driven

Un Run Loop è una parte fondamentale dell'infrastruttura associata al Thread, utilizzata per la gestione di eventi asincroni che giungono ad esso. Rappresenta un gestore ciclico di eventi utilizzato per schedulare i task e coordinare la ricezione degli eventi in arrivo per un singolo Thread. Non appena è notificata la presenza di eventi, il sistema si incarica di farli giungere al Run Loop competente, il quale interrompe lo stato di sospensione del Thread e si preoccupare di inviare tali eventi all'*handler* che si è speci-

ficato; nel caso in cui non siano presenti eventi pronti per essere eseguiti, il Run Loop sospende temporaneamente il Thread.

Ogni Run Loop può ricevere eventi da due differenti tipi di sorgenti di informazione; le *Sorgenti di Input* (*Input Sources*) generano eventi asincroni, tipicamente sotto forma di messaggi provenienti da un altro Thread o un'altra applicazione, mentre le *Sorgenti Temporali*(*Timer Sources*) generano eventi sincroni, che si verificano ad ogni tempo di scheduling, oppure ripetuti con un certa periodicità.

Lo scopo principale di tale meccanismo risiede nella creazione di Thread caratterizzati da un tempo di esecuzione maggiore della media, quindi adatto ad una programmazione event-driver; inoltre si utilizza un minimo quantitativo di risorse disponibili, dato che il Thread viene posto in stato di sospensione e riattivato solo quando effettivamente necessario, eliminando la necessità di ricorrere al meccanismo di polling, che porterebbe ad un inevitabile aumento dei cicli di CPU.

Resa evidente la necessità di un efficiente modello di gestione degli eventi, la soluzione appena descritta consente di ottenere il minimo intervallo di tempo di *turnaround* per un dato evento ed anche di impostare un determinato livello di priorità ad un insieme ordinato di essi; inoltre, negli istanti temporali che intercorrono fra due eventi successivi il sistema può adottare misure di risparmio energetico, andando a sgravare sulla durata della batteria di cui sono provvisti tutti i dispositivi mobile.

Anche se ad ogni oggetto NSThread viene automaticamente associato un Run Loop, l'utilizzo di tale infrastruttura aggiuntiva non è indispensabile; ad ogni modo occorre impostare correttamente tutti i parametri per poter usufruire di tale gestione ottimizzata, come è elencato di seguito.

- Attivare il thread mettendolo in esecuzione;
- ottenere da quest'ultimo il riferimento al suo Run Loop
- installare gli handlers necessari per la gestione di altrettanti eventi;
- mandare in esecuzione il Run Loop stesso.

A questo proposito l'ambiente Cocoa prevede la configurazione automatica del Run Loop per il Main Thread, mentre per tutti quelli secondari occorre effettuare i passi sopra indicati, in una procedura che presenta un grado di complessità mediamente alto.

Un ulteriore aspetto negativo si riscontra nella gestione dell'eventuale coda di eventi all'interno del Run Loop; se viene percepito un determinato input da una sorgente registrata mentre lo stesso Run Loop è impegnato nell'esecuzione di un altra porzione di codice, prima di considerare gli eventi in coda termina l'esecuzione corrente; sostanzialmente non adotta uno scheduling prelazionale (preemtive).

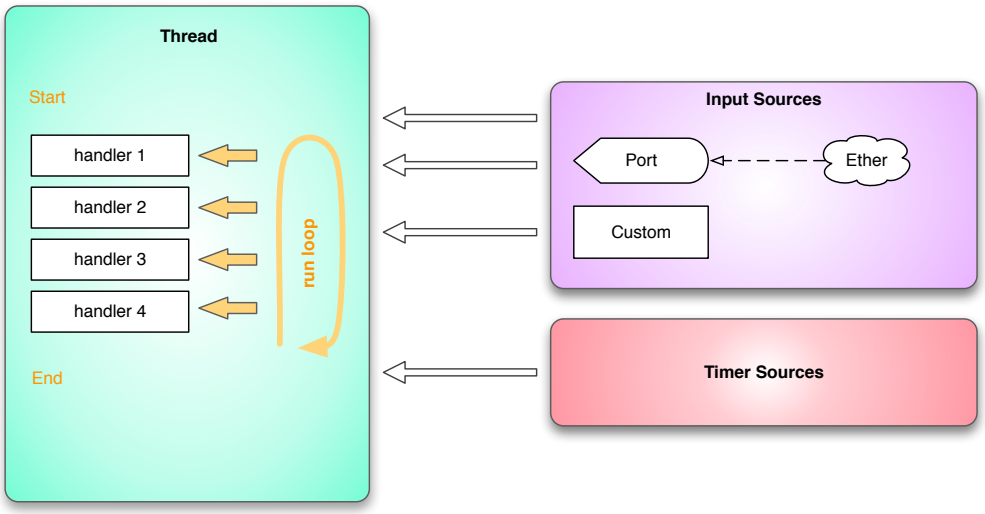

Figura 2.4: Funzionamento di un Run Loop

2.3.1.2 Aspetti negativi nella gestione diretta dei Threads

Come è ben noto, il concetto di thread rappresenta un approccio, sotto certi aspetti, relativamente agile (basti pensare alla caratteristica condivisione di risorse all'interno dello stesso processo) utilizzato per mettere in atto una programmazione di tipo concorrente, volta quindi allo sfruttamento delle architetture hardware multi–core.

É da tenere in considerazione, però, che giungere ad una gestione corretta dei threads, utilizzandoli direttamente, significa operare direttamente con strumenti di basso livello. Tradizionalmente questa era una delle poche soluzioni brillanti, ma che non risolve tutt'ora il problema della scalabilità non ottimale dei threads rispetto ad un numero arbitrario di core, accentuatosi con l'aumentare dei core presenti nelle architetture hardware. In

particolare, dato che il numero ottimale di threads per una determinata applicazione può variare a seconda del sistema correntemente in esecuzione e in relazione alla struttura architetturale dell'hardware sottostante, implementare una soluzione ottimale, basata sull'utilizzo di thread in modo diretto diviene ardua, se non impossibile da raggiungere, perché ciò implica la conoscenza del numero di cores che è possibile sfruttare efficientemente, fattore che presenta un alto grado di variabilità.

Anche se si riuscisse ad ottenere costantemente il numero ottimale di thread, molto spesso rimangono insoluti i compiti di programmare un numero anche molti alto di tali elementi, riuscire a mandarli in esecuzione con un alto grado di efficenza e in fine non permette che si verifichino fenomeni di interferenza con altri thread. In aggiunta, i meccanismi di sincronizzazione tipicamente utilizzati introducono ulteriore complessità e rischi per quanto concerne il design applicativo, senza troppe garanzie sull'incremento delle performance.

Volendo riassumere la problematica appena discussa, vi è la necessità di definire un metodo per sfruttare appieno il numero variabile dei cores all'interno dei processori, facendo sì che i flussi in esecuzione simultanea siano in grado di adattarsi dinamicamente per accogliere positivamente le condizioni di mutevolezza del sistema; inoltre la soluzione deve essere abbastanza semplice in modo tale da mantenere favorevole lo sfruttamento dei cores, quindi cercando di non aumentare la quantità di lavoro necessario per il loro utilizzo.

Invece che fare affidamento sull'utilizzo diretto dei thread e delle relative tecnologie messe a disposizione, Mac OS X e iOS adottano un diverso approccio progettuale di tipo asincrono per risolvere il problema di concorrenza; le soluzione che propone Apple si concentrano sull'idea di dividere tutti i task di lunga durata in unità di lavoro ed inserire queste in una struttura modellata come una coda, la quale permette di gestire in maniera automatizzata i vari flussi di esecuzione.

Le due tecnologie e meccanismi che si sono considerati costituiscono passi successivi per giungere ad una soluzione accettabile, che produca un notevole miglioramento per quanto riguarda la stabilità e performance della singola applicazione; in questo modo si attribuisce sempre più peso alla gestione avanzata del carico computazionale complessivo, svolta in quasi totale autonomia dal sistema.

2.3.2 Le Operazioni come prima soluzione alternativa al concetto di thread

Introducendo i concetti di *Operazione* e *Coda di Operazioni*, Apple ha definito un modo per incapsulare il codice e i dati associati ad un singolo task in un modello Object–Oriented che fornisce un'infrastruttura di supporto, minimizzando la quantità di lavoro che altrimenti ogni sviluppatore dovrebbe affrontare nel campo della programmazione multithreading. Tale soluzione, basata interamente su interfacce Objective-C, è comunemente utilizzata in applicazioni Cocoa.

Ogni oggetto Operazione è un'istanza della classe *NSOperation* (appartenente al framework Foundation), alla quale è attribuito il compito di eseguire il task associato una sola volta, senza cioè presentare un comportamento ciclico. In una Operazione è inclusa gran parte della logica necessaria per una coordinazione sicura dello specifico task; proprio grazie alla presenza di tale logica, è possibile focalizzare l'attenzione sull'implementazione del task da eseguire, senza preoccuparsi della produzione e gestione di ulteriore codice necessario per assicurare una buona fase di esecuzione.

La seconda parte della soluzione prevede l'introduzione del concetto di Coda di Operazioni, modellata dall'oggetto *NSOperationQueue*; essa regola l'esecuzione di un determinato insieme di oggetti NSOperation. Una volta inseriti nella coda, gli oggetti NSOperation vi rimangono fino a quando vengono esplicitamente cancellati o hanno concluso la propria porzione di lavoro. Inoltre per quelli ancora non mandati in esecuzione è prevista una precisa organizzazione all'interno della coda, in accordo con il livello di priorità che è possibile settare e alle dipendenze che intercorrono fra gli oggetti. Queste ultime costituiscono un modo conveniente per poter eseguire le Operazioni in un ordine specifico, aggiungendo e personalizzando la semantica attribuita a tale strumento di controllo; generalmente un oggetto NSOperation non è considerato attivo (ready) fintanto che tutti gli oggetti, con i quali vi è un rapporto di dipendenza, non hanno concluso la loro esecuzione, anche se il concetto di dipendenza esula dal controllo relativo al successo o insuccesso delle esecuzioni critiche.

Un altro aspetto che caratterizza positivamente tale approccio è la possibilità di monitorare i cambiamenti avvenuti a tempo di esecuzione degli oggetti Operazione, utilizzando un particolare tipo di meccanismo che consente di inviare notifiche all'applicazione, chiamato *KVO* (*Key–Value*

Observing), sul quale non ci si soffermerà oltre.

Per quanto concerne l'apparato tecnologico, i concetti di Operazione e Code di Operazioni fanno uso dei Thread per supportare tale meccanismo. Questi ultimi sono mandati in esecuzione direttamente dall'oggetto NSOperationQueue per ogni operazione contenuta; la logica descritta rimane valida fino alla versione 10.6 del sistema desktop e fino a iOS 4.0; successivamente vi è stato un importante sviluppo nel settore delle tecnologie adottate da Apple, inerenti al settore di programmazione concorrente, con l'introduzione di una nuova soluzione rivoluzionaria, conosciuta in ambiente Unix come *Grand Central Dispatch*. L'utilizzo dei thread nell'approccio

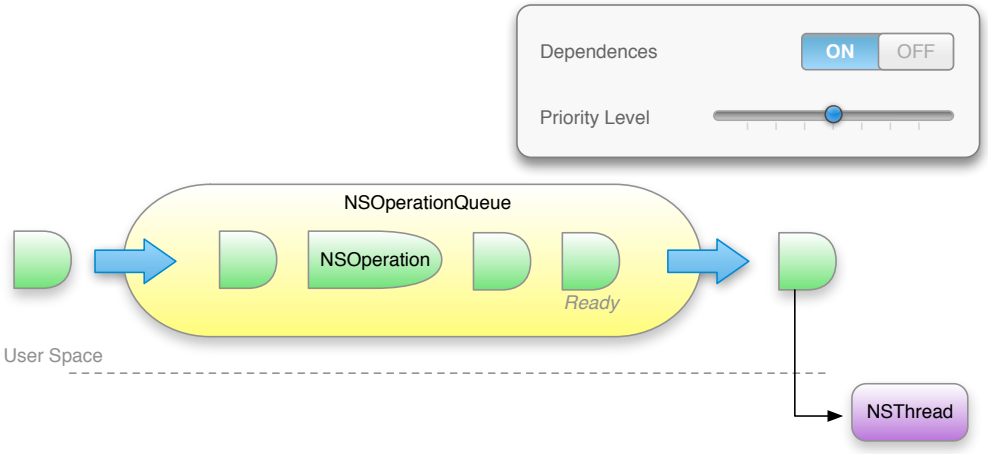

Figura 2.5: Operazioni e Code di Operazioni

corrente sottolinea come si sia effettuato solamente un piccolo passo in direzione di una gestione delle risorse, correlate a differenti flussi di esecuzione paralleli, più consapevole e scalabile rispetto al numero variabile di cores presenti; ad ogni modo l'utilità della soluzione associata al concetto di Operazioni è resa evidente dal fatto che esse siano state designate per affinare il concetto di concorrenza all'interno dell'applicazione.

Il livello di affidabilità e semplicità d'uso si è innalzato, organizzando ed incapsulando il comportamento dell'applicazione in moduli di piccola dimensione, ai quali fare riferimento per eseguire, in maniera asincrona, un qualunque task personalizzato, mappato su uno o più thread separati e risultando del tutto trasparente rispetto al livello applicativo.

2.3.3 GCD: la soluzione attuale

Grand Central DIspatch (GCD) è il nome della rivoluzionaria tecnologia sviluppata da Apple che costituisce un nuovo tipo di approccio pervasivo nell'ambito della programmazione concorrente, indirizzato ad una *programmazione multicore*. Non rappresenta una caratteristica del livello applicativo, ma bensì un potente strumento, distribuito su diversi livelli architetturali, per lo sviluppo e l'intera gestione dei concetti di task e operazione; in particolare è un insieme esteso di API di basso livello, scritte in linguaggio C, inserite dalla versione 10.5 di Mac OS X e a partire da iOS 4.0, a supporto di molti framework Cocoa Object-Oriented, al fine di rendere le applicazioni maggiormente orientate al pieno sfruttamento delle architetture multicore, nel modo più semplice e veloce possibile. Per poter usufruire direttamente di tale tecnologia è disponibile una particolare libreria (*libdispatc*), anch'essa in C, la quale rappresenta il fulcro implementativo del GCD nello spazio utente.

A differenza della soluzione precedentemente descritta, non considera la modellazione del concetto di thread come oggetto NSThread (valido per il linguaggio Objective-C), ma utilizza direttamente i POSIX Thread nei livelli sottostanti rispetto a quello applicativo, come verrà spiegato successivamente.

L'idea centrale è quella di traslare gran parte della responsabilità nell'esecuzione e gestione dei thread, intesi come flussi informativi associati al singolo utilizzo del programma, dal livello applicativo a quello del sistema operativo. Le versioni aggiornate della piattaforma desktop e mobile forniscono una serie di APIs tramite le quali molti dei framework più importanti, fra cui Foundation, mettono a disposizione un meccanismo che entra in gioco a tempo di esecuzione, efficiente e altamente scalabile, in rapporto al numero di core presenti e/o disponibili per processare tutto il lavoro all'interno di una applicazione.

Il GCD semplifica la progettazione e sviluppo di applicazioni, descrivendo con semplicità la dipendenza che spesso sussiste fra le vari parti di lavoro, che caratterizzano il comportamento dell'applicazione target. Le unità di lavoro sono descritte in termini di *blocchi* (*blocks*), associati a strutture essenziali per la loro gestione e per la successiva mappatura in *pthread*, chiamate *Dispatch Queue*, letteralmente *Code di Spedizione*. Come risultato, lo sviluppatore non si preoccupare della gestione diretta dei threads e il codice

risulta più ordinato e comprensibile. Il GCD può contare sulla presenza

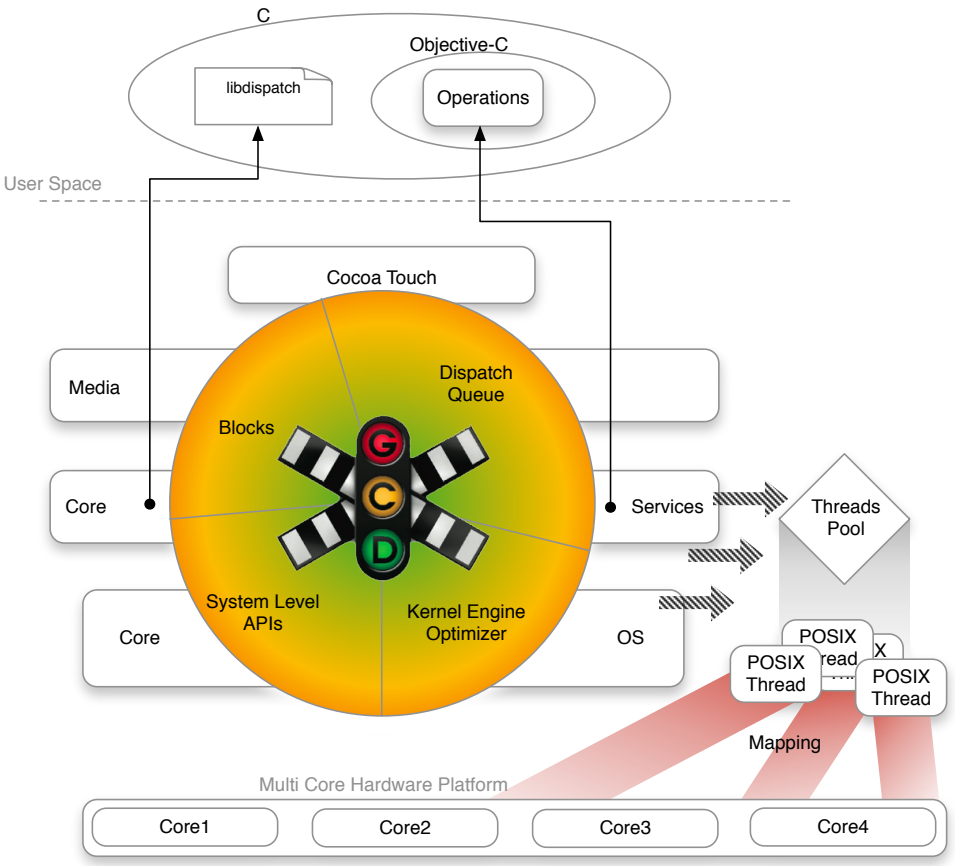

Figura 2.6: Grand Central Dispatch: visione d'insieme

di motore specificatamente creato per l'esecuzione multicore, utilizzato per avere una visione complessiva di tutte le Dispatch Queue utilizzate nelle applicazioni, in modo tale da assegnare conseguentemente determinati compiti da queste ultime all'insieme di thread sottostante, sempre gestiti dal GCD. In particolare la gestione dei threads avviene in base al numero di core disponibili e alle richieste pendenti che intercorrono fra le applicazioni ed il sistema, quindi distribuendo dinamicamente il carico computazionale.

Nelle applicazioni mobile iOS, l'adozione di tale approccio si traduce in un'esecuzione molto più fluida e in un parsimonioso utilizzo delle risorse,

due caratteristiche già menzionate nella parte introduttiva, le quali sono continuamente al centro dei principali dibattiti mirati al confronto fra due o più piattaforme mobile.

Il concetto probabilmente più importante è che senza un approccio altamente pervasivo, come è nel caso del GCD, anche la migliore applicazione non riuscirebbe a garantire la performance ottimale di un dato sistema, a causa della naturale posizione svantaggiosa occupata da essa nei confronti dell'intero sistema; infatti la singola applicazione non ha una totale consapevolezza su ciò che accade nell'ambiente circostante, mentre GCD ha la consapevolezza dello stato in cui si trova il sistema totale in un dato istante, riuscendo automaticamente a mappare le unità di lavoro previste dallo sviluppatore in threads individuali, ed a indirizzare questi ultimi nei core appropriati. Riassumendo, Grand Central Dispatch introduce un certo numero di benefici, quali:

- aumento della reattività;
- introduzione della Scabilità Dinamica;
- miglioramento nell'utilizzo dei processori;
- maggior pulizia e chiarezza del codice prodotto.

Si ricorda inoltre che tali obiettivi sono raggiunti attraverso una combinazione di:

- *Blocchi*, come estensioni del linguaggio C, C++ ed Objective-C;
- un efficiente motore a tempo di esecuzione che aumenta la scalabilità;
- un ricco insieme di API appartenenti ad un livello di sisema relativamente basso;
- un conveniente insieme di API Cocoa ad alto livello.

Grand Central Dispatch è stata sviluppata come progetto *Open source* e distribuita sotto *licenza Apache* a settembre 2009; è tutt'ora oggetto di studio anche grazie al grado di portabilità verso le altre piattaforme Unix, per l'utilizzo del linguaggio di programmazione C, anche se in questo caso il supporto a livello dello stesso kernel consente un'ottimizzazione delle performance ed è richiesta nel compilatore l'integrazione del supporto al concetto di blocco, necessario per implementare l'intero pacchetti di API del GCD.

2.3.3.1 I Blocchi come unità elementare di lavoro

In questo breve capitolo sarà data una descrizione per sommi capi dei blocchi al fine di motivare la loro introduzione e meglio comprendere il funzionamento all'interno del sistema.

Insieme al GCD, Apple ha aggiunto qualche nuova sintassi al linguaggio di programmazione C per implementare l'idea di *blocco*, molto importante per il funzionamento della soluzione adottata. Dietro al concetto di blocco vi è l'idea di trattare il singolo pezzetto di codice similmente agli altri tipi di dato in C; esso rappresenta tipicamente piccoli pezzetti di codice autonomo, e per questo è particolarmente utile come metodo per incapsulare unità di lavoro che potrebbero essere eseguite concorrentemente, oppure come elementi in una struttura o anche come funzioni callback richiamate al termine di una operazione.

Un blocco è molto simile al concetto di puntatore a funzione in C, anche se con qualche differenza sostanziale; infatti oltre al codice eseguibile, può contenere un collegamento automatico con uno stack o ad una porzione di memoria nello heap. Inoltre è in grado di mantenere un set di dati utilizzati per condizionare il comportamento quando è in esecuzione. Per quanto riguarda il loro utilizzo come funzioni di callback, ciò risulta molto utile in quanto:

- permettono la scrittura di codice direttamente al punto di invocazione, anche se eseguito successivamente, nel contesto di implementazione del metodo;

- possono accedere a variabili locali, mentre le normali funzioni devono necessariamente appoggiarsi ad una struttura dedicata.

2.3.3.2 Dispatch Queue: parte dell'"intelligenza" del GCD

Le Code di Invio (*Dispatch Queue*) sono un meccanismo basato sul linguaggio di programmazione C per l'esecuzione di task personalizzati. Ognuna di esse è in grado di eseguire determinati task sia in modo seriale che concorrente, ma sempre con una logica *FIFO (first–in–first–out)*. In più ogni tasks inviato ad una di queste code deve essere necessariamente incapsulato all'interno di una funzione o (ancora meglio) un blocco. Tali strutture dati, appartenenti allo user–space, costituiscono una elemento di grande impor-

tanza sia per aspetti legati alla serializzazione che per la concorrenza; infatti sussistono differenti tipi di code, in relazione all'obbiettivo da perseguire:

- Code Globali (Global Queue);
- Code Private (Private Queue);
- Coda Primaria (Main Queue).

A causa dell'importante ruolo che rivestono, devono essere estremamente efficienti e thread–safe, per poter accedere ai threads in modo veloce e sicuro. Per riuscire a raggiungere tale scopo, i blocchi devono essere aggiunti e rimossi tramite operazioni atomiche; fortunatamente queste esigenze sono automaticamente supportate da tutti i moderni processori Intel, i quali garantiscono un'esecuzione senza interruzione, anche in presenza di core multipli.

Il funzionamento generale delle code prevede che non appena un blocco sia pronto per essere eseguito, venga rimosso dalla coda e mappato in uno dei thread disponibili, contenuti nel *pool* di threads gestiti dal GCD. In questo modo si risparmia il costo dovuto alla creazione di un nuovo thread per ogni richiesta; inoltre si riduce drasticamente la latenza associata al processare il singolo blocco. Per ridurre il numero di threads inattivi o concorrenti, la dimensione del pool è automaticamente determinata dal sistema.

Tale modello di gestione dei thread nello spazio utente si può considerare simile al *cooperative multitasking*, ma non identico; infatti quest'ultimo presenta lo svantaggio di portare in alcuni casi ad una pesante riduzione della reattività dell'intero sistema, mentre il GCD evita questa spiacevole situazione, producendo altri thread a livello di kernel se i thread "cooperativi" non completano i blocchi abbastanza velocemente.

Analizzando più in profondità si nota che il funzionamento generale delle code si basa sulla famiglia di chiamate `pthread_workqueue` per gestire i threads; queste permettono al kernel di determinare il numero ottimale di thread da creare; il comportamento di default prevede la creazione di un thread per ogni CPU e per ogni livello di priorità, ma non si esaurisce qui la logica introdotta con il GCD; ne è una prova l'esempio proposto:

Si supponga di voler creare alcune code con due differenti livelli di priorità su un processore quad–core, avendo a disposizione otto threads che possono essere eseguiti in parallelo; a causa del livello di consapevolezza di cui è provvisto il kernel, se sono già in esecuzione diversi threads occupati nello

svolgimento di determinate mansioni, verrà generato un numero variabile di thread, generalmente in crescita in relazione all'aumentare della priorità della coda associata.

Global Queue Sono code per la gestione concorrente dei threads, disponibili per ogni processo. Ognuna di esse è associata ad un pool di thread che operano tutti con una data priorità. L'aspetto importante è che tali code sono in grado di monitorare le richieste di risorse che si verificano nell'intero sistema operativo, non solamente in un singolo processo. Tale veduta ad ampio raggio è la caratteristica che consente al GCD di bilanciare la domanda e richiesta di thread, in maniera del tutto automatica.

Private Queue In questo caso le code forniscono un accesso seriale e personalizzato, come avviene nel caso di strutture dati condivise. Dato che il comportamento tipico del GCD prevede una generale schedulazione di ogni cosa all'interno di una coda concorrente, tali code seriali sono in realtà schedulate utilizzando le Code Globali. In particolare, per ogni coda seriale ne viene attribuita una concorrente, per la quale viene impostato il livello di priorità predefinito; quando il primo blocco è inserito all'interno della prima (Private Queue), essa è automaticamente aggiunta alla seconda (Global Queue) e dato che le operazioni in questione sono atomiche, i blocchi addizionali sono aggiunti alla coda seriale.

Quando si giunge al momento di utilizzare ed eseguire i task, la stessa coda seriale viene rimossa da quella concorrente ed eseguita con lo stesso criterio e meccanismo con cui vengono trattati ipotetici blocchi aggiunti direttamente alla coda di destinazione. Il processo in cui ogni blocco viene eseguito in modo seriale è chiamato *draining*.

Main Queue Ogni processo ha un'unica e ben precisa coda principale, in particolare un coda seriale, la quale è associata al thread principale (main thread) del programma. Solo in questo caso alla coda è correlato un oggetto run-loop, accoppiato con il thread principale, i quali "drenano" i blocchi della coda seriale fino alla fine del ciclo di attività.

2.3.3.3 Sincronizzazione e gestione delle risorse condivise

In aggiunta alle Dispatch Queues, il Grand Central Dispatch fornisce diverse altre tecnologie che sfruttano il concetto di coda per fornire supporto alla gestione di fattori legati alla sincronizzazione. Anche in questo caso si tratta di strutture modellate come code, alle quali si può pensare come sottoclassi.

In generale, in questo tipo di code dedicate la sincronizzazione è molto più efficiente rispetto all'utilizzo dei normali *locks* perché in questi ultimi sono necessarie più interazioni che coinvolgono il kernel, mentre le code agiscono principalmente a livello di processo dell'applicazione, riferendosi allo spazio dedicato al kernel solo quando strettamente necessario.

Dispatch Groups Rappresentano un modo per monitorare un set di blocchi, bloccando un specifico thread fintanto che uno o più tasks non abbiamo terminato la propria esecuzione. Tipicamente sono utilizzati in situazione dove vi è un certo grado di correlazione fra alcuni thread, che comporti la sospensione fino a che tutti i task appartenenti non siano stati completati.

Dispatch Semaphores Tale elemento di sincronizzazione è molto simile al tradizionale semaforo, ma più efficiente; in particolare, quando una risorsa è disponibile, impiega un lasso di tempo inferiore rispetto al classico semaforo perchè il GCD non effettua alcuna chiamata all'ambiente del kernel in questo caso; l'unico caso in cui ciò accade è quando le risorse non sono disponibili e conseguentemente il sistema necessità di sospendere l'esecuzione del thread relativo fino ad un nuovo comando.

2.3.3.4 Il rapporto con la programmazione event–driven: le Dispatch Sources

Dopo un'attenta analisi effettuata, appare evidente che il GCD è una tecnologia event-driven; infatti le code mettono in esecuzione i blocchi in risposta ad eventi. Riguardo a questo importante aspetto, è presente un'ulteriore tecnologia creata a partire dalla struttura della coda, incaricata della generazione di notifiche in risposta a specifici eventi di sistema, chiamata *Dispatch Source*.

Una Dispatch Source rappresenta un tipo di struttura fondamentale che coordina l'elaborazione di specifici eventi di sistema, verificatisi a basso livello. É possibile monitorare una grande varietà di tipi di eventi, per ognuno

dei quali è stata introdotta una Sorgente specifica; in generale, quando si verifica un evento al quale si è registrati, il codice del task viene inviato alla Dispath Queue competente, in maniera sincrona o asincrona.

Questa soluzione sostituisce molto bene il concetto di run-loop; infatti, invece che creare e gestire manualmente uno di questi, si collega semplicemente una Dispatch Source ad una Dispatch Queue, contenente le unità di lavoro da eseguire, la quale offre inoltre molte più opzioni nel processare i dati generici, e non si preoccupa dell'interazione con i threads.

2.3.3.5 Non dimentichiamoci il Main Thread

Come già puntualizzato precedentemente, il framework UIKit non è thread–safe. La comunicazione fra thread in cui uno di essi appartenga al comparto delle interfacce utente è fortemente sconsigliata, se non negata. Inoltre, i task che concernono la GUI devono essere eseguiti all'interno del thread principale (*Main Thread*), per cui la Main Queue è la sola candidata per questo tipo di task nel GCD.

Al fine di ottimizzare le intense operazioni grafiche, le quali altrimenti andrebbero a gravare sulla reattività dell'applicazione nei confronti dell'input dell'utente, il GCD permette di portare a termine tutte le attività, considerate esose in termini di utilizzo di risorse e di tempo, in un thread separato, utilizzando una Global Queue, per poi letteralmente passare il risultato dell'elaborazione alla Main Queue, tramite l'utilizzo della funzione `dispatch_get_main_queue`, come è mostrato nello snippet di codice sottostante:

2.3.3.6 Operazioni vs GCD: le due soluzioni a confronto

Con l'introduzione del GCD si è mantenuta la compatibilità e il supporto al concetto di Operazione e Coda di Operazioni, anche se ne è stata re–implementata l'intera logica interna, utilizzando la soluzione rappresentata dal Grand Central Dispatch come infrastruttura fondante, che unifica in se tutti gli strumenti di alto livello utilizzati per affrontare il tema di programmazione concorrente, quindi escludendo la manipolazione e gestione diretta dei thread, sia come POSIX thread, sia come oggetti NSThread.

A partire da Mac OS X 10.5 e iOS 4.0 le Operation Queue utilizzano la libreria *libdispatch* per dare inizio all'esecuzione delle singole Operazioni; come risultato si ottiene un'esecuzione di Operazioni su un thread separato, in

```
...
- (IBAction)doWork:(id)sender {
    NSDate *startTime = [NSDate date];
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        NSString *fetchedData = [self fetchSomethingFromServer];
        NSString *processedData = [self processData:fetchedData];
        NSString *firstResult = [self calculateFirstResult:processedData];
        NSString *secondResult = [self calculateSecondResult:processedData];
        NSString *resultsSummary = [NSString stringWithFormat:
            @"First: [%@]\nSecond: [%@]", firstResult, secondResult];
        dispatch_async(dispatch_get_main_queue(), ^{
            resultsTextView.text = resultsSummary;
        });
        NSDate *endTime = [NSDate date]; NSLog(@"Completed in %f seconds");
    });
}
...
```

Figura 2.7: Come ritornare al Main Thread

particolare su un thread disponibile, sia nel caso di Operazioni concorrente che seriali.

Non è immediatamente ovvio cogliere le differenze che intercorrono fra il GCD e il concetto di Operazione (si intende dopo l'introduzione del GCD). GCD è sostanzialmente un'API di basso livello che offre all'utilizzatore un certo grado di flessibilità per strutturare il codice in una varietà di modi differenti. L'oggetto NSOperation fornisce una struttura ben definita e standard, in Objective-C, quindi un'astrazione ad un livello più alto rispetto al GCD, adattandosi alla perfezione per lo sviluppo di applicazioni Cocoa.

2.4 Interazione con l'Utente: metodi a supporto della Comunicazione

Il concetto generale di comunicazione in un sistema software esprime l'interazione di due o più parti che lo compongono, con lo scopo di ottenere un determinato livello di organizzazione e consapevolezza, mediante scambio di informazioni, siano esse dati direttamente utilizzabili nell'ambiente in cui si sviluppa l'applicazione, ché elementi di sincronizzazione e coordinamento delle parti che compongono il prodotto software.

Con la nascita di una vasta eterogeneità di prodotti software, ognuno ideato per un particolare ambito di applicazione, gli approcci alle proble-

matiche di comunicazione si sono diversificati; questo fattore ha assunto un'importanza crescente con l'aumento della complessità delle singole applicazioni, ma non solo. Il secondo fattore discriminante è costituito dalla prepotente comparsa di applicazioni mobile; infatti molte delle soluzioni sfruttate fino ad ora divengono inappropriate per un ambiente caratterizzato da una comunicazione di tipo *event-driven*, ovvero con un flusso applicativo di esecuzione determinato dalla successione di eventi, sotto forma di messaggi scambiati o, come nel caso specifico, il risultato dell'output di sensori e interazioni dell'utente con il dispositivo di tipo *touch*.

Inevitabilmente si sono introdotte nuove soluzioni per supportare i meccanismi base non sufficientemente evoluti e specifici per soddisfare un ventaglio di sotto–possibilità così vasto.

Nel caso di Apple, essa ha riutilizzato meccanismi già presenti e collaudati nella piattaforma desktop, introdotti però solo in Mac OS X, adattandoli in relazioni a fattori tecnologici, di forma ed utilizzo.

Avendo già effettuato un'analisi delle principali caratteristiche architetturali e dei meccanismi messi a disposizione per la gestione della memoria e a supporto della programmazione concorrente, è sempre più evidente la precisa intenzione di ottenere un prodotto software mobile, come lo è iOS, che sia considerato un *surrogato* di un'unica infrastruttura, provvisto di un ampio set di strumenti e tecnologie comuni.

Nel seguito dell'analisi verranno considerati i principali meccanismi e paradigmi di comunicazione fra oggetti, introdotti nel framework UIKit e Foundation; in particolare:

- Target-Action;
- Delegazione e Sorgenti Dati;
- Notifiche.

In questo capitolo si considereranno solamente le parti strettamente inerenti ai metodi comunicativi, perché ognuno di questi paradigmni sarà analizzato sotto il punto di vista progettuale, come design pattern.

Non saranno presenti riferimenti a metodi comunicativi inerenti a tecnologie presenti in framework specifici, quindi molto spesso utilizzate indirettamente, perché non fondamentali ai fini di un primo approccio alla fase di progettazione e sviluppo di applicazioni mobile iOS.

2.4.1 Introduzione agli Eventi

Una caratteristica peculiare degli smarphone è l'essere il supporto per una quantità di sensori costantemente in crescita, utilizzati per acquisire informazioni sull'ambiente esterno.

Gran parte dei dispositivi Apple oggi in commercio, sono dotati di una serie di elementi hardware che hanno la possibilità di generare flussi informativi ai quali l'applicazione può accedere:

- la tecnologia Multi-Touch abilita la diretta manipolazione del sistema di view, inclusa la tastiera virtuale;
- la presenza di tre differenti accelerometri sono impiegati nella misurazione dell'accelerazione lungo i tre assi spaziali;
- il giroscopio ha la funzione di calcolare il grado di rotazione attorno a tali tre assi;
- il GPS (Global Position System), insieme alla bussola, forniscono il supporto per la misurazione della posizione ed orientamento.

Ognuno di questi sistemi hardware produce dati non altamente strutturati, perciò occorre un sistema orientato alla gestione generale degli input di sistema; per fare questo è stato utile considerare il concetto di Evento.

Un *Evento* è un oggetto che rappresenta una determinata azione dell'utente, rilevata dagli apparati hardware del dispositivo ed indirizzata al livello di sistema operativo; infatti tutti i dati ricevuti dalle periferiche di input sono passati al sistema di frameworks sovrastante, i quali, a loro volta, aggiungono informazioni ai dati inviati costituendo un *package* (pacchetto), successivamente inviato come evento per una migliore modulazione delle elaborazioni.

Il concetto di Evento è rappresentato in gran parte da istanze della classe *UIEvent*, presente nel framework UIKit; esso può incapsulare lo stato relativo ad un evento utente, come per esempio i tipi di "gesto touchscreen" rilevati sullo schermo; inoltre mantiene memorizzato il momento in cui è stato generato tale evento.

Attualmente gli eventi sono suddivisi in tre sotto–categorie principali che si dividono in *eventi "touch"*, *eventi di movimento* ed *eventi legati al controllo remoto*; per i primi due tipi non occorre un'ulteriore spiegazione,

mentre l'ultimo di essi, introdotto solo nella versione 4.0 di iOS, rappresenta la classe di eventi originati come comandi tipicamente provenienti da un accessorio esterno conforme alle specifiche Apple, come possono essere considerati gli auricolari.

Per quanto riguarda l'invio dell'evento ad un oggetto per la sua manipolazione, esso avviene seguendo un percorso specifico. Non appena l'applicazione è mandata in esecuzione, essa imposta l'infrastruttura per il *main event–loop*, già introdotto in capitoli precedenti; tale gestore continuo di eventi stabilisce una connessione con i componenti di sistema responsabili dell'invio e ricezione degli eventi–utente di basso livello. In questo modo ogni applicazione riceve gli eventi attraverso la nota struttura chiamata Dispatch Source, installata nel run-loop principale, come previsto dal GCD.

Dato che ogni applicazione deve gestire ogni evento separatamente e nell'ordine di arrivo, questi eventi (non ancora "impacchettati") sono posti in una coda di eventi FIFO. Una volta renderizzata sullo schermo l'interfaccia utente iniziale, l'applicazione è letteralmente guidata dagli eventi esterni che si susseguono.

L'oggetto UIApplication ricopre un ruolo di primaria importanza nella gestione degli eventi, perché è incaricato di prelevare un oggetto per volta dalla coda di eventi, incapsulandolo in un oggetto UIEvent, ed affidare l'elaborazione ad altri componenti specifici. Una volta che si è conclusa l'esecuzione di un determinato evento, sempre UIApplication recupera l'oggetto nella posizione successiva della coda, ripetendo l'insieme di operazione fino alla terminazione dell'applicazione.

La gestione dell'invio del singolo evento avviene seguendo uno specifico percorso, attraverso una cosiddetta "catena di ricevitori" (*Responder Chain*), in cui il primo oggetto–ricevitore costituisce il punto di partenza, a cui è inviato qualsiasi evento prelevato dalla coda dedicata; solitamente non gestisce nessun evento specifico, ma è utilizzato come collegamento per passare l'evento in questione a posizioni della catena relativamente più a monte, fino a ché non si giunga al ricevitore indicato.

Il concetto di ricevitore è modellato tramite l'oggetto *UIResponder*, che presenta svariate sotto-classi; lo stesso UIApplication eredita da tale oggetto, come risulta anche nel caso delle classi *UIView*, *UIControl*, *UIViewController*.

La posizione degli oggetti nella code è correlata alla relazione gerarchica presente fra le parti fondamentali che costituiscono l'interfaccia utente.

Nella maggior parte dei casi il primo ricevitore è una UIView o una UIViewController, mentre in posizioni avanzate si ritrova sempre la finestra (*UIWindow*) dell'applicazione, che ospita tutte le strutture dell'interfaccia grafica.

L'ultima posizione della catena è assegnata all'oggetto Applicazione e nel caso in cui non ci sia alcun ricevitore idoneo, la gestione dell'evento è affidata ad esso, il quale, molto spesso, scarta l'evento, rilasciando la proprietà che deteneva su di esso, in modo tale da liberare le risorse in memoria associate; quando si giunge ad una situazione come questa, evidentemente l'evento generato non era stato ricercato, né atteso da nessun componente dell'interfaccia utente.

2.4.2 Il meccanismo Target–Action

L'interfaccia utente di una tipica applicazione consiste in un determinato numero di oggetti grafici, appartenenti al campo di presentazione dell'informazione; la controparte, oggetti di controllo, hanno la funzione di interpretare l'intenzione dell'utente che interagisce con la GUI, ed istruire qualche altro oggetto per soddisfare la richiesta.

Quando avviene un'interazione con l'utente, i componenti hardware del dispositivo generano eventi, accettati dalla parte di controllo che li incapsulati in apposite strutte appropriate per l'ambiente Cocoa, traducendo il tutto in un'istruzione che risulta specifica per il contesto della singola applicazione. Purtroppo, il solo concetto di Evento non porta con sé un quantitativo di informazioni sufficienti circa l'intenzione dell'utente; infatti essi riportano solamente l'istante e il tipo di comportamento avvenuto.

Risulta necessario un meccanismo ideato al di sopra di tutto ciò, che provveda alla traduzione del concetto di evento ed istruzione, chiamato *Target–Action*. Lo stesso meccanismo viene utilizzato in ambiente Cocoa in forme differenti a seconda della piattaforma utilizzata (Mac OS X o iOS); ad ogni modo risulta utile per la comunicazione fra uno o più elementi di controllo ed un qualunque altro oggetto, in modo tale che i primi mantengano le informazioni necessarie per inviare un messaggio al secondo, al verificarsi di un generico evento. Tali informazioni sono rappresentate da due elementi principali:

- l'azione di selezione, che identifica il metodo da invocare;

- il target, l'oggetto ricevente.

L'evento che innesca l'azione può essere di qualunque tipo, anche se il meccanismo target–action è molto spesso usato in relazione ad oggetti di controllo come buttoni o sliders. Per quanto riguarda l'oggetto di controllo, il framework UIKit ha dichiarato ed implementato diverse classi di controllo, tutte ereditanti da *UIControl*, che definisce molti meccanismi target–action specifici per iOS.

2.4.3 Delegazione e le Sorgenti Dati

Il concetto di delegazione offre un modo per realizzare un canale comunicativo specifico, instaurato fra un oggetto personalizzato ed altri implementati e resi disponibili nei diversi frameworks. In iOS esso rappresenta un meccanismo a livello di programmazione, che permette agli oggetti interessati di auto–coordinarsi quando si stanno verificando modifiche in un'altra posizione all'interno del programma, tipicamente attraverso un'interazione con l'utente.

Lo scopo è di delegare le descrizioni che risultano specifiche, in termini di metodi implementati e funzionalità, in un oggetto dedicato, attribuendo a quest'ultimo la responsabilità di definirle in maniera specifica, in modo tale da ottenere una più ordinata e performante soluzione a scambio di messaggi, rispetto alla gestione diretta degli eventi, oltre che risultare una valida soluzione all'ereditarietà, trattandosi di programmazione Object-Oriented.

I metodi all'interno di molti oggetti presenti nei framework, come è nel caso di UIKit, implementato il concetto di delegazione seguendo determinati *protocolli* per aderire all'interfaccia resa disponibile; alcuni tra tali oggetti consentono di personalizzare comportamenti molti specifici, mediante cui è possibile bloccare gli eventi pendenti, alterare il valore di default ed anticipare eventuali comportamenti dell'oggetto delegando.

La delegazione è una tecnica utilizzata in ogni applicazione iOS; infatti anche all'oggetto UIApplication è associato un delegato,
UIApplicationDelegate, che risulta estremamente importante perché ad esso è affidata la gestione gli eventi–chiave atti alla comunicazione fra la singola applicazione e il sistema operativo, come la terminazione del lancio della stessa, la terminazione della sua esecuzione, l'avviso di un basso livello di memoria, un nuovo orientamento del dispositivo oppure la transizione di uno stato applicativo.

Una *Sorgente Dati* è praticamente identica ad una delegato, eccetto per la relazione che intercorre con l'oggetto delegando; il delegato rappresenta un elemento di controllo dell'interfaccia utente, mentre le Sorgenti Dati sono elementi atti al controllo dei dati.

Un esempio sull'utilizzo di quest'ultimo meccanismo di comunicazione è dato dalla richiesta di un UIViewController, in particolare di un UITableViewController (un oggetto di controllo che gestisce la presentazione della view sotto forma di tabella), di popolare o modificare determinate sue celle con specifici informazioni, provenienti dalla sezione di modellazione dei dati.

2.4.4 Sistema Centralizzato di Notifiche

La via preferenziale per passare informazioni fra oggetti è tramite lo scambio di messaggi, nel quale un oggetto invoca il metodo di un altro. Questo modo di operare implica che il mittente sia a conoscenza del ricevente e del tipo di risposta; tale considerazione risulta valida sia per il meccanismo di delegazione, ché per tutti gli altri tipi, basati sempre sullo scambio di messaggi. Alcune volte però l'accoppiamento dei due oggetti, indispensabile per tale modello di comunicazione, non è il più adatto, come può accadere quando sia richiesta l'interazione fra due sotto–sistemi indipendenti, oppure è impraticabile, per la necessità di ricorrere a collegamenti difficoltosi dal punto di vista della scritturazione del codice.

In casi come quelli citati sopra, in cui l'adozione del tradizionale scambio di messaggi non è consigliabile, l'ambiente Cocoa mette a disposizione il modello broadcast delle *Notifiche*.

Nel seguito non sarà fatto alcun riferimento al sistema di *Notifiche Locali*, né al concetto di *Notifiche Push*, utilizzate per informare l'utente circa l'avvenimento di determinati eventi esterni all'applicazione stessa, quando essa è in esecuzione in primo piano, (per esempio l'arrivo di un messaggio di testo o un'appuntamento imminente, provenienti dall'applicazione dedicata); questo perché sono aspetti considerati secondari ai fini di un'analisi svolta sugli elementi sostanziali per la progettazione e sviluppo.

Una notifica è modellata all'interno del framework Foundation, e rappresenta un'istanza della classe *NSNotification*; tale oggetto contiene un nome, come identificativo della notifica stessa, e un oggetto generico (di tipo *NSObject*), utilizzato per scambiare informazioni fra l'elemento che ha introdotto la notifica e tutti gli altri interessati ("osservatori").

Per poter impostare e gestire l'invio in broadcast di notifiche è stata introdotta un'infrastruttura globale chiamata *Centro Notifiche*, la quale consente l'inserimento e la conseguente attivazione di una molteplicità di notifiche, preoccupandosi di far pervenire le informazioni richieste ad ognuno degli oggetti che si sono registrati ad esse.

Ogni oggetto ha la capacità di utilizzare il meccanismo delle notifiche per fare pervenire informazioni circa il risultato di un evento, ma solo a chi è registrato ad essa. L'oggetto che ha reso disponibile una notifica circa un evento che lo riguarda non ha bisogno di avere una conoscenza approfondita degli osservatori, mentre questi ultimi devono mantenere almeno il nome identificativo della notifica. In questo caso, di default, la notifica giunge a ciascun osservatore in modo sincrono, quindi il controllo ritorna all'oggetto da cui la notifica si è originata solo dopo che tutte quante sono state inviate ed elaborate. Inoltre, in un'applicazione multithreaded le notifiche sono inviate tramite lo stesso thread utilizzato dall'oggetto che ha inserito la notifica, il quale potrebbe anche non essere il medesimo thread su cui sono in esecuzione gli osservatori.

Per inserire la notifica in modo asincrono, l'unica scelta è quella di utilizzare la *Coda di Notifiche*, un'ulteriore struttura opzionale, in aggiunta al Centro Notifiche, la quale permette l'inserimento di notifiche con un certo ritardo temporale e di gestirle tramite il concetto di "fusione di notifica", per quelle che risultano similari, in accordo con criteri specifici, impostati dallo sviluppatore.

La Coda di Notifiche agisce come buffer per il Centro Notifiche, mantenendo le notifiche in ordine con una logica FIFO; quando una notifica raggiunge la cima della coda, è automaticamente inserita nel Centro Notifiche, che agisce come descritto precedentemente. Inoltre, essa consente di personalizzare la modalità e l'istante temporale di invio delle notifiche, per esempio inserendola non appena è possibile farlo, immediatamente, oppure anche quando si riscontra lo stato *idle* per il run loop associato al flusso di esecuzione corrente.

In ambito di programmazione concorrente, ad ogni thread è sempre associata una Coda di Notifiche, la quale, come noto, specializza il Centro Notifiche per soddisfare una comunicazione di tipo asincrono; tutti i metodi resi disponibili consentono all'oggetto che mette a disposizione una notifica di riottenere immediatamente il controllo in esecuzione, non appena si è inserita la notifica nella coda.

Una peculiarità della struttura a coda è data dalla possibilità di settare un'opzione, chiamata *coalescing* (*unione*, *fusione*), mediante la quale è attivato il processo di rimozione di notifiche in qualche modo simili ad una che è stata accodata in precedenza; tale possibilità è utilizzata in situazioni in cui vi è la necessità di inserire una notifica per eventi verificatisi al più una volta. In questi casi non è possibile utilizzare direttamente il Centro Notifiche a causa del suo comportamento sincrono; in particolare perché le notifiche sono inserite prima che il controllo ritorni all'oggetto che l'ha originata, quindi non vi è alcuna opportunità di ignorare notifiche eventualmente duplicate, mentre la Coda di Notifiche, utilizzando un meccanismo sincrono, può evitare questo tipo di comportamento.

Sostanzialmente le notifiche costituiscono un meccanismo mediante il quale un determinato oggetto può essere informato in maniera continua su ciò che accade ad un oggetto ulteriore, indipendentemente dal legate che intercorre fra i due; in questo senso è simile al meccanismo di delegazione, anche se presenta importanti differenze. La comunicazione realizzata mediante delegazione interessa due sole entità, utilizzando un percorso di comunicazione *one–to–one*; il sistema di notifiche utilizza una forma di comunicazione che è potenzialmente one–to–many (*broadcast*). Inoltre quest'ultimo meccanismo è particolarmente adatto per l'azione di coordinamento e coesione all'interno dell'applicazione, ponendo meno vincoli rispetto alle relazioni che necessariamente devono intercorrere fra gli oggetti in questione.

Non bisogna dimenticare, però, i vantaggi presenti nel modello di comunicazione one–to–one realizzato nella delegazione; esso offre la possibilità di incidere sulla gestione dell'evento stesso, come la manipolazione del risultato che ritorna al delegando, mentre gli osservatori di una notifica assumono un ruolo tipicamente passivo.

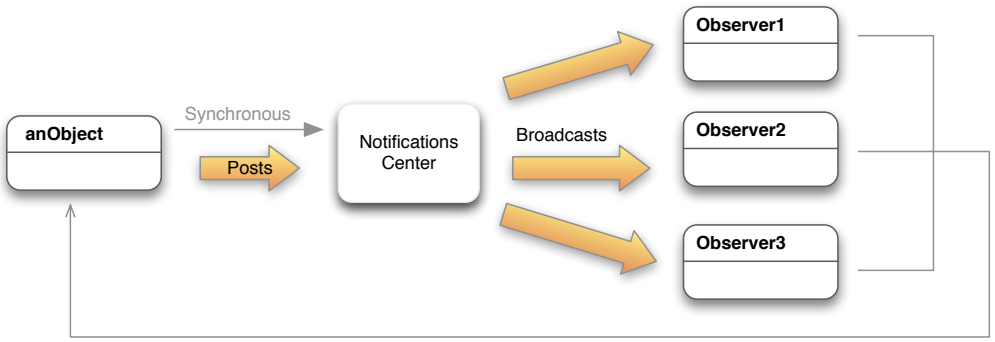

Figura 2.8: Struttura del Centro Notifiche

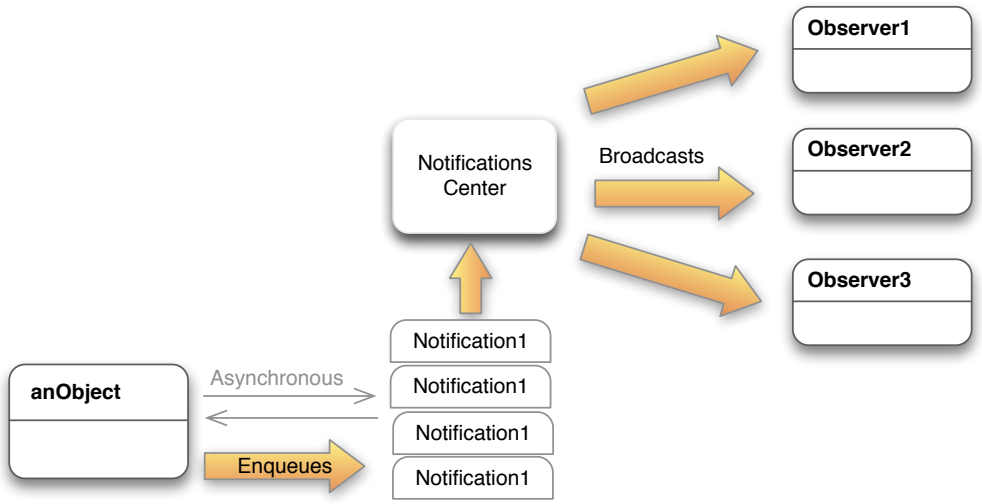

Figura 2.9: Struttura della Coda di Notifiche

Capitolo 3

Design Patterns

Nelle fasi progettuali, dopo un attenta analisi delle problematiche alle quali è soggetta l'applicazione target, una buona conoscenza dei principali design pattern è fondamentale per la scrittura di codice finalizzato alla costituzione di componenti software il più possibile estendibili e riusabili, rendendo eventuali modifiche di facile attuazione, se futuri requisiti o esigenze lo richiedessero.

Le applicazioni basate sull'utilizzo dei design pattern sono generalmente più eleganti ed efficienti, traducendosi molto spesso in una diminuzione delle linee di codice che portano al medesimo obiettivo.

L'importanza che assumono i design patterns nell'ambiente di programmazione Cocoa è nettamente amplificata, data la loro pervasività nella maggior parte delle architetture e meccanismi in entrambi le piattaforme software (Mac OS X, iOS) nelle quali, come dovrebbe essere chiaro dopo i capitoli introduttivi, il sistema di frameworks dedicati fornisce un'infrastruttura cruciale per l'applicazione e in molti casi rappresenta l'unica via di accesso alle risorse sottostanti.

Di seguito si analizzeranno alcuni dei principali design pattern che furono per la prima volta descritti e catalogati in *"Design Patterns: Elements of Reusable Object-Oriented Software"* [9]. Non ci si limiterà ad un riassunto, ma si ricercheranno le motivazioni che hanno spinto Apple ad adattare o modificare alcuni dei principali pattern per lo specifico design applicativo e quali caratteristiche rilevanti debbano possedere per essere considerati così importanti nella progettazione di applicazioni mobile.

Innanzi tutto i Design Patterns rappresentano stili astratti utilizzati per

risolvere problematiche che ricorrono frequentemente all'interno di un particolare contesto; si possono considerare anche come un insieme di *template* pronti all'uso, oppure come linee guida per realizzare un particolare design, il quale in un certo senso è l'istanziazione di un pattern.

Dato che ai pattern non è attribuito un significato assoluto, è possibile applicarli a casi di studio concreti con un certo grado di flessibilità, come è nel caso di applicazioni mobile; questo è l'aspetto che ha permesso un approccio personalizzato, caratterizzando fortemente le applicazioni iOS da tutte le altre concorrenti.

Nell'ambiente Cocoa le classi associate ad un particolare framework e gli stessi linguaggi di programmazione o di runtime, implementano già molti dei design patterns catalogati; in questo modo l'ambiente di sviluppo è in grado di soddisfare gran parte delle esigenze di un normale sviluppatore, utilizzando uno o più di questi adattamenti ad un modello di progettazione. Sono inseriti alcuni tratti e caratteristiche distintive in tali templates perché il design che si vuole ottenere è fortemente influenzato da fattori come la capacità dei linguaggi adottati o da strutture architetturali esistenti, indispensabili per la logica adottata nella piattaforma software. Tali modifiche sono state apportate anche per il sussistere di un certo grado di incompatibilità con le definizioni "classiche" di design pattern, in concomitanza con la comparsa di nuove tecnologie, che hanno affondato le radici nella società moderna.

L'implementazione dei design patterns in Cocoa avviene in viarie forme; infatti alcuni costituiscono caratteristiche proprie del linguaggio Objective-C, in altri casi l'istanza di un pattern è implementata all'interno di una classe o in un gruppo di classi correlate; altre volte il pattern adattato rappresenta l'architettura stessa di un framework principale.

Non tutti i patterns, introdotti e catalogati fino ad oggi, verranno presi in considerazione; nello specifico si effettuerà un'analisi critica dei patterns che rappresentano il modello di astrazione utilizzato da Apple nella progettazione e sviluppo dei meccanismi fondamentali trattati nei capitoli precedenti.

Se in precedenza si è data una descrizione implementativa che rappresenta la soluzione specifica, ora si affronterà, nel corso di questo capitolo, la modalità in cui si può impostare l'architettura logica di tale soluzione, in un percorso a ritroso che focalizza l'attenzione sul meta–livello utilizzato per catturare aspetti essenziali di un sistema software, all'interno di un preciso

spazio concettuale.

Siccome i design patterns variano secondo il grado di granularità e livello di astrazione introdotto, è opportuno suddividerli in un insieme di "famiglie di patterns". Seguendo l'approccio adottato nel testo "Design Patterns-Element of Reusable Object Oriented Software", si è scelto di classificarli i base a due criteri, appartenenti a dimensioni fra loro ortogonali, lo scopo e l'ambito. Il primo criterio riflette il motivo per cui il pattern è stato introdotto, mentre il secondo specifica se il pattern è applicato principalmente a classi, con relazioni più o meno statiche e definite a tempo di compilazione (*compile–time*), o ad oggetti, dove si stabiliscono relazioni caratterizzate da una maggiore dinamicità, le quali possono variare a tempo di esecuzione (*run–time*). La fase di analisi dei principali Design

		Scopo		
		Creazionali	Strutturali	Comportamentali
Ambito	Classi	Adapter	Template Method	Chain of Responsability
	Oggetti	Singleton	Adapter	Command
			Composite	Mediator
			Decorator	Observer
				Strategy

Tabella 3.1: Classificazione dei Design Pattern

Patterns si concluderà con la trattazione del *Model–View–Controller* design pattern e il *Delegate* design pattern, non riportati nella tabella sovrastante, perché considerati pattern compositi, risultato dell'interazione e coesione di diversi design patterns, non sempre appartenenti alle medesime aree individuate.Tali due pattern giocano un ruolo chiave nella progettazione di tutte le attuali applicazioni.

3.1 Processo di istanziazione

I patterns "Creazionali" sono stati introdotti per generalizzare ed astrarre dallo specifico processo di istanziazione, quando risulti necessaria la creazione di oggetti. Contribuiscono a rendere il sistema indipendente dal modo in cui i suoi oggetti sono creati, composti e rappresentati; utilizzano il concetto di *ereditarietà* per variare la classe da istanziare.

Sono divenuti importanti dal momento in cui i sistemi software, attraverso il naturale processo evolutivo, hanno incominciato a dipendere sempre più dal concetto di *composizione* degli oggetti, invece che dall'ereditarietà fra classi. Quando questo è accaduto, si è verificato un graduale cambiamento nel modo di progettare le funzionalità associate ad un determinato oggetto, passando da un approccio che prevedeva l'attribuzione di un fissato numero di funzionalità, alla definizione di un set ridotto di comportamenti fondamentali, ma con la possibilità di realizzare composizioni fra questi ultimi, ottenendo aggregati sempre più complessi.

3.1.1 Singleton Design Pattern: l'oggetto UIApplication

Tale pattern viene utilizzato per assicurare che una determinata classe sia presente in una sola istanza, e per fornire un accesso globale ad essa. Per garantire l'esistenza di un'unica istanza, la quale è oltretutto facilmente accessibile, la soluzione migliore è fare in modo che sia la stessa classe a tenere traccia di ciò. Il diagramma delle classi UML sottostante rappresenta la struttura dell'oggetto Singleton. L'utilizzo del *Singleton* nello sviluppo

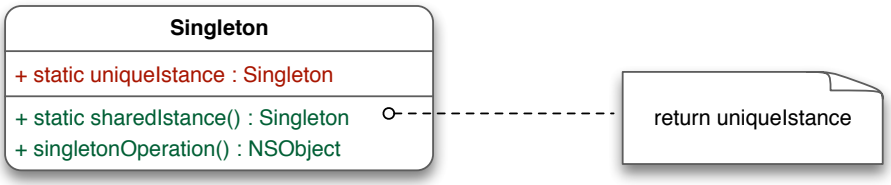

Figura 3.1: Struttura Singleton: diagramma UML

di software per dispositivi mobile diviene necessario quando si modella, tramite un oggetto, una risorsa fisica per usufruire delle funzionalità messe a disposizione, come è nel caso del dispositivo GPS, l'unico componente hardware che fornisce le coordinate della posizione del dispositivo iOS in tempo reale. In questo caso particolare la classe *CLLocationManager*, appartenente al framework *CoreLocation*, è incaricata di garantire un solo punto di accesso (una singola istanza) ad ogni servizio messo a disposizione dal GPS.

Un ulteriore caso d'uso è rappresentato dall'esigenza di utilizzare un gestore centralizzato, il cui scopo si riassuma nella coordinazione degli elementi che compongono il sistema software, e responsabile delle interazione fra questo e l'ambiente esterno. In Cocoa tale componente è essenziale in ogni singola applicazione, implementato dall'oggetto UIApplication, il quale fornisce un punto di controllo e coordinazione centralizzato per ogni applicazione. Esso è creato non appena inizia l'esecuzione dell'applicazione dalla funzione *UIApplicationMain*, dopodiché è possibile accedere alla stessa istanza attraverso il metodo di classe statico `sharedApplication`, dove si eseguono i controlli necessari per garantire il mutuo accesso all'oggetto. Lo snippet di codice sottostante mostra come è realizzato l'oggetto Singleton in Objective-C. Si può osservare che la variabile di tipo *Singleton*, utilizza-

```
@interface Singleton
{
}
+ (Singleton *) sharedInstance;
@end

 - - - - - - - - - - - - - - - - - -

#import "Singleton.h"
@implementation Singleton
static Singleton * sharedSingleton = nil;
+ (Singleton*) sharedInstance
{
    if (sharedSingleton_ == nil) {
        sharedSingleton_ = [[super allocWithZone:NULL] init];
    }
    return sharedSingleton_;
}
```

Figura 3.2: Implementazione dell'oggetto Singleton

ta come punto di accesso condiviso, è trattata come variabile statica, per mantenere valido il vincolo di unicità; inoltre nel metodo che verrà chiamato ogni qual volta si vorrà utilizzare l'oggetto in questione sarà necessario effettuare i controlli opportuni.

3.2 Composizione della Struttura

I patterns "Strutturali" rappresentano soluzioni riguardanti le modalità di composizione di classi ed oggetti, per formare strutture di dimensioni mag-

giori, realizzando nuove funzionalità. Aggiungendo flessibilità alla struttura risultante, tali patterns rendono possibile modificare la struttura della composizione a tempo di run–time, impossibile per classi statiche.

Tramite la modellazione delle classi in una struttura organizzata, i design pattern strutturali rappresentano modalità per creare relazioni fra gli oggetti.

3.2.1 Adapter Design Pattern: i Protocolli

Nella progettazione di software Object-Oriented, alcune volte si ha necessità di riutilizzare classi testate e molto utili in una nuova area di un applicazione, ma è molto comune che le funzionalità da introdurre richiedano nuove interfacce o che non si possano adattare alle classi esistenti.

Per giungere ad una soluzione, superando le problematiche che intercorrono fra le classi già esistenti e le nuove interfacce, si introduce il design pattern *Adapter*, il quale rende possibile l'interoperabilità fra le due classi considerate, cosa che non potrebbe avvenire altrimenti, a causa dell'incompatibilità di base riscontrata. Esso converte l'interfaccia associata ad una determinata classe originale (classe *Adaptee* nei diagrammi UML sottostanti), in quella che l'utente si aspetta di utilizzare (classe *Target* nei diagrammi UML sottostanti).

Sostanzialmente ci sono due metodi per implementare il pattern Adapter. Il primo è chiamato *Class Adapter*; si basa sul concetto di ereditarietà per adattare la nuova interfaccia a quella già esistente. Il secondo metodo è chiamato *Object Adapter*, dove non si eredita dalla classe adattativa, ma si detiene un riferimento ad esso, realizzando una relazione di composizione.
Nel seguito dell'analisi si considererà il metodo Class Adapter, perché utile ad introdurre il concetto di *Protocollo*, una caratteristica del linguaggio Objective-C, specifica del livello di linguaggio, mediante la quale è possibile definire interfacce come istanze del pattern Adapter.

Un Protocollo è essenzialmente la dichiarazione di una serie di metodi non associati ad alcuna classe, quindi se lo scopo è quello di permettere la comunicazione fra un oggetto client ed un altro oggetto, la quale risulta difficoltosa per un'incompatibilità a livello di interfacce, si può facilmente definire un protocollo, adottato dalla classe che si vuole adattare. In generale, per poter essere conforme al protocollo assegnato, è necessario

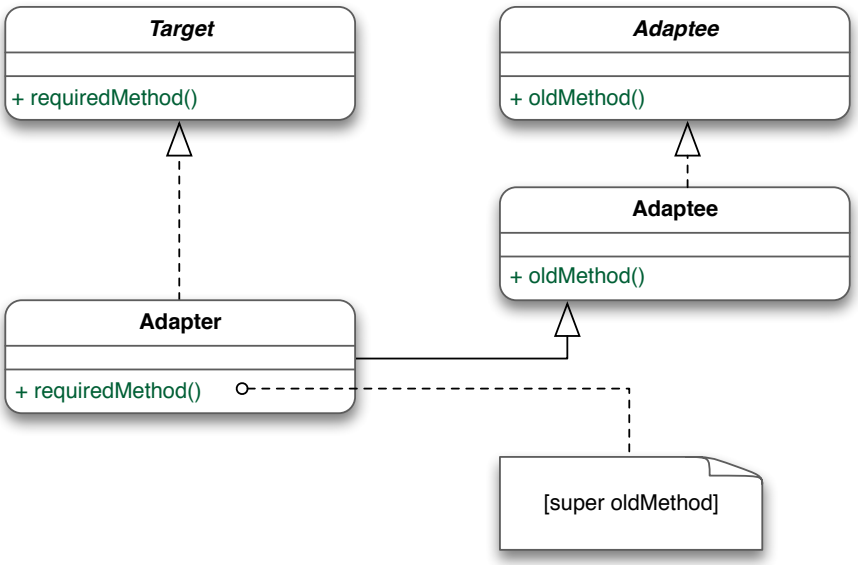

Figura 3.3: Struttura Class Adapter: diagramma UML

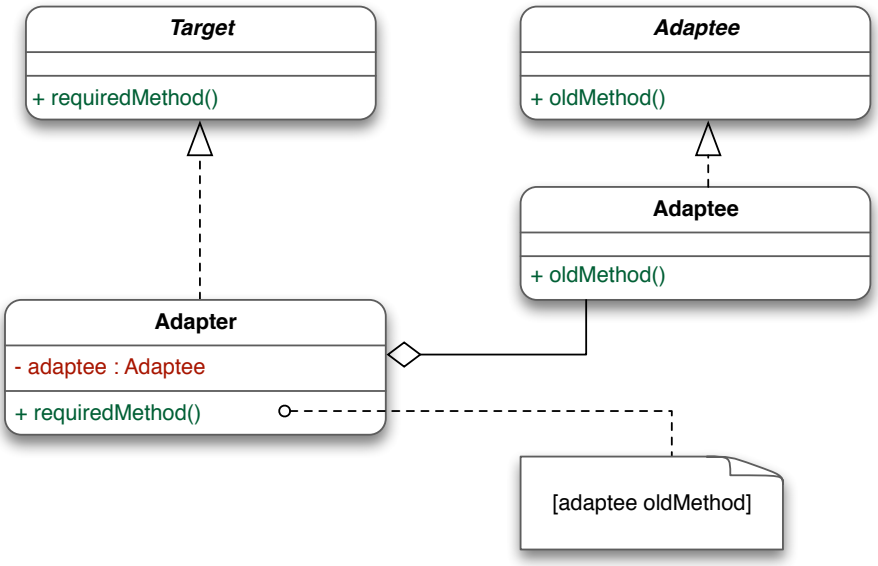

Figura 3.4: Struttura Object Adapter: diagramma UML

implementare i metodi dichiarati necessari, tramite la direttiva *@required*, mentre si può decidere di lasciarne altri opzionali (*@optional*).

Utilizzando i protocolli si rende indipendente la dichiarazione d un set di metodi dalla gerarchia di classi.

Nell'ambiente Cocoa si ha una suddivisione ulteriore in *Protocolli Formali* e *Protocolli Informali*; i primi sono supportati dal linguaggio e dal sistema di runtime, e tipicamente non rappresentano una classe specifica, ma utilizzati come elementi addizionali; i secondi sono specificati solo nella documentazione perché non esplicitamente presenti nel linguaggio e tipicamente descrivono l'interfaccia di una classe, per rendere pubblici i metodi contenuti in una di essa.

3.2.2 Composite Design Pattern: UIView

Il *Composite* pattern permette la disposizione degli oggetti, caratterizzati dallo stesso tipo base, in una struttura gerarchica ad albero, dove ogni "nodo padre" può contenere uno o più "nodi figli"; tale relazione si presenta in maniera ricorsiva all'interno della gerarchia. La struttura descritta termina con "nodi foglia", ovvero con elementi che non contengono a loro volta alcun oggetto, ma che presentano la medesima interfaccia comune a tutti i componenti dell'albero.

Data la condivisione del tipo base degli oggetti, è possibile riferirsi, con una medesima interfaccia, in modo uniforme sia all'intera struttura che ad ogni singolo oggetto, ignorando le differenze concettuali fra i due. Nel framework Cocoa Touch si utilizza questo design pattern per modellare la struttura di cui è composta l'interfaccia grafica; in particolare ogni elemento, nodo, è rappresentato dall'oggetto *UIView*, utilizzato per visualizzare il contenuto dell'applicazione sullo schermo, definendo una porzione della finestra totale. Alla base della gerarchia è presenta l'oggetto *UIWindow*, che modella l'idea di "contenitore" più esterno, con dimensioni pari a quelle dello schermo considerato. La classe UIWindow, come è previsto dal Composite design pattern, condivide il tipo base della gerarchia, perché eredita dalla classe UIView.

Nella progettazione dell'oggetto UIView, Apple ha introdotto alcune specificazioni; ad eccezione fatta per l'oggetto UIWindow, ad ogni istanza della classe UIView è attribuita un'unica *superview* e zero o più *subview*, come è reso evidente dai due diagrammi sottostanti. Il modello UML sotto-

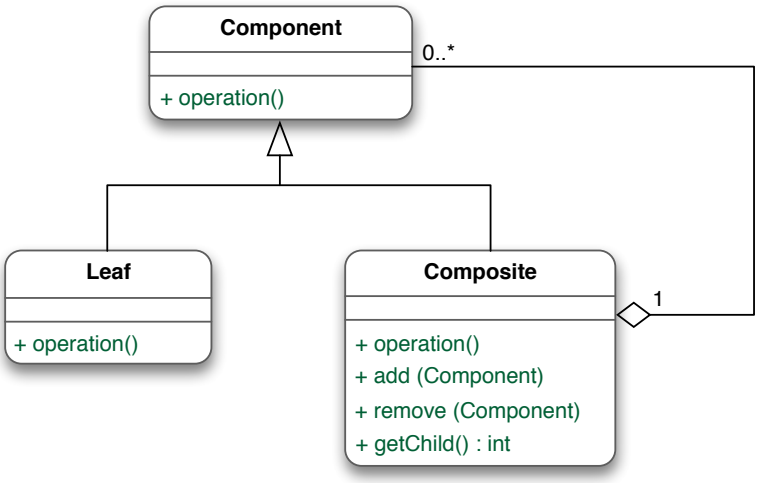

Figura 3.5: Struttura Composite: diagramma UML

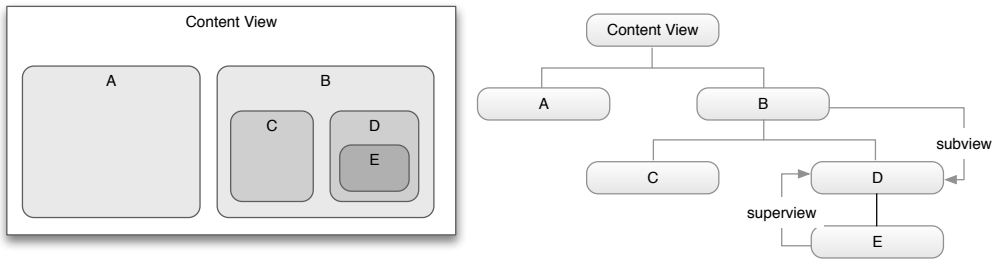

Figura 3.6: Gerarchia di view: Composite design pattern

stante rappresenta la classe UIView, in cui sono riportati alcune dei metodi e proprietà rivelanti. Si può notare che ad ogni view è associato un array di subview, dove l'ordinamento riflette la disposizione visibile sullo schermo, da quella disposta sullo sfondo, (posizione 0), a quella presente in primo piano. La composizione costituita dagli oggetti UIView gioca un ruolo fondamentale sia per la visualizzazione del contenuto, ché per la risposta ad eventi. Ogni volta che è richiesta la visualizzazione di una porzione della finestra, il messaggio è inviato prima alla superview e successivamente alle subviews, tramite la medesima struttura ad albero definita dal pattern Composite. Inoltre, tale struttura unificata di views è utilizzata come *ca-*

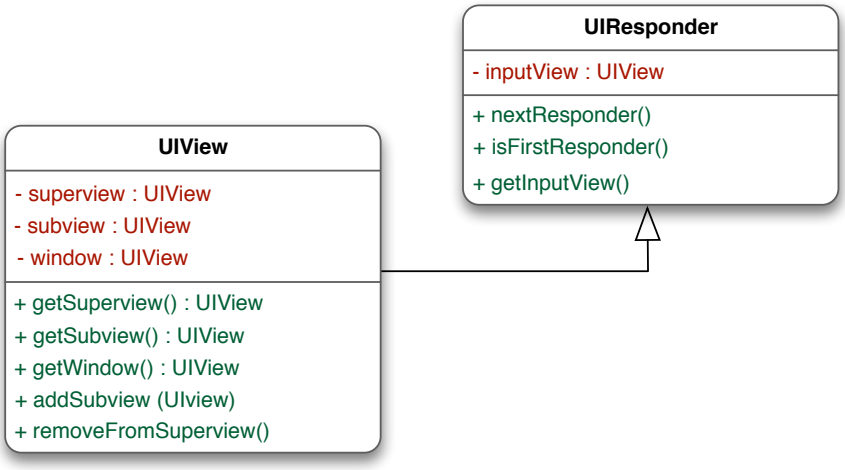

Figura 3.7: Struttura UIView: diagramma UML

tena di ricevitori (*Responder Chain*), per propagare la ricezione di eventi ed azioni dell'utente, concetto modellato mediante l'ausilio di un design pattern dedicato, il *Resonder Chain design pattern* e descritto in uno dei capitoli successivi.

3.2.3 Decorator Design Pattern: le Categorie

É utilizzato per introdurre responsabilità addizionali in un determinato oggetto, dinamicamente. Così facendo è possibile proporre un'alternativa flessibile al concetto di sottoclasse, quando risulti necessario estendere le funzionalità di base, senza ricorrere alla definizione di un'ulteriore classe.

Il pattern Decorator permette sostanzialmente di incorporare l'insieme dei comportamenti designati, senza dover modificare o adattare la struttura delle classi progettate in precedenza. La modellazione classica di questo pattern prevede la presenza di una classe generale che dichiara alcune operazioni comuni, utili agli altri componenti concreti; fra queste due tipologie di classi vi è la mediazione del Decorator, il quale ha il riferimento della classe generale, richiamata per eseguire le operazioni comuni, mentre i componenti concreti si specializzano, estendendo le loro funzionalità dalla classe Decorator, senza dover apportare modifiche alla classe che rappresen-

ta il componento base. Il Decorator design pattern è largamente utilizzato

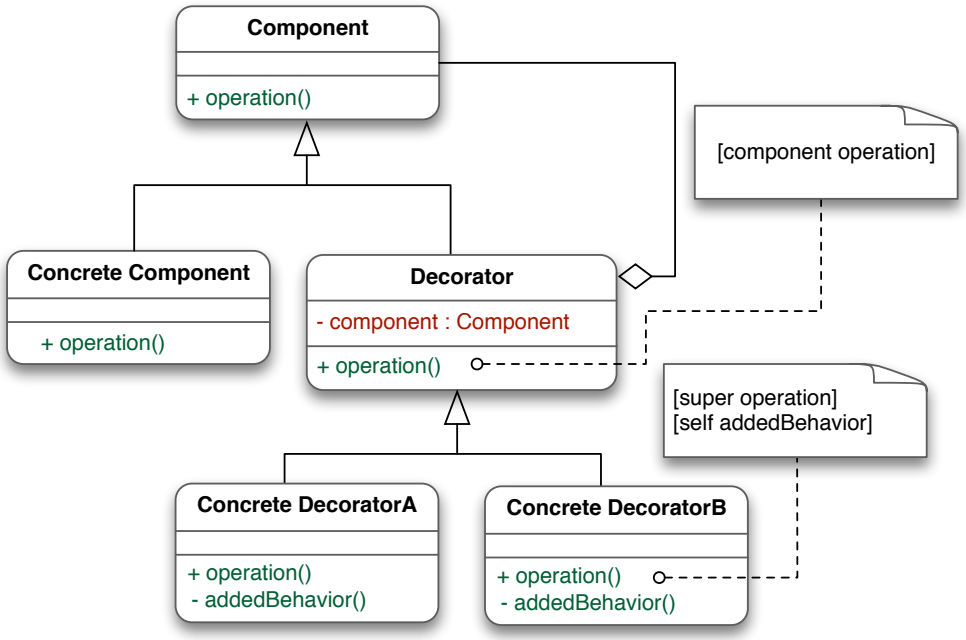

Figura 3.8: Struttura Decorator: diagramma UML

in iOS all'interno degli elementi che costituiscono l'interfaccia grafica, come accade in ogni altra piattaforma software, per introdurre alcune proprietà (bordi delle finestre) o funzionalità (scrolling delle views) solo in alcuni componenti.

Nelle piattaforme Apple è implementata una variante di tale pattern strutturale, riassunta nel concetto di *Categoria*. Una Categoria è una caratteristica propria del linguaggio Objective-C, che permette di aggiungere funzionalità ad una determinata classe, in termini di definizione delle interfacce dei metodi e relativa implementazione, senza ricorrere ad una sottoclasse. Non vi è alcun effetto negativo sui metodi originali; con l'utilizzo delle Categorie i metodi aggiunti divengono parte integrante della classe originaria, a tempo di compilazione.

Anche se il significato attribuito al pattern Decorator rimane immutato, è stato introdotto un adattamento percé l'insieme dei comportamenti aggiunti tramite il concetto di Categoria sono evidenti già a tempo di com-

pilazione, anche se il linguaggio Objective-C supporta il *binding dinamico*, diminuendo in questo modo il livello di dinamicità che caratterizza il pattern. Inoltre per definizione la Categoria non incapsula un'istanza della classe estesa.

Nonostante l'utilizzo di Categorie per implementare il modello di Decorator devi leggermente l'attenzione dalle caratteristiche originali, esse rappresentano un metodo più leggero e veloce per implementare oggetti Decorator che introducono solo poche nuove funzionalità, rispetto all'adozione del vero e proprio subclassing.

3.3 Modello di Comportamento

I patterns comportamentali si interessano degli algoritmi relativi all'assegnamento delle responsabilità tra gli oggetti; essi tendono a focalizzare l'attenzione sul modo in cui i vari oggetti sono interconnessi. Si suddividono in patterns comportamentali relativi alle classi, i quali utilizzano il concetto di ereditarietà per ridistribuire le funzionalità; in questa categoria sarà considerato il pattern *Template Method*. I patterns comportamentali relativi agli oggetti si appoggiano al concetto di composizione invece che sull'ereditarietà; in questo caso si descriveranno diversi pattern, tra cui *Chain of Responsability*, *Command*, *Mediator*, *Observer*, *Strategy*.

3.3.1 Chain of Responsability Design Pattern: UIResponder

L'idea principale alla base del *Chain of Responsability design pattern* è di creare una struttura a catena, dove ogni oggetto detiene il riferimento a quello successivo, e tutti implementano il medesimo metodo per gestire in modi differenti una determinata richiesta, eseguita partendo dal primo oggetto della catena. Se uno di questi non è in grado di gestire la richiesta, la passerà al "ricevitore successivo" (*Successor Responder*).

Il diagramma UML sottostante riassume tali relazioni. É interessante visualizzare il modo in cui due istanze degli oggetti formino una struttura a catena, mostrato nel diagramma sottostante. Tramite il pattern Chain of Responsability è possibile disaccoppiare il mittente di una richiesta dal

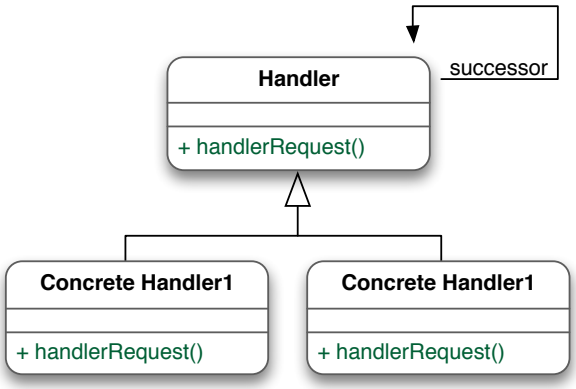

Figura 3.9: Struttura Chain of Responsability: diagramma UML

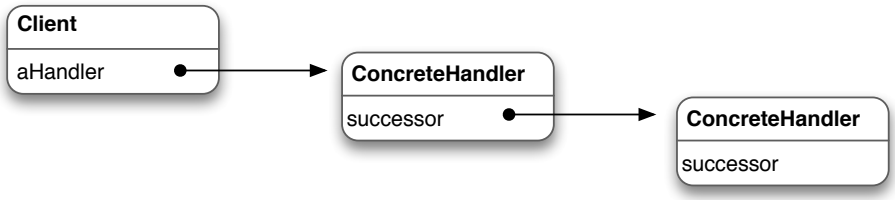

Figura 3.10: Responder Chain: Chain of Responsability design pattern

ricevente, dando la possibilità ad un insieme di oggetti di poter gestire la richiesta.

I frameworks Cocoa includono un'architettura conosciuta come *Responder Chain*, affrontata all'interno del capitolo relativo al supporto per realizzare diverse forme di comunicazione. Tale architettura implementa perfettamente il modello proposto, modellando gli elementi della catena nel framework UIKit, tramite l'oggetto UIResponder.

3.3.2 Command Design Pattern: il meccanismo Target–Action

Il pattern *Command* si basa sull'idea di incapsulare in un oggetto le informazioni riguardanti la modalità di esecuzione delle istruzioni che costituiscono

una determinata richiesta; in questo modo un ipotetico client non ha bisogno di conoscere tutti i dettagli circa la natura della richiesta, continuando ad essere in grado di eseguire ogni operazione su di essa. In altri termini tale pattern disaccoppia un'azione, modellata come oggetto, e il ricevente che la esegue.

Il pattern descritto è implementato in ambiente Cocoa tramite il meccanismo "Target–Action", analizzato nel capitolo omonimo inerente ai fattori di supporto per la comunicazione in applicazioni mobile iOS. Come già affermato, tale meccanismo permette ad un oggetto di controllo, come un bottone, uno slider, un campo di testo o uno switch, di inviare un messaggio ad un altro, in grado di interpretarlo e gestirlo attraverso istruzioni specifiche.

Il frammento di codice sottostante rappresenta l'implementazione del meccanismo Target–Action utilizzato da un'istanza della classe *UISwitch* (modella il concetto di *switch*) per cambiare il proprio stato interno ogni volta che l'utente interagisce con esso spostando il bottone associato da un estremo all'altro di questo componente. Come si può notare, il ricevente è

```
#import "CustomViewController.h"
@implementation CustomViewController
...
- (void) changeVisualOption {
    [prefSwitch isOn];
}
...
CGRect theRect = CGRectMake(196, 8, 50, 40);
prefSwitch = [[UISwitch alloc ]initWithFrame:theRect];
[prefSwitch addTarget:self action:@selector(changeVisualOption)
forControlEvents:UIControlEventValueChanged];
...
@end
```

Figura 3.11: Implementazione del meccanismo Target–Action

la classe stessa che ospita l'oggetto *prefSwitch*, incaricata di gestire anche la visualizzazione del contenuto della view sullo schermo.

3.3.3 Mediator Design Pattern: UIViewController

La progettazione Object-Oriented incoraggia la distribuzione delle funzionalità fra differenti oggetti, ottenendo spesso una fitta rete di interconnessi

fra questi. Anche se tutto ciò aumenta il grado di riusabilità dei componenti all'interno del sistema, l'aumentare delle dipendenze che si vengono a creare rendono sempre più difficile modificare il comportamento dell'intera applicazione, senza dover mettere mano a molti degli oggetti considerati.

Il pattern *Mediator* è utilizzato per definire uno spazio centralizzato, dove le interazioni fra gli oggetti dell'applicazione possono essere gestisce tramite un oggetto Mediator. In tale circostanza tutti gli altri oggetti non hanno bisogno di interagire l'uno con l'altro direttamente; in questo modo diminuisce il grado di dipendenza fra essi. Una possibile struttura del pattern Mediator a run-time è illustrata nel diagramma sottostante. Il pattern

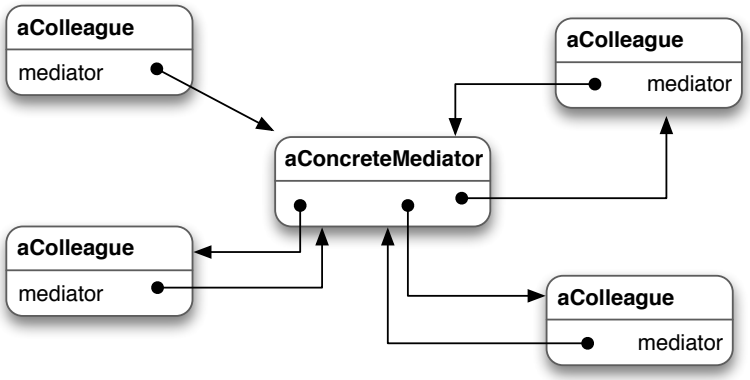

Figura 3.12: Struttura a run–time: Mediator design pattern

Mediator è utilizzato nella piattaforma iOS soprattutto per organizzare le transizioni di differenti views. La classe *ViewController* è al centro di questo design. Essa rappresenta una classe generale per gestire la view che contiene, con il compito specifico di mediare il flusso di dati che intercorre fra gli oggetti UIView e ciò che rappresenta il modello dei dati.

3.3.4 Observer Design Pattern: le Notifiche

Il design pattern *Observer* definisce una dipendenza uno–a–molti fra due o più oggetti, in modo tale che ad ogni modifica allo stato dell'oggetto al quale si è interessati (il soggetto), corrisponda l'invio di un messaggio automatico a tutti gli oggetti "osservatore". È considerato anche un modello denominato *publish–and–subscribe*, per il fatto che, come avviene per la sottoscrizione

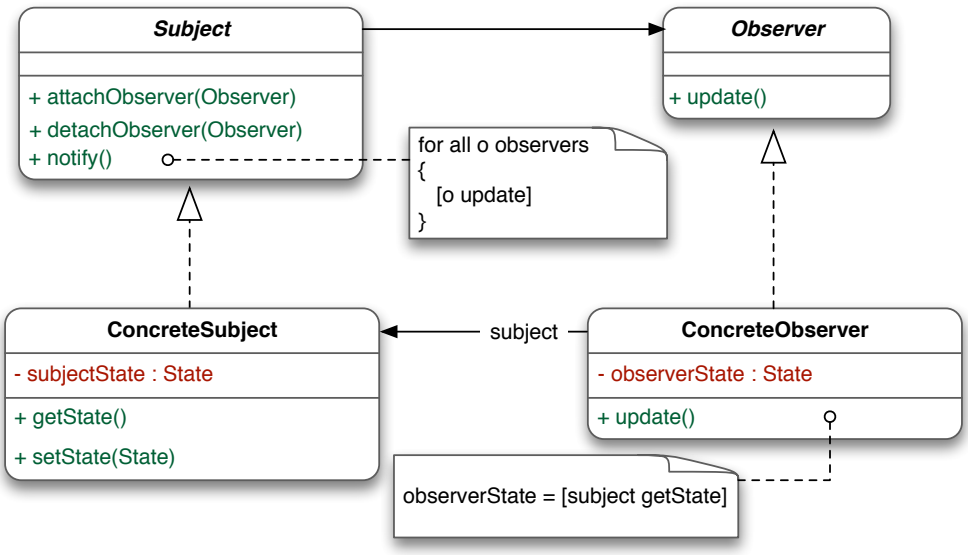

Figura 3.13: Struttura Observer: diagramma UML

a determinate pubblicazioni, il soggetto e i suoi osservatori sono accoppiati in maniera lasca.

Il diagramma UML delle classi che viene propoto offre informazioni aggiuntive circa la struttura degli elementi fondamentali che compongono il pattern Observer. La classe *ConcreteSubject* mantiene una lista interna di tutti gli osservatori che si sono registrati; non appena si verifica un determinato evento, notifica la modifica dello stato inviando un messaggio in broadcast. La classe *Observer* implementa un metodo personalizzato (*update*) eseguito quando riceve un messaggio dal soggetto dell'osservazione; può richiedere lo stato attuale dell'oggetto ConcreteSubject, mediante il metodo *getState*.

Nel framework Cocoa Touch sono state sviluppate alcune classi per utilizzare il pattern Observer senza doverlo implementare interamente, adattando tale pattern mediante la tecnologia delle *Notifiche*, modellate attraverso gli oggetti *NSNotification* e *NSNotificationCenter*, già analizzati dal punto di vista comunicazionale.

Un oggetto che voglia rendere visibili eventuali modifiche inerenti al suo stato deve creare, prima di tutto, un oggetto NSNotification, identifican-

dolo tramite un nome globale associato alla notifica ed inserirlo nel centro notifiche. Quando un oggetto si vuole registrare ad una particolare tipo di

```
...
NSNotification *notification = [NSNotification notificationWithName:@"data
changes" object:self];
NSNotificationCenter * notificationCenter = [NSNotificationCenter
defaultCenter];
[notificationCenter postNotification:notification];
...
```

Figura 3.14: Implementazione Notifiche passo1: Observer design pattern

notifica, viene aggiunto al centro notifiche, specificando l'identificativo della notifica e un metodo da invocare quando è si è ricevuta. Dal momento in

```
...
[notificationCenter addObserver:listener selector:@selector(notify:)
name:@"data changes" object:subject];
...

- (void)notify:(NSNotification *)notification {
    id notificationSender = [notification object];
    //do stuff
}
```

Figura 3.15: Implementazione Notifiche passo2: Observer design pattern

cui non si vuole più essere informati, ogni ossevatore si può rimuovere dal centro notifiche per una determinata notifica.

```
...
[[NSNotificationCenter defaultCenter] removeObserver:listener name:@"data
changes" object:subject];
...
```

Figura 3.16: Implementazione Notifiche passo3: Observer design pattern

3.3.5 Strategy Design Pattern

Il pattern *Strategy* è utilizzato per isolare, in un oggetto, la parte inerente ad un algoritmo. Lo scopo è quello di definire una famiglia di algoritmi, incapsularli uno ad uno e resi intercambiabili. Per fare questo si modella il generico algoritmo mediante un'interfaccia, mentre si implementeranno gli algoritmi specifici in classi concrete. La Struttura appena descritta è presente come composizione in un generico oggetto (*Context*), il quale può accedere ad una molteplicità di algoritmi riferendosi all'interfaccia condivisa da essi.

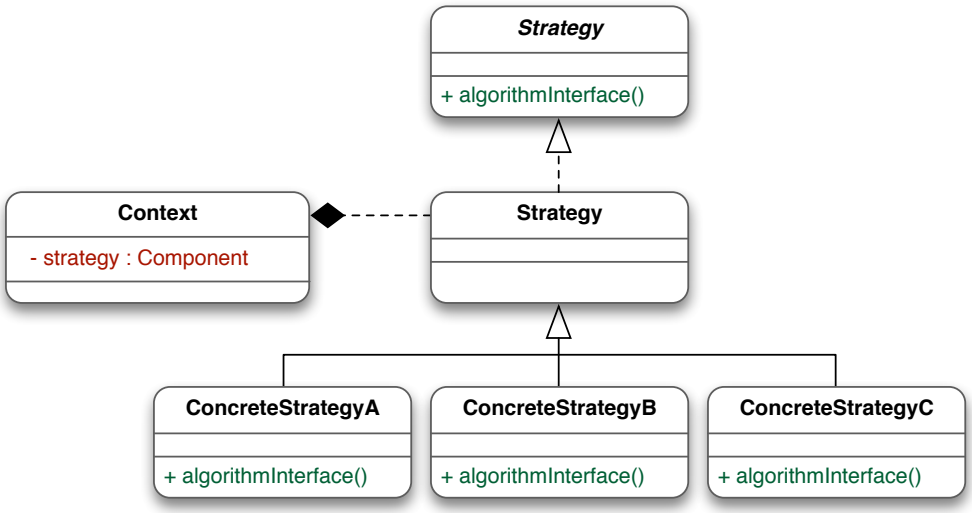

Figura 3.17: Struttura Strategy: diagramma UML

3.3.6 Template Method Design Pattern

Il pattern *Template Method* è uno delle forme più semplici di design patterns utilizzati nella progettazione Object-Oriented di sistemi software. Esso definisce la struttura portante di un algoritmo, all'interno di un'operazione, delegando alle sottoclassi il compito di completare o ridefinire alcune parti dell'algoritmo, senza però cambiarne l'intera struttura.

L'idea di base è definire un algoritmo considerato standard in un metodo (*templateMethod*) appartenete ad una classe generale; all'interno di tale

metodo si richiamano altre operazioni primitive, per le quali si suppone avvenga l'override nelle rispettive sottoclassi.

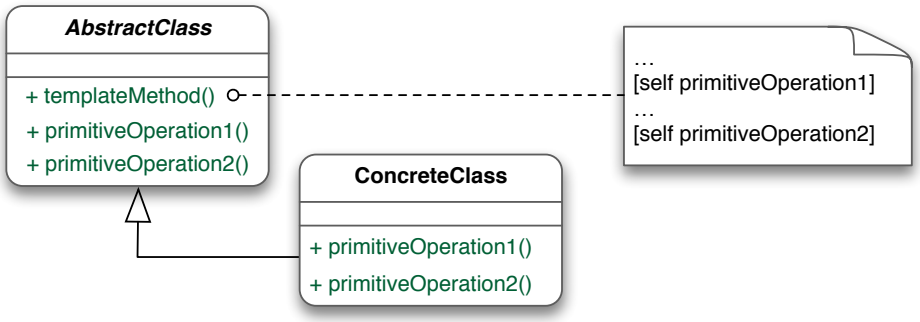

Figura 3.18: Struttura Template Method: diagramma UML

3.4 Delegate Design Pattern

La delegazione, come specificato nel capitolo inerente agli aspetti di comunicazione, è un meccanismo mediante il quale un oggetto, chiamato *host*, detiene un riferimento ad un altro oggetto (il suo delegato) e periodicamente invia messaggi ad esso quando necessita di un input per compiere un determinato task. Tale comunicazione è resa possibile mediante la dichiarazione, senza implementarli, di uno o più metodi che vanno a costituire un protocollo formale o informale; nell'approccio del protocollo formale (il più utilizzato) il delegato è incaricato di implementare i metodi indicati come obbligatori.

Il modello alla base del meccanismo adottato da Apple è conosciuto come pattern *Delegate*. Esso rappresenta principalmente una specializzazione del pattern Adapter, perché si riscontrano tutte le caratteristiche necessarie per essere considerato tale; infatti le funzioni del design pattern Adapter si riassumono nel converte l'interfaccia di una classe in un'altra che i cliente si aspettano di utilizzare, permettendo una cooperazione fra tali classi, cosa altrimenti non possibile a causa del sussistere di un'incompatibilità fra interfacce; confrontandolo con il pattern Delegate si può osservare che:

- i clients sono rappresentati dalle classi appartenenti al framework Cocoa Touch;

- l'oggetto target è rappresentato dal protocollo di delegazione;
- la classe Adapter concreta è, in questo caso, quella che implementa il protocollo.

Inoltre, per implementare il pattern Delegate si utilizza il metodo chiamato Object Adapter, definito nel capitolo relativo al pattern Adapter, perché in questo caso l'oggetto delegando detiene un riferimento alla classe adattiva (delegato), realizzando fra i due una relazione di composizione.

Tale descrizione non esaurisce completamente il concetto di pattern Delegate; il meccanismo di delegazione soddisfa gli obbiettivi comuni anche al pattern Decorator (nel caso specifico si utilizza il metodo implementativo Object Adapter); infatti esso è utilizzato per estendere in maniera incrementale le funzionalità della classe deleganda dinamicamente.

Alcuni framework del livello Cocoa Touch implementano il pattern Delegate come parte del pattern Template Method; infatti il metodo omonimo utilizzato contiene un set di algoritmi predefiniti e parametrizzati, permettendo di delegare l'implementazione delle specifiche funzionalità. Nell'esecuzione di un operazione all'interno del metodo *templateMethod*, sono inviati molteplici messaggi al delegato (classe adattativa) quando è richiesta una funzionalità specifica; ogni informazione dettagliata al riguardo può essere ottenuta da ogni classe deleganda tramite l'interfaccia di cui è fornito il delegato.

Il diagramma riportato mostra un esempio implementativo che assume un'importanza rilevante in ogni applicazione iOS; si considera il meccanismo di delegazione che sussiste fra ogni oggetto UIApplication, che rappresentate il client, e lo specifico delegato (classe *Adapter*), il quale adotta il protocollo *UIApplicationDelegate* (che rappresenta nei modelli la classe astratta Target), alla luce dell'analisi svolta sugli aspetti comunicazionali. La classe *Adaptee*, di cui il delegato ne detiene il riferimento, rappresenta tipicamente un elemento di controllo, in questo caso costituito da una sottoclasse di *UIViewController*.

Oltre che ad un diagramma generale, è riportato il codice relativo all'interfaccia che rappresenta il protocollo utilizzato, all'interfaccia e classe del delegato.

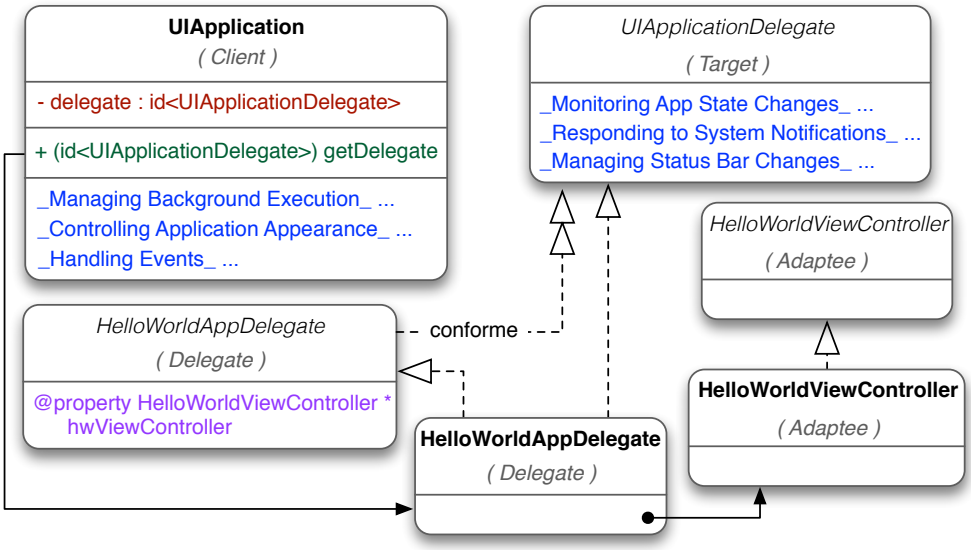

Figura 3.19: Struttura HelloWorld App: Delegate design pattern

```
//  UIApplicationDelegate.h

#import <UIKit/UIKit.h>

@protocol UIApplicationDelegate
...
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions;
- (void)applicationWillResignActive:(UIApplication *)application;
- (void)applicationDidEnterBackground:(UIApplication *)application;
- (void)applicationWillEnterForeground:(UIApplication *)application;
- (void)applicationDidBecomeActive:(UIApplication *)application;
- (void)applicationWillTerminate:(UIApplication *)application;
...
@end
```

Figura 3.20: Protocollo Hello World App: Dlegate design pattern

```
// HelloWorldAppDelegate.h

#import <UIKit/UIKit.h>

@class HelloWorldViewController;

@interface HelloWorldAppDelegate : NSObject <UIApplicationDelegate>
{
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet HelloWorldViewController *viewCtrl;

@end
```

Figura 3.21: Interfaccia Delegato Hello World App: Dlegate design pattern

```
// HelloWorldAppDelegate.m

#import "HelloWorldAppDelegate.h"
#import "HelloWorldViewController.h"

@implementation HelloWorldAppDelegate

@synthesize window=_window;
@synthesize viewController=_viewController;
...
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window.rootViewController = self.viewCtrl;
    [self.window makeKeyAndVisible];
    return YES;
}
...
@end
```

Figura 3.22: Implementazione Delegato Hello World App: Dlegate design pattern

3.5 Model–View–Controller Design Pattern

Il pattern *Model–View–Controller* (*MVC*) è molto diffuso nello sviluppo di sistemi software Object–Oriented per modellare e attribuire un ruolo preciso agli elementi correlati alla presentazione dell'interfaccia grafica; si sono introdotte molte variazioni ad esso, fin dal suo primo utilizzo nel linguaggio di programmazione *Smaltalk*. Non è considerato propriamente un design pattern, ma piuttosto un *pattern architetturale*, perché il suo utilizzo concerne l'organizzazione dell'architettura globale, classificando gli oggetti in base ai ruoli generici che ricoprono in un'applicazione, con l'obbiettivo di giungere ad un disaccoppiamento delle varie parti che compongono il sistema.

Gli oggetti che costituiscono applicazioni Object–Oriented conformi al pattern MVC presentano interfacce ben definite e risultano maggiormente riusabili, specie al variare dei requisiti applicativi. Nel pattern MVC si prendono in considerazione tre principali classi di oggetti, che si differenziano per il ruolo che ricoprono:

Model - classe di oggetti che rappresenta il modello del dominio e fornisce metodi per potervi accedere; generalmente non dovrebbe essere correlata con problematiche relative alla presentazione dell'interfaccia;

View - classe di oggetti con la funzione di visualizzare l'interfaccia utente, permettendo all'utente di interagire con essa, utilizzando indirettamente i dati dell'applicazione;

Controller - classe di oggetti che agisce da intermediario fra View e Model. Tendenzialmente è responsabile del corretto accesso alla classe di oggetti Model, necessario al fine di visualizzarne il contenuto ed eventualmente modificarne alcune parti.

Il pattern MVC non è considerato un vero e proprio design pattern, ma più che altro un aggregato di diversi pattern elementari; grazie a tale integrazione è possibile ottenere la separazione funzionale degli elementi e il percorso comunicazionale caratteristici di un'applicazione suddivisa nelle parti Model, View, Controller.

La nozione tradizionale di pattern MVC utilizza differenti patterns, rispetto alla nozione introdotta da Apple, ciascuno attribuito ad un ruolo ricoperto nel sistema. Nella soluzione tradizionale il pattern MVC è basato sui pattern Composite, Strategy e Observer:

Composite - gli oggetti View sono rappresentati come una composizione di views innestate, cooperando per gestire le modalità in cui presentare l'interfaccia utente e coordinare gli oggetti che essa contiene.

Strategy - l'oggetto Controller implementa una determinata strategia per manipolare uno o più oggetti della classe View o Model.

Observer - gli oggetti appartenenti alla classe Model mantengono attiva una comunicazione per notificare ad oggetti (tipicamente appartenenti alla classe View) interessati nella presentazione del contenuto applicativo.

Il diagramma sottostante rappresenta le modalità tradizionali nelle quali interagiscono i patterns. Nella versione Cocoa, il pattern MVC presenta una

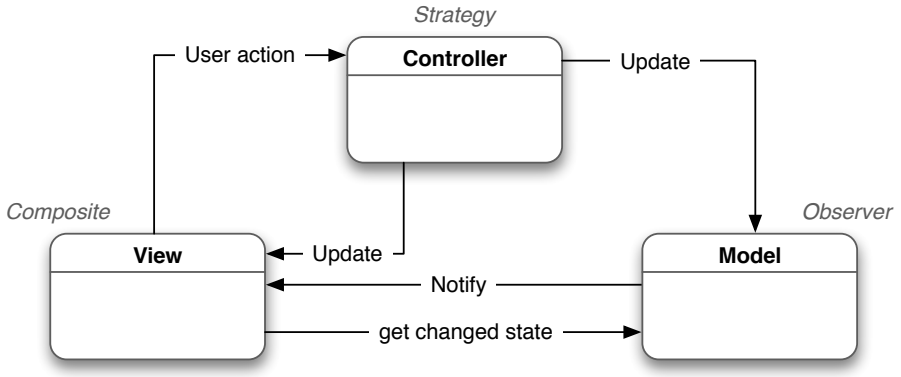

Figura 3.23: Struttura MVC: soluzione classica

differenza sostanziale, la totale separazione fra Model e View, resa possibile attribuendo maggiori responsabilità agli oggetti Controller. L'approccio adottato produce un aumento della complessità della parte relativa al controllo, che ora deve mediare il flusso dei dati fra Model e View in entrambi le direzioni. Risulta necessario introdurre un ulteriore pattern che modelli effettivamente le funzionalità introdotte; in questo caso il pattern Mediator si presta perfettamente per giungere ad una corretta soluzione.

In iOS gli oggetti Controller sono modellati come oggetti *UIViewController*; per semplificare la gestione del modello dei dati e la coordinazione per la

presentazione dell'interfaccia grafica, il framework UIKit fornisce diverse classi pronte per essere utilizzate, che si differenziano a seconda della modalità in cui predispongono la suddivisione dell'interfaccia utente. La classe UIViewController è essenziale in ogni applicazione mobile iOS perché in essa sono implementati molti dei metodi base per la gestione dei comportamenti comuni ad ogni applicazione; ad essa spesso è affiancato un delegato personalizzato, reso conforme ad una serie di protocolli specifici, in modo da rinviare la definizione ed implementazione di funzionalità caratteristiche delle classi che estendono e specializzano UIViewController.

Le due parti restanti del pattern MVC, inerenti rispettivamente al modello dei dati e alla presentazione dell'interfaccia utente, costituiscono i componenti maggiormente riusabili in un'applicazione; la soluzione più tradizionale introduce un grado innegabile di accoppiamento fra i due, non sempre ottimale dal punto di vista progettuale.

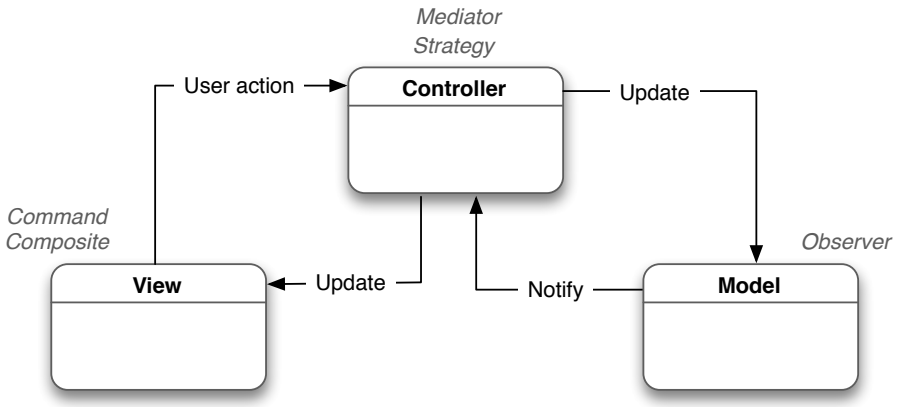

Figura 3.24: Struttura MVC: soluzione Apple

Apple, consapevole dei rischi e benefici che ne derivano, ha preferito introdurre maggiore indipendenza fra Model e View, attribuendo agli oggetto Controller un ruolo centrale, non mi più marginale. A causa della rilevanza di tale posizione assunta, è opportuno ottimizzare la coordinazione di tutti gli eventi che devono essere gestiti all'interno dell'oggetto UIViewController; il design pattern Command risponde a questa esigenza; l'obbiettivo è rendere variabile l'azione del client senza conoscere i dettagli dell'operazione stessa, quindi il destinatario della richiesta può anche non essere deciso

staticamente, ma a tempo di esecuzione, suddividendo in moduli più piccoli l'area di coordinamento. Esso è incorporato negli oggetti View, attraverso l'implementazione del meccanismo Target–Action e viene utilizzato ogni qual volta l'interazione dell'utente produca un evento.

Il diagramma sovrastante rappresenta sinteticamente la struttura del pattern MVC, nella soluzione adottata in iOS.

Capitolo 4

Conclusioni

Lo Smartphone, la classe dei dispositivi mobili dalla quale è iniziata la trattazione, presenta un'infrastruttura hardware–software molto articolata, in cui sono presenti tutti gli elementi che si riscontrano nei moderni computer desktop, ma non solo; infatti raccoglie in sé una certa quantità di sensori e punti di accesso all'ambiente circostante, per integrare sempre più l'esperienza del singolo individuo nel mondo virtuale che i sistemi software rappresentano.

Nell'analisi appena conclusa si è scelto di non focalizzare l'attenzione sugli ultimi accessori e strumenti o, come nel caso specifico di iOS, sui frameworks specifici necessari per applicazioni complesse relative al settore dell'intrattenimento, raccolta e fruizione dati, elaborazioni audio, video; si è invece affrontato il tema di progettazione e sviluppo di applicazione per la piattaforma mobile Apple, mettendo in luce aspetti prettamente concettuali dei quali, molto spesso, ne è sottovalutata la fondamentale importanza, soprattutto dai tecnici informatici che si affacciano per la prima volta al tema *mobile*, dopo aver maturano esperienza sufficiente operando con sistemi software concentrati/distribuiti ed ottimizzati per l'ambiente desktop.

Data la natura della piattaforma e soprattutto la relativa novità degli argomenti trattati, non è stato sempre possibile ottenere informazioni dettagliate circa il reale funzionamento di alcuni componenti del sistema; ad ogni modo l'analisi effettuata ha permesso di tracciare un quadro ingegneristico minimale, ma completo, delle logiche che caratterizzano la piattaforma mobile iOS, riferendosi ad ogni argomento trattato con atteggiamento critico.

Ogni capitolo è stato strutturato come parte integrante di uno specifico percorso, introdotto con una visione d'insieme sulle potenzialità ed architettura di iOS, che ha inizio con l'analisi dei meccanismi ed aspetti riguardanti la performance e stabilità applicativa che interessano i frameworks più vicini all'apparato hardware; successivamente si è considerata l'interazione con l'utente, avvicinandosi al livello applicativo, per terminare considerando i design pattern, quindi la dimensione prettamente progettuale, per la quale risultano necessarie le conoscenze maturate precedentemente.

Apple ha puntato sulla creazione di una piattaforma software mobile che sia la continuazione del precedente successo in campo desktop, utilizzando quindi la medesima struttura architetturale a livelli, i quali sono a loro volta suddivisi in frameworks, ognuno orientato alla gestione di un particolare aspetto del sistema operativo. Facendo buon uso del concetto di interfaccia, tali frameworks sono reciprocamente basati su altri appartenenti a livelli sottostanti, garantendo un buon grado di disaccoppiamento fra le funzionalità basilari e il contesto nel quale vengono utilizzate versione perfezionate. Tutto questo è stato possibile mediante un'attenta analisi delle prospettive future, importante fase progettuale e fattore determinante per un settore in costante crescita come è quello informatico; il risultato di tali riflessioni è stato applicato fin dalle prime versioni di Mac OS X, quando si è dato inizio ad un imponente processo di "ristrutturazione", gettando da subito le basi della futura piattaforma iOS.

Come si può notare, in molti punti della tesi sono presenti continui riferimenti alla piattaforma desktop, circa la condivisione delle soluzioni adottate, sia per quanto riguarda la programmazione concorrente ché molti meccanismi di comunicazione alla base dell'interazione con l'utente e addirittura nei principali design patterns; sostanzialmente sono molte le similitudini che intercorrono fra i due sistemi software, rese possibili mediante l'introduzione dell'ambiente Cocoa, dimostrando che è stato fatto un grande passo in avanti rispetto alla concezione classica di progettazione e sviluppo. In quest'ultima i due ambienti, desktop e mobile, risultavano nettamente separati a causa dell'incompatibilità negli obbiettivi da perseguire e dei metodi di comunicazione fondanti; occorreva utilizzare un approccio ingegneristico differente perché tali erano i meccanismi su cui si fondavano le rispettive applicazioni. Ora invece sembra confermata la volontà di convergere verso un'unica infrastruttura software, su cui erigere piattaforme operative che presentino poche, ma sostanziali differenze che riguardano l'interfaccia gra-

fica per l'interazione con l'utente e i settori a supporto di tutte le tecnologie fisiche che rappresentano i sensori.

Bibliografia

[1] Alex Wright, Get Smarter, 2009, *Communication of the ACM*, vol. 52, num. 1, pp. 15-16.

[2] James A. Landy and Anthony D. Joseph and Franklin Reynolds, Smarter Phones, 2009, *Pervasive Computing - IEEE*, vol. 1536-1268, num. 9, pp. 12-13.

[3] Sarah Allen and Vidal Graupera and Lee Lundrigan, Chapter 1: The Smartphone is the new PC, *Pro Smartphone Cross-Platform Development*, Editors Mark Beckner and Ewan Buckingham, September 2010, Apress.

[4] Apple Inc., iOS Developers Library, 2012, `https://developer.apple.com/library/ios/navigation/`.

[5] Ole Henry Halvorsen and Douglas Clarke, *OS X and iOS Kernel Programming*, Editor James Markham, Dicember 2011, Apress.

[6] Ambra Molesini e Antonio Natali, *Costruire sistemi software: dai modelli al codice*, Esculapio Editore, Ottobre 2009, Progetto Leonardo, Bologna.

[7] Vandad Nehavandipoor, *Concurrency Programming in Mac OS X and iOS*, Editor Andy Oram, June 2011, O'Reilly Media.

[8] Dave Mark and Jack Nutting and Jeff LaMarche, *Beginning iPhone 4 Development Exploring the iOS SDK*, Editor Steve Anglin, January 2011, Apress.

[9] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Editor Brian W. Kernigham, November 2000, Addison-Wesley Professional Computing Series.

[10] Carlo Chung, *Pro Objective-C Design Patterns for iOS*, Editor Douglas Pundick, April 2011, Apress.