ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TITOLO DELLA TESI

PROGETTAZIONE PLATFORM-INDEPENDENT DI APPLICAZIONI PER DISPOSITIVI MOBILI

Tesi in

Ingegneria dei Sistemi Software LM

Relatore

Chiar.mo Prof. Antonio Natali

Presentata da

Maria Letizia Moretti

Sessione III

Anno Accademico 2010/2011

Indice

Introduzione

Dalla necessità di moltissime aziende di realizzare uno stesso prodotto su più piattaforme nasce la sfida di definire nuovi ambienti di sviluppo che facilitino e supportino questo genere di lavoro. Occorre pertanto pensare, definire e sviluppare un ambiente di sviluppo evolutivo. Evolutivo significa che questo ambiente possa essere facilmente esteso, e personalizzato, ovvero concepito e realizzato per le necessità dell'azienda, in modo da renderlo efficiente ed efficace per la risoluzione di una o più classi di problemi specifici, in grado di scrivere codice che scaturisca da un'analisi indipendente dalla piattaforma. La realizzazione di un ambiente di lavoro di questo tipo offre molteplici vantaggi, quali l'abbattimento dei costi di produzione e il miglioramento della qualita' del prodotto software.

L'obiettivo di questo elaborato è quello di tracciare un percorso di riferimento per una software factory che si ponga come obiettivo quello dello sviluppo di applicazioni per l'ambito estremamente attuale che è il *mobile*, i cui dispositivi target sono smartphone e tablet.

Il mondo del mobile è dominato principalmente da due grandi realtà, che sono iOS della Apple e Android della Google. Le due piattaforme presentano modelli di programmazione piuttosto diversi. Progettare la stessa applicazione per dispositivi di entrambi i tipi prevede perciò l'affronto di due stili architetturali differenti. A partire dallo studio delle due piattaforme, è necessario capire quali astrazioni esse offrono nativamente e dove invece potrebbe esistere un *gap* tra ciò che emerge all'analisi del problema e ciò che è possibile fare con i mezzi messi a disposizione dallo stack software di interesse.

Per evitare allo sviluppatore lo sforzo di progettare e scrivere due applicazio-

ni uguali su due piattaforme diverse si potrebbe utilizzare appunto una software factory, ovvero un insieme di generatori Model-To-Model e Model-To-Code che a partire da una specifica scritta con un certo linguaggio *custom* portino alla generazione di codice applicativo già funzionante. Se il linguaggio custom presentasse già le astrazioni necessarie a rappresentare il problema, il progetto dell'applicazione potrebbe già fermarsi all'analisi del problema stesso, perché il gap tra il problema e la piattaforma sarebbe colmato direttamente dal generatore.

L'analisi di questo scenario verrà condotta attraverso un caso di studio proveniente da una realtà aziendale che presenta proprio la necessità di sviluppare un'applicazione su entrambe le piattaforme. L'applicazione è Order Sender, un'App per agenti di commercio presente sui mercati delle piattaforme che sono qui considerate.

Nel primo capitolo si tratterà il problema dello sviluppo su dispositivi mobili, gettando uno rapido sguardo sulle due piattaforme e facendone un rapido confronto. Si introdurrà inoltre il caso di studio. Nel secondo capitolo il caso di studio sarà descritto insieme alla parte platform-independent del progetto. Nel terzo capitolo si descriveranno le soluzioni adottate sulle due piattaforme. Nell'ultimo capitolo verranno approfonditi i modelli e le astrazioni presenti nelle due piattaforme, con particolare attenzione all'aspetto delle interazioni e in rapporto al linguaggio Contact. Le due piattaforme di sviluppo sono ampiamente descritte nelle due appendici, mentre nella terza è riportato un semplice esempio che le mette a confronto.

Capitolo 1

Lo sviluppo di applicazioni su mobile

Sviluppare applicazioni su dispositivi mobili significa interfacciarsi con un ambiente diverso dal mondo della programmazione classico. Nella categoria dei dispositivi mobili rientrano principalmente smartphone e tablet. Questi dispositivi sono sempre più simili a computer ma quello che li contraddistingue sono una serie di caratteristiche particolari, quali:

- le dimensioni ridotte. Ciò impone un utilizzo diverso e nuovo della visualizzazione; non è possibile, come su un computer normale, eseguire tante applicazioni in parallelo e visualizzarle contemporaneamente, perché lo schermo è piccolo. Nella maggior parte dei sistemi è obbligatorio mostrare all'utente una sola schermata per volta.

- le caratteristiche hardware limitate. Gli apparecchi più costosi sono anche più veloci, ma sul mercato sono presenti tutte le fasce di prezzo; in ogni caso, molti dispositivi dispongono di CPU poco potenti e di una relativamente piccola quantità di memoria RAM e di massa. I dispositivi saranno sempre più veloci e potenti fino ad essere effettivamente paragonabili a computer e quindi questa caratteristica andrà a scomparire grazie anche al progresso sempre rapido nel campo hardware, ma se si desidera interfacciarsi con il mercato attuale occorre considerare anche questo aspetto.

- il touchscreen. Lo schermo di questi dispositivi normalmente è touch, quindi deve prevedere un tipo di interazione con l'utente diverso dalla classica combinazione formata da mouse e tastiera. L'utilizzatore normalmente interagisce con le dita, e a volte utilizzando pennette. Questo non è necessariamente vero, poiché molti smartphone sono dotati della classica tastiera QWERTY e hanno tasti funzione veri e propri, come quelli tipici dei cellulari normali. Ad ogni modo, l'utilizzo di uno o più dita sul touchscreen fa sì che si instauri un nuovo tipo di interazione, costituito dalle *gesture*, ovvero un insieme di movimenti che possono essere fatti con le dita sullo schermo e che il dispositivo deve essere in grado di riconoscere ed intepretare. Esempi di queste *gesture* in un'App che mostri le fotografie di un album, sono il movimento del singolo dito per scorrere foto o quello di due dita per effettuare lo zoom.

- la possibilità di collegarsi a Internet e in generale la capacità di interfacciarsi ad una rete pressoché sempre, utilizzando le reti dati degli operatori (GPRS, EDGE, 3G...) laddove il WiFi non sia disponibile.

- fotocamere sempre più potenti e altre periferiche integrate nel sistema operativo.

- la batteria. La vera limitazione di questi dispositivi è costituita dal fatto che essi non sono fatti per essere costantemente collegati ad una fonte di corrente, ma invece sono dispositivi portatili. Le batterie ovviamente devono essere piccole e leggere, quindi non garantiscono un tempo di carica molto lungo. Questo significa che le App e i sistemi operativi devono evitare di sprecare risorse e perciò gli sviluppatori devono trovare il modo di far eseguire i task nel modo più leggero possibile.

- un punto di accesso per acquisire applicazioni appositamente pensate e sviluppate per sfruttare queste caratteristiche.

Proprio perché questi dispositivi dispongono normalmente di scarse risorse hardware e, soprattutto, di poca memoria RAM, ai sistemi operativi è demandata

la responsabilità di mantenere la memoria il più pulita possibile, eventualmente terminando processi ed applicazioni che usano troppa memoria o che non sono considerate troppo importanti per l'utente o per il sistema operativo.

Un'applicazione può essere in background o in foreground e solo un'applicazione alla volta, normalmente, è in foreground. Questo anche per favorire una migliore gestione dell'interfaccia che nella maggioranza dei casi è di dimensione ridotta.

Nel momento in cui si renda necessario liberare la memoria, si parte dalle applicazioni in background, magari inattive, e si va avanti a chiudere le applicazioni considerate il più lontane possibile dall'utente, fino ad arrivare al caso estremo di terminare l'applicazione in foreground che l'utente sta utilizzando in quel momento.

Occorre infine tenere conto dell'ergonomia del dispositivo e della facilità d'uso necessaria al grande bacino di utenti che questo tipo di mercato incontra.

Tutte queste caratteristiche impongono una reimpostazione dell'architettura software rispetto alle architetture classiche. Vedremo nel seguito quali sono le architettura software attualmente adottate dalle due grandi case che sviluppano e distribuiscono sistemi operativi e applicazioni per dispositivi portatili.

1.1 Piattaforme di sviluppo su mobile

Le due piattaforme che si andranno ad analizzare sono Android della Google, e iOS della Apple. Nel seguito si confronteranno rapidamente i due mondi; per una trattazione più approfondita dei modelli di programmazione delle due piattaforme software si rimanda all'Appendice. Si analizzeranno gli stili architetturali, le caratteristiche delle applicazioni e infine si darà uno sguardo ad alcuni aspetti non prettamente legati alle applicazioni, ma di cui lo sviluppatore deve comunque tenere conto quando decide di sviluppare e distribuire software su queste piattaforme.

Android è una piattaforma software sviluppata da Google Inc. che include un sistema operativo, un middleware per l'integrazione sui dispositivi e alcune

applicazioni di base[1]. Esistono diverse versioni di Android e l'ultima release maggiore rilasciata ad oggi è la 4.0, Ice Cream Sandwich. Le varie versioni coesistono normalmente sul mercato e sui dispositivi in uso e alcune sono open source.

iOS è un sistema operativo proprietario di Apple Inc., sviluppato dalla Apple stessa per i propri dispositivi con interfaccia touch. Questi dispositivi sono iPod touch, iPhone e iPad. La versione corrente di iOS è la 5.1. Creare applicazioni, ovvero App, per ambiente iOS significa sviluppare per un target di dispositivi noti e poco numerosi. Infatti, iOS è distribuito e utilizzato soltanto sulle varie versioni dei dispositivi citati. Inoltre, tutti questi dispositivi sono Apple, quindi il produttore del sistema operativo e della piattaforma è lo stesso produttore dei dispositivi.

1.2 Stili Architetturali

Android presenta uno stile architetturale che prevede una struttura fatta di componenti che comunicano a Message-Passing. Le Activity sono le entità che rappresentano una schermata mostrata all'utente e un'insieme di azioni che esso può effettuare; i Service permettono di eseguire task in backgroud; i Content Provider offrono un'interfaccia pubblica per il salvataggio e il recupero di dati su un supporto; i Broadcast Receiver reagiscono a determinati eventi interni o esterni all'applicazione. Tutti i componenti interagiscono attraverso una comunicazione mediata dalla piattaforma, utilizzando tipi di messaggi particolari detti Intent. Questi messaggi sono ricevuti e interpretati dalla piattaforma che si occupa di attivare il task di cui il mittente intendeva richiedere l'esecuzione.

iOS presenta una architettura che ricalca il Model-View-Controller. I componenti fanno parte di una delle tre categorie: Model, View, o Controller. Essi sono oggetti che comunicano tra di loro utilizzando come forma di interazione la chiamata a metodi. Alcune funzionalità fornite dalla piattaforma permettono di astrarre da questo aspetto, ma di fatto all'interno di uno stesso processo le interazioni avvengono di base a method-call. La piattaforma inoltre introduce primitive

e funzionalità che permettono la sincronizzazione e comunicazione tra thread in modo non diretto.

1.3 Sviluppo di applicazioni

In questa sezione si analizzeranno le differenze tra le due piattaforme dal punto di vista del puro sviluppo di applicazioni.

1.3.1 I linguaggi

La prima e più evidente differenza tra le due piattaforme sono i linguaggi. Su iOS si utilizza Objective-C, mentre le applicazioni Android sono scritte in Java. Sono entrambi linguaggi ad oggetti, ma mentre il primo è sviluppato come estensione di C ed è quindi concepito come linguaggio di livello più basso, il secondo è nato come linguaggio ad oggetti ed è quasi puramente ad oggetti.

Un'altra differenza degna di nota è che in Objective-C il typing è dinamico mentre in Java no.

In Objective-C troviamo una serie di astrazioni che invece Java non ha, quali le categorie, i protocolli e il concetto di delegate, nonché le declared properties. I protocolli assomigliano molto al concetto di interfaccia in Java.

In iOS inoltre è necessario gestire la memoria a mano, per cui il linguaggio fornisce gli strumenti per farlo. In Android questo non è necessario perché la memoria è gestita dalla Dalvik Virtual Machine, anche se per applicazioni critiche occorre sempre un po' di accortezza perché il Garbage Collector può non essere efficiente al 100%.

La versione di Java che viene utilizzata su Android non è esattamente quello di un normale JDK, ma si tratta di una versione estesa con una serie di astrazioni ad hoc per la programmazione su mobile. Le classi aggiunte più importanti sono quelle che corrispondono ai componenti delle applicazioni, quali Activity, Intent, Service, Content Provider e Broadcast Receiver. La gestione del ciclo di vita (dalla creazione alla distruzione) di questo genere di oggetti non è quella classica, ma è gestita dalla piattaforma.

1.3.2 Gestione degli eventi

La gestione degli eventi su Android avviene nello stesso modo di Java, a parte il fatto che ci sono eventi caratteristici come quelli associati al tocco dell'utente. In questi casi, la piattaforma intercetta l'interazione dell'utente con lo schermo, crea un oggetto apposito e lo recapita all'applicazione. L'applicazione verifica se ci sono listener associati a quel tipo di evento e invoca il metodo che ne è preposto alla gestione. Il listener normalmente fa parte dell'Activity visualizzata in quel momento, o comunque è nello spazio di memoria che la riguarda, perciò l'evento o è gestito a questo livello o viene ignorato.

In iOS la gestione degli eventi è più complicata e coinvolge una serie di meccanismi. Normalmente l'evento viene intercettato dalla piattaforma e viene creato un oggetto apposito in grado di descriverlo; questo oggetto viene inviato all'applicazione in foreground che è responsabile di capire come gestirlo. Il caso più semplice è che, nel momento in cui l'utente interagisce con una normale View, sia il First Responder a gestire l'evento; il First Responder viene identificato dall'applicazione dopo che essa ha ricevuto l'oggetto associato all'evento. Se ciò non accade, si innesca il meccanismo della Responder Chain che porta la piattaforma a risalire la catena fino a che non trova un elemento in grado di gestire l'evento, fino eventualmente a scartarlo. Un ulteriore caso è quello in cui l'oggetto con cui l'utente sta interagendo sia di tipo `UIControl`; in quel caso l'applicazione cercherà il Target a cui fare eseguire l'Action associata, altrimenti, di nuovo, cercherà un oggetto in grado di gestire l'evento risalendo la Responder Chain.

Per quanto riguarda invece gli eventi esterni all'applicazione, come ad esempio il collegamento delle cuffie al dispositivo, in iOS abbiamo lo stesso meccanismo di gestione degli eventi classico, con un First Responder e una Responder Chain da risalire nel caso in cui il First Responder non sia in grado di gestire l'evento.

Inoltre si possono utilizzare anche le Notifications; registrandosi ad un Notification Center è possibile ricevere notifiche sia relative ad eventi di sistema sia a eventi interni all'applicazione.

In Android invece si deve utilizzare un Broadcast Receiver. Il Broadcast Re-

ceiver può essere utilizzato anche per eventi interni all'applicazione. Per inviare un evento interno o esterno all'applicazione a un componente del genere si utilizza sempre il meccanismo degli Intent.

1.3.3 Interazione con l'utente

L'utente interagisce con un insieme di View sullo schermo, ma ciò che rende effettiva qualunque sua azione è il componente che è responsabile di gestire queste strutture. In iOS, questi componenti sono detti View Controller; in Android, sono le Activity a occuparsi dell'interazione con l'utente. In entrambe le piattaforme, questi componenti sono attivati dalla piattaforma, mettono in campo l'interfaccia (normalmente una singola schermata) che compete loro e sono i principali attori nell'interazione tra l'utente e il resto del sistema.

L'intero ciclo di vita della Activity è regolato dalla piattaforma. Infatti, per quanto sia possibile instanziare, ad esempio, una Activity, è concettualmente sbagliato farlo e non serve a nulla al fine di interagire con l'utente. Le Activity vengono create dalla piattaforma tramite un Intent mandato da un'altra Activity che esprima tale intenzione.

Una conseguenza interessante di questo aspetto è che non è possibile passare dati e oggetti in modo diretto da una Activity ad un'altra, quindi di fatto esse non posseggono uno spazio di memoria condiviso. L'unico modo per condividere oggetti tra Activity è l'utilizzo di istanze statiche, quindi visibili a tutti all'interno del programma, con tutte le problematicità del caso. Altrimenti, occorre passare attraverso gli Intent, che regolano il passaggio di dati tra Activity integrandoli nella loro struttura. È possibile infatti passare tipi primitivi utilizzando gli extra degli Intent, ma anche oggetti serializzabili. Questo implica che gli oggetti vengano serializzati e deserializzati nel passaggio, anzichè semplicemente utilizzare il riferimento come si fa normalmente. Quindi passare un oggetto da una Activity ad un'altra non è una operazione da fare a cuor leggero, e comunque non è possibile con i classici metodi.

In iOS invece i View Controller vengono semplicemente creati e trattati come oggetti; la piattaforma viene coinvolta al momento di rendere il View Controller

così creato quello che correntemente interagisce con l'utente.

1.3.4 Interfacce utente

In entrambe le piattaforme le schermate mostrate all'utente sono costruite come gerarchia. Ogni View Controller definisce la propria schermata in fase di costruzione, e lo stesso vale per le Activity. È possibile, ma sconsigliabile, utilizzare questi componenti per schermate mutiple, modificando la schermata a fronte di un certo evento; ma in questo caso è meglio cambiare View Controller o Activity.

In iOS le gerarchie partono da una Window dentro alla quale ci sono una serie di View nidificate a piacere. Una View può anche essere un `UIControl`, che ha le particolarità descritte in Appendice A, Sezione A.6.5.

In Android le View invece possono essere solo le foglie dell'albero, perché ogni View deve essere contenuta in un ViewGroup che rappresenta un certo modo di disporre i componenti al suo interno.

1.3.5 Gestione del dato

In iOS, esiste una procedura precisa per creare un database interno a un App e una serie di astrazioni che possono essere usate per trattare i dati all'interno dell'applicazione; tutta l'infrastruttura prende il nome di Core Data, e viene utilizzata in generale per il salvataggio di dati su qualunque tipo di supporto, come ad esempio file. XCode mette a disposizione strumenti visuali per la creazione della struttura di un database, sotto forma di una sorta di schema relazionale, e astrazioni quali Entity, Attribute e Relationship per descriverne gli elementi. Lo schema di un database dentro il programma è descritto da un oggetto `NSManagedObjectModel`, mentre un oggetto detto `NSManagedObjectContext` è utilizzato per tenere traccia dei dati correntemente caricati dall'App. Infatti l'applicazione può caricare i dati da supporto e conservarne una sorta di immagine rappresentata dal *context*, e modificarli a proprio piacimento. Quando il *context* viene salvato, le modifiche vengono salvate sul supporto designato e diventano perma-

nenti. La manipolazione dei dati, una volta caricati, avviene attraverso oggetti `NSManagedObject`, che rappresentano le Entity dello schema.

Per essere caricati, i dati sono raggiungibili attraverso query fatte sul `NSManagedObjectContext` con opportuni oggetti dedicati quali `NSFetchRequest`, `NSPredicate`, ecc.

In Android invece esiste un meccanismo completamente diverso che è il *Content Provider*, concepito per condividere dati tra applicazioni (cosa invece impossibile in iOS): il content provider è raggiungibile grazie ad un Intent attraverso un identificativo (un URI) ben preciso, che opportunatamente esteso diventa l'identificativo anche dei dati in esso contenuti. I contenuti del database sono raggiungibili con i metodi che il Content Provider fornisce; comunque non è necessario definirne uno per utilizzare un database. Ad ogni modo, il Content Provider è un contenitore di dati che possono provenire anche da file o altri supporti.

I dati che si ottengono da questo contenitore sono racchiusi in un cursore, che permette di navigarli in base alla loro struttura. Un database viene definito utilizzando un `SQLiteOpenHelper` e l'oggetto che rappresenta il database è un `SQLiteDatabase`.

Su entrambe le piattaforme i database sono di tipo SQLite.

1.3.6 Altre peculiarità di Android

La piattaforma di programmazione su Android fornisce una certo numero di nuove astrazioni molto interessanti, che si distaccano dalla programmazione classica. Ma mentre le Activity sono facilmente paragonabili come utilizzo (ma non come concetto) a un View Controller, e i Broadcast Receiver per certi aspetti possono essere accostati a un Responder per un certo tipo di evento, né gli Intent, né i Service trovano un corrispettivo diretto nella piattaforma concorrente. Il concetto di Intent mette in campo tutta una infrastruttura di interazione tra i componenti mediata dalla piattaforma; i Service, concepiti per l'esecuzione di task in background, sono a loro a volta qualcosa di innovativo.

Al contrario di iOS che è un sistema progettato in modo da rendere le applicazioni chiuse e non raggiungibili dall'esterno, Android permette la condivisione

di dati tra esse tramite l'utilizzo dei Content Provider; inoltre ogni componente di un'applicazione può essere attivato da un componente di un'altra applicazione, se dichiarato pubblico nel file Manifest. Il Manifest è proprio il modo in cui le applicazioni dichiarano pubblicamente ciò che mettono a disposizione e i permessi di cui hanno bisogno per eseguire i loro task.

1.3.7 Altre peculiarità di iOS

In iOS sono introdotte alcune astrazioni originali già a livello di linguaggio. Ad esempio, protocolli, delegazione e categorie (si veda Appendice A, Sezione A.3.2) sono tre modi diversi per implementare il pattern Decorator evitando il subclassing diretto. A livello di piattaforma, le Notifications (si veda Appendice A, Sezione A.6.6) permettono di implementare il pattern Observer.

1.4 Altri aspetti

In questa sezione infine verrano presentati altre caratteristiche dei due mondi di cui occore tenere conto quando si sviluppa software.

1.4.1 Supporto software e dispositivi

Una delle prime caratteristiche che diversificano le due piattaforme è il supporto differente tra software e hardware.

iOS La Apple produce sia il sistema operativo sia i dispositivi.

Ciò implica innanzitutto che è possibile sapere quali sono le caratteristiche hardware dei dispositivi dove gireranno le applicazioni. La Apple fa uscire approssimativamente un modello di iPhone ogni anno e un modello di iPad ogni due; documentandosi, è possibile isolare un numero limitato di tipologie di dispositivi, per cui si può sapere se il telefono sarà dotato di fotocamera o meno, di multitasking o meno, e così via.

In secondo luogo, questo significa che, con qualche eccezione dovuta a un dispositivo particolarmente datato o al non collegarsi ad App Store con una certa frequenza, esiste una sola versione del sistema operativo in circolazione (ce ne possono essere due differenti per iPhone e una per iPad) e il programmatore dovrà interfacciarsi prevalentemente con quella.

Android La Google Inc. ha recentemente comprato la Motorola, ma questo non significa che adotterà una politica come quella della Apple, rendendo disponibile il proprio sistema operativo solo sui cellulari e dispositivi prodotti nella propria casa.

Invece, ci sono svariati produttori che personalizzano versioni del sistema operativo per i propri dispositivi. È perciò una scelta del produttore quella di mettere a disposizione il sistema operativo dell'ultima versione per i propri cellulari, e normalmente anche per modelli diversi della stessa casa produttrice esistono versioni di Android diverse. Ad oggi lo stesso Android 4.0, uscito in data 19 ottobre 2011, è supportato ufficialmente su pochissimi dispositivi.

Questo significa per lo sviluppatore il dover fare i conti con la molteplicità di versioni esistenti in circolazione, e trovare il giusto compromesso tra uso delle *feature* avanzate del linguaggio e un bacino di utenza di una certa dimensione. Inoltre lo sviluppatore dovrà considerare che non tutti i cellulari su cui girerà la sua applicazione saranno dotati delle stesse caratteristiche hardware. Un cellulare potrebbe essere dotato di un led di notifica o di qualche tasto particolare e un altro no; occorre pertanto essere pronti alla massima genericità possibile per permettere l'utilizzo di tutte le funzionalità dell'applicazione.

Inoltre esistono molte *release* non ufficiali, preparate da comunità di sviluppatori a partire da una versione del sistema operativo già esistente, che a volte non sono ben configurate su Android Market. Questo accade proprio perché spesso succede che una casa produttrice venda un tablet o un cellulare senza poi supportarlo adegutamente, lasciando il compratore con un software obsoleto rispetto agli standard di mercato. La presenza di queste release non ufficiali aiuta la rimozione delle versioni più datate, ma aumenta la confusione su quali piattaforme sono

utilizzate e sull'effettivo supporto che Android Market può dare.

1.4.2 Canali di distribuzione

iOS Nel mondo Apple, il canale di distribuzione più diffuso e pressoché unico è App Store. L'altro modo ufficiale di distribuire App è creare un profilo cosiddetto *Enterprise*, per poter installare una certa applicazione su dispositivi registrati presso una ditta. Per poter fare questo occorre eseguire una procedura per essere riconosciuti come azienda dalla Apple allo scopo di ottenere permessi, certificati e *Provisioning profile*, necessari per poter far girare su un dispositivo vero un'applicazione sviluppata con XCode.

Android Android Market è il corrispettivo di App Store, nel senso che è lo Store ufficiale di Android e l'applicazione per utilizzarlo è preinstallata su ogni dispositivo, ma non è l'unico store esistente. Infatti, oltre ad esso esistono altri Store, come quelli specifici dei produttori di dispositivi (Samsung Apps, ad esempio). Inoltre è possibile installare e scaricare un'App semplicemente ottenendo il corrispondente file *.apk*, che non deve essere necessariamente firmato.

1.4.3 Sviluppo e distribuzione di App

iOS Per diventare uno sviluppatore di App per dispositivi Apple, sono disponibili sul sito numerosi manuali e il tool di sviluppo XCode. Per ottenere XCode occorre registrarsi come sviluppatori sull'iOS Dev Center. Al momento di distribuire le App, è necessario ottenere dalla Apple un certificato apposito. Il certificato di distribuzione si ottiene richiedendo un *provisioning profile*.Alla scadenza, occorre rinnovare il profile, altrimenti le App vengono rimosse da App Store. Il profile è necessario anche per ottenere un certificato di sviluppo, che serve a far girare le App su dispositivi reali. I dispositivi su cui possono girare le App in prova devono aver installato il profile e ne viene tenuta traccia da XCode.

Android Su Android, per sviluppare, occorre scaricare e installare l'SDK che viene fornito sul sito di Android Developer. Il software è liberamente accessibile senza registrazione alcuna, e lo è anche il plugin di Eclipse che permette lo sviluppo in questo ambiente in modo semplice e veloce. È inoltre possibile distribuire le App senza passare necessariamente da Android Market e l'*apk* non deve neanche essere firmato, purché il possessore del device target (raggiungibile, ad esempio, attraverso un sito) si fidi di scaricare e installare un'applicazione senza nessun tipo di garanzia sul contenuto. Se si vuole utilizzare il Market occorre registrarsi come sviluppatore.

1.5 La necessità di uno sviluppo platform-independent

Come si è visto le due piattaforme presentano due stili architetturali diversi e astrazioni differenti. Occorre capire come queste differenze influenzano lo sviluppo di applicazioni.

Un caso pratico è il modo migliore per mostrare perchè sia necessario sviluppare questo tipo di approccio allo sviluppo di applicazioni su diverse piattaforme software.

Order Sender è un'applicazione per iPad di cui si è reso necessario lo sviluppo anche per tablet Android. Non esisteva documentazione riguardo all'App originale se non l'App stessa e il codice sorgente. Per affrontare un problema del genere possono essere intraprese più strade.

- Lo sviluppo della versione per Android poteva semplicemente limitarsi all'utilizzo degli strumenti forniti dalla piattaforma di sviluppo in modo che l'App derivata fosse il più possibile simile all'App originale. La cosa non è di per sé semplice, perché si parla di due architetture software, come detto, diverse. Inoltre, molto probabilmente non si sfrutterebbero numerose feature della piattaforma Android che su iOS non esistono.

- Si potrebbe invece ricreare interamente la progettazione facendo *reverse-engineering* sul codice esistente per iPad. Perciò occorrerebbe ricavare il problema a partire dall'applicazione già scritta. Il problema di per sé sarebbe però già sporcato dalle astrazioni presenti in iOS, quindi anche astrarre in questo modo potrebbe non essere qualcosa di banale. Arrivati alla fase di progetto si potrebbe adottare una delle seguenti soluzioni:
  - Si potrebbe pensare di riprogettare l'applicazione per Android, per cui il progetto sarà orientato alla piattaforma di destinazione.
  - Dopo aver astratto il più possibile dalle piattaforme anche in fase di progetto, si potrebbero sviluppare due architetture logiche di progetto differenti: una per Android e una per iOS.
  - Si potrebbe decidere di terminare la fase di progettazione con un'architettura logica platform-independent relazionata a quella della fase di analisi del problema, per poi realizzare, di nuovo, due ulteriori architetture logiche di progetto relative alle due piattaforme.

Alla luce dello studio delle due piattaforme si è ricavato che esse non sono affatto simili (come riportato in Sezione 1.2 e nelle due Appendici) e che il porting diretto delle varie parti dell'applicazione risulta difficile e può portare a scelte sbagliate. Inoltre entrambe le architetture si discostano dalla programmazione classica e perciò non è possibile utilizzare soltanto le astrazioni per la programmazione object-oriented.

È stato perciò necessario ricavare una documentazione di progetto quali analisi dei requisiti, analisi del problema e progetto platform-independent. In fase di analisi del problema si sono utilizzati UML e *Contact* come supporti alla formalizzazione dell'architettura logica. Contact è un linguaggio custom che permette di rappresentare le interazioni. Superata la fase platform-independent del progetto, occorrerà vedere come il modello platform-independent diventa platform-dependent sulle due piattaforme, e quindi anche e soprattutto come le astrazioni di Contact vengono mappate sulle applicazioni. Ciò è interessante anche perché a partire dal linguaggio Contact sono già stati sviluppati generatori di codice per

normali applicazioni Java per desktop; è interessante la possibilità di generare codice a partire da Contact anche per le due piattaforme per mobile, in modo da creare un supporto per lo sviluppatore che lo utilizzi.

Lo sviluppo dell'applicazione è diviso in due parti. Su entrambe le piattaforme l'App è dapprima uscita con una serie di feature, e successivamente è stata aggiunta una funzionalità importante, che è la sincronizzazione dei dati del dispositivo con un server web.

Per quanto riguarda la prima parte dell'applicazione, ragioni di tempo e di praticità hanno portato alla scelta del primo approccio: l'App doveva essere sviluppata e pubblicata nel più breve tempo possibile, con conoscenze limitate di entrambe le piattaforme. Perciò si è partiti da una semplice spiegazione a parole di come era fatta l'App originale ed è stato effettuato il porting su Android nel modo più conforme all'applicazione originale possibile. Rimanere vicini all'applicazione originale, più diffusa ed utilizzata, ha un ulteriore vantaggio: permette all'utente che possegga l'applicazione su dispositivi diversi di accorgersi meno del gap tra i due mondi, cosa molto importante al fine di mantenere la semplicità d'uso, uno degli obiettivi di questo genere di App. Molti utenti infatti utilizzano la tecnologia in modo più o meno consapevole, e sono felici di non dover imparare da capo a fare una cosa dopo la prima volta. Grazie a questa scelta è stato anche possibile familiarizzare con la piattaforma Android e impattare il mondo della programmazione su mobile per la prima volta.

Lo sviluppo della seconda parte dell'applicazione, in quanto meno urgente, è ancora in corso. L'aggiunta della nuova funzionalità pone la necessità di modificare parti dell'applicazione originale. La complessità di integrazione della nuova funzionalità con il resto dell'App, insieme alle caratteristiche del compito che essa dovrà svolgere la rendono particolarmente interessante dal punto di vista ingegneristico. Ciò significa che è necessaria una parziale reingegnerizzazione della parte Android già sviluppata, alla luce dell'ultimo approccio presentato per lo sviluppo. Ovvero, è interessante ripartire dal principio, ricavando a posteriori l'analisi dei requisiti e del problema e, ponendosi all'inizio della fase di progetto, produrre un Platform-Independent Model che venga poi trasformato e adattato in

due architetture di progetto per le due piattaforme interessate. Questo approccio è molto interessante anche perché Order Sender è stata e sarà prodotta in una serie di varianti personalizzate per le esigenze di alcune aziende che lo hanno richiesto. Inoltre l'applicazione è stata pubblicata sugli appositi canali di distribuzione ed ha un bacino di utenza molto grande (soprattutto la versione su iPad, che è uscita prima), quindi è suscettibile di modifiche, migliorie ed aggiornamenti in continuazione.

Capitolo 2

Piattaforme e interazioni

2.1 Verso una piattaforma di sviluppo platform-independent

In questo capitolo si analizzeranno più nel dettaglio alcuni aspetti delle due piattaforme. Lo scopo di questa analisi è di definire un modello di programmazione per esse, ovvero un meta-modello per ogni applicazione che si possa sviluppare su queste piattaforme. Ciò al fine di portare avanti il lavoro di impostazione di una software factory che permetta di sviluppare lo stesso applicativo su due applicazioni diverse.

Un metamodello è un modello esso stesso, e come tale è caratterizzato da tre dimensioni: struttura, interazione e comportamento. La struttura delle due piattaforme è già stata definita (si veda Sezione 1.2 e le due appendici A e B). Il comportamento normalmente è definito in parte dalla piattaforma e in parte dallo sviluppatore. Ciò che interessa approfondire sono le interazioni. Infatti, in un mondo che non è più completamente object-oriented da tempo, ma vede coinvolti thread ed agenti tanto negli ambienti concentrati quanto in quelli distribuiti, è necessario uno strumento di modellazione diverso da UML, che è interamente object-oriented. È proprio nella dimensione di interazione che una modellazione diversa da quella orientata ad oggetti gioca un ruolo fondamentale, poiché la semplice interazione a method call è molto limitante in un ambiente che potrebbe

obbligare a comunicare a message passing o a spazio di memoria condiviso.

Contact è un linguaggio che può aiutare in questo contesto. Esso è un esempio di come un approccio di tipo Model-Driven Software Development possa aiutare lo sviluppatore nella progettazione e implementazione di un sistema software. Come detto, si è scelto di utilizzarlo come strumento di ausilio per la progettazione platform-independent di Order Sender, perciò è interessante vedere come alcune astrazioni presenti in questo linguaggio siano implementabili sulle due piattaforme.

Si descriveranno perciò nel dettaglio le interazioni presenti nelle piattaforme, dopo aver introdotto il linguaggio *Contact*. Infine si vedrà come sia possibile tradurre i concetti di *Contact* sulle due piattaforme. Grazie a questo, sarà anche possibile in futuro definire generatori di codice ad hoc per le piattaforme utilizzando Contact come linguaggio per specificare la struttura e le interazioni di un sistema. Infatti grazie a questo linguaggio e a tool come *XText* è già possibile generare codice Java per normali applicazioni desktop, utilizzando generatori M2M o M2C iniettati in Eclipse tramite l'utilizzo di plugin.

2.2 Contact

Contact[1] è un linguaggio *custom* definito in ambito universitario che permette, attraverso un approccio di tipo Model-Driven Software Development, di generare codice applicativo a partire da una semplice specifica in cui si descrivono la struttura in termini di *subject* e le interazioni tra questi. Con Contact quindi possiamo descrivere la struttura (in modo semplice e generale) di un sistema e l'interazione tra i suoi componenti. Dopo che il codice che costituisce lo scheletro del sistema sia stato generato, il comportamento di questi componenti può essere specificato dallo sviluppatore.

Si illustreranno ora gli elementi fondamentali del linguaggio. In Figura 2.1 è mostrata l'ontologia di Contact come schema `ECore`.

[1] Per un'ampia trattazione di Contact si veda [6]

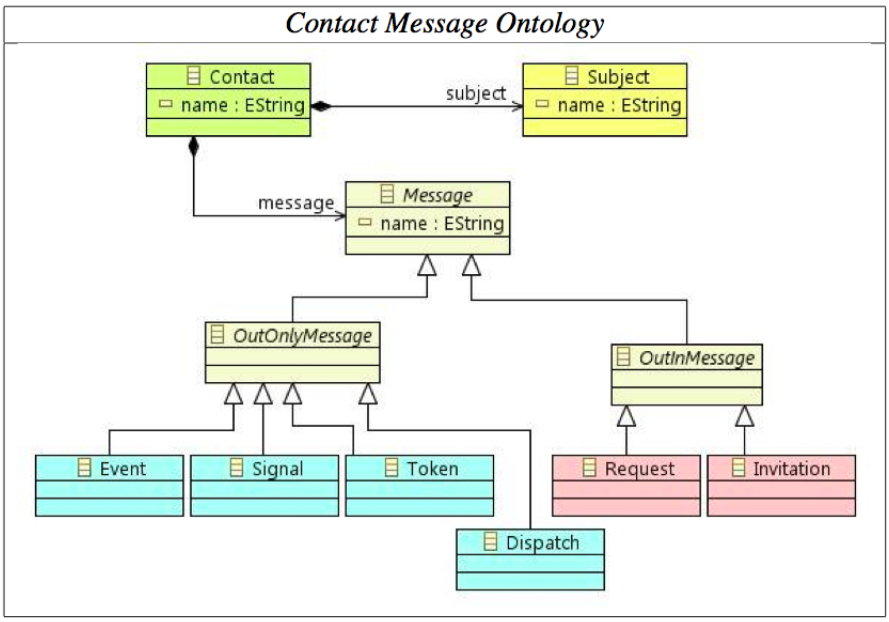


Figura 2.1: Ontologia di Contact in ECore

Le entità interagenti sono dette `subject`. I subject possono essere oggetti, agenti o qualunque altra cosa. Grazie ai subject è possibile definire la struttura del sistema seppur in modo molto generale.

Sono state individuate e formalizzate cinque modalità di interazione tra queste entità, mostrate in Figura 2.2 insieme alle azioni che devono essere intraprese da parte dei subject per gestirle correttamente.

In ognuna di queste modalità si trovano un messaggio, un `Sender` che inizia l'interazione tramite una certa azione e un `Receiver` che la porta a termine tramite un'altra azione.

I messaggi (`Message`) si dividono in `OutOnlyMessage`, ovvero i messaggi che vengono inviati senza attendere una risposta, e `OutInMessage`, ovvero quelli per i quali invece il Sender si aspetta una risposta da almeno un Receiver.

Un `Event` è un `OutOnlyMessage` che viene generato (`raise`) dal Sender con l'aspettativa che cambi l'ambiente che lo circonda. L'evento viene percepito (`perceive`) da 0 o più Receiver qualsiasi in modo *time-uncoupled*.

Message Type	Sender	Receiver
Event	raise	perceive
Signal	emit	sense
Dispatch	forward	serve
Invitation	ask	accept
Request	demand	grant

Figura 2.2: Tabella delle interazioni formalizzate in Contact

Un `Signal` è un `OutOnlyMessage` che viene emesso (`emit`) dal Sender e può essere sentito (`sense`) da 0 o più Receiver qualsiasi in modo *time-uncoupled*. Il segnale viene trasmesso in modo broadcast attraverso un mezzo apposito.

Un `Dispatch` è un `OutOnlyMessage` che viene inoltrato (`forward`) dal `Sender` verso uno specifico Receiver. Il Sender si aspetta che il Receiver serva (`serve`) il messaggio.

Una `Invitation` è un `OutInMessage` che viene inviato (`ask`) dal Sender e si aspetta di acquisire (`acquire`) `ack` da 0 o più Receiver. Il Receiver risponderà allora (`answer`) con una `response`.

Una `Request` è un `OutInMessage` che viene inviato (`demand`) dal Sender e ricevuto (`grant`) da 1 o più Receiver. Il Receiver invierà allora (`reply`) un `ack`.

2.3 Interazioni nella piattaforma iOS

In questa sezione si analizzeranno più attentamente le interazioni che avvengono nella piattaforma iOS.

2.3.1 L'invio di messaggi

Le interazioni in questa piattaforma sono quasi interamente caratterizzate dal message-passing, inteso in questo caso però come chiamata a metodi, quindi bloccante e uno-a-uno. Infatti, nei vari manuali Apple sulla programmazione su iOS e

sul linguaggio Objective-C si parla spesso di invio di messaggi, ma questo è inteso normalmente come method call. Infatti, come spiegato in Appendice A descrivendo la piattaforma di programmazione iOS, nel paragrafo che tratta le caratteristiche del linguaggio Objective-C[2], un oggetto manda ad un altro un messaggio con accluso un selettore, ovvero il nome di un metodo, e una serie di parametri con cui completare la chiamata. Siccome il binding è dinamico, solo a runtime viene verificato che l'oggetto ricevente il messaggio sia effettivamente in grado di rispondere nel modo giusto invocando un metodo. Nel caso in cui tutto vada a buon fine l'invio del messaggio si trasforma in una chiamata al metodo desiderato; altrimenti, la piattaforma genera un errore. Per evitare questo tipo di situazione è possibile effettuare un controllo (che viene effettivamente eseguito a runtime) per verificare che l'oggetto interessato sia in grado di rispondere a un certo tipo di messaggio e quindi di invocare un certo metodo. Ciò è possibile mandando un altro messaggio all'oggetto di interesse per l'invocazione del metodo il cui nome del selettore è `respondsToSelector:`. Questo metodo è dichiarato nella classe `NSObject`, per cui a meno che non si sia deciso di ridefinire la root class della gerarchia (cosa rara e, nella maggior parte dei casi, sconsigliabile), l'oggetto sarà in grado di rispondere a questo specifico messaggio.

Ciò significa che, di fatto, questo invio di messaggi è regolato e mediato dal sistema runtime di Objective-C. Infatti, quando un oggetto vuole invocare il metodo di un altro oggetto, è il sistema runtime che prende il messaggio, verifica che l'oggetto ricevente sia in grado di soddisfare la richiesta dell'oggetto mittente consultando la tabella che contiene tutti i nomi dei selettori dell'oggetto, genera un errore nel caso in cui non lo sia o notifica all'oggetto che deve eseguire quel particolare metodo; infine consegna all'oggetto chiamante il valore di ritorno, se presente, e gli restituisce il controllo per proseguire nell'esecuzione.

2.3.2 Delegazione

La delegazione è un meccanismo a method call, quindi funziona come sopra descritto. La peculiarità della delegazione è che l'oggetto delegante chiama

[2] Si veda Sezione A.3

metodi dell'oggetto delegato; il delegato è un membro della classe dell'oggetto delegante, ed è conforme a un protocollo, quindi risponde certamente ai messaggi che contengono i selettori dei metodi che fanno parte del protocollo stesso. È interessante notare che se un componente dell'applicazione chiama il metodo di un altro componente proponendosi come delegato, normalmente la cosa coinvolge un trasferimento del controllo temporaneo, poiché questo meccanismo è usato per operazioni che richiedono tempo e si spostano su thread in background. Il componente delegato fa perciò "partire" il delegante, che poi si occupa di chiamare i metodi del delegato sul thread di partenza, prendendosi anche carico della comunicazione tra questi. La chiamata dei metodi del delegato preposti a segnalare un avvenimento di un certo tipo può essere vista come la consegna di un messaggio e della conseguente azione da parte del componente delegato, e quindi come una sorta di design pattern per lo scambio di messaggi tra componenti indipendenti. Questo schema è ampiamente usato nella piattaforma per l'esecuzione asincrona di operazioni, come quelle relative alle comunicazioni di rete.

2.3.3 Eventi

Quando viene generato un evento, la piattaforma lo trasforma in un oggetto di tipo `UIEvent` che le applicazioni possano interpretare. Ogni oggetto viene inserito nella coda della applicazione che è in foreground in quel momento, in modo che ogni evento sia processato secondo l'ordine di arrivo in maniera sequenziale. A questo punto l'applicazione può gestire o meno l'evento, a seconda di come è sviluppata. L'accoppiamento temporale tra la piattaforma e l'applicazione dipende dalla velocità dell'applicazione di gestire quantità di eventi più o meno grandi.

Dopo che l'applicazione ha ricevuto l'evento, lo consegna al *First Responder* della *Responder Chain*, e lo fa utilizzando un metodo al quale aggiunge come parametro l'oggetto `UIEvent`. Se il *First Responder* non è in grado di gestire quell'evento l'applicazione risale la *Responder Chain* interrogandone gli elementi allo stesso modo, utilizzando cioè una chiamata a metodo con accluso l'oggetto che rappresenta l'evento in questione.

Si è verificato attraverso alcune prove che il meccanismo della coda in ingresso non permette di bufferizzare tutti gli eventi. Infatti, se il main thread esegue una'operazione time-consuming ad esempio dentro l'handler di un evento associato ad un `UIButton`, altri eventi di tipo Touch vengono persi, e non vengono bufferizzati né gestiti dall'applicazione. Altri eventi però, come le notifiche relative al cambio di orientamento del dispositivo, sebbene non vengano ricevuti immediatamente, si è osservato che vengono bufferizzati e successivamente possono essere gestiti.

2.3.4 Target-Action

Nel meccanismo Target-Action le entità coinvolte sono tre. Il primo è un *Control*, oggetto di classe `UIControl` che può essere ad esempio un bottone o uno slider. Questo oggetto è in grado di gestire una serie di eventi e di mappare su di essi una coppia Target-Action. I Target sono oggetti in grado di eseguire la Action di interesse; le Action invece consistono in nomi di selettori che i vari Target possiedono. Quando la piattaforma riceve l'oggetto `UIEvent` relativo a un Control, lo manda all'Application nel solito modo; essa lo riconosce come relativo al Control e utilizzando il selettore `sendAction:to:from:forEvent:`, invia il messaggio, detto *action message*, al target così che esegua il metodo giusto. Se il Target non è specificato, l'Application si occupa di risalire la Responder Chain dal First Responder in poi per trovare un Responder in grado di gestire l'action message, ovvero di eseguire un metodo il cui selettore sia quello contenuto nell'action message stesso.

2.3.5 Notifications

Utilizzando le notifiche, un oggetto (*observed*) ha la possibilità di avvisare altri oggetti (*observers*) di "eventi" che accadono al suo interno, come cambiamenti di stato. In questo caso, la forma di interazione è uno-a-molti, al contrario del method call che ha preponderato finora, che è uno-a-uno. Le notifiche sono una forma di interazione broadcast.

Un oggetto si registra ad un *Notification Center* per ricevere oggetti di classe `NSNotification` che contengono informazioni riguardo ciò che si desidera notificare. Al momento della registrazione, l'oggetto specifica anche quali tipi di notifiche vuole ricevere. L'oggetto observed deve chiamare un metodo del Notification Center per dare il via all'invio broadcast delle notifiche. Le notifiche vengono "inviate" chiamando il metodo che gli *observers* hanno specificato all'atto della sottoscrizione al center. L'oggetto `NSNotification` che viene passato contiene un nome, un oggetto (che normalmente è l'oggetto che ha originato la notifica) e opzionalmente un dizionario (una struttura dati chiave-valore) che può essere riempito con tutti i dati che si desiderano.

Ci sono due tipi di notification center che possono essere utilizzati. I notification center classici sono utilizzati all'interno della stessa applicazione. Esistono anche notification center distribuiti, che permettono l'invio di notifiche anche ad altre applicazioni, ma non sono stati inseriti dentro iOS; quindi è possibile inviare notifiche solo all'interno della stessa applicazione. Esiste un notification center locale ad ogni applicazione. Le notifiche sono sempre consegnate nel thread da dove la notifica si è originata.

L'`NSNotificationCenter` invia le modifiche in modo sincrono, quindi chiamando uno per uno i metodi dei vari observer registrati per quel tipo di notifica. Infatti i metodi degli *observers* devono essere di rapida esecuzione. Terminato l'invio dei messaggi, l'oggetto observed può riprendere l'esecuzione. Quindi, di fatto, il meccanismo broadcast è implementato comunque con una serie di method-call.

Un'alternativa è l'uso di una `NSNotificationQueue` come ausilio al notification center, che permette l'invio di notifiche in modo asincrono. La `NSNotificationQueue` è una struttura di tipo FIFO (First In First Out, quindi una coda) dove le notifiche vengono memorizzate fino al momento in cui vengono rese visibili. Quando una notifica raggiunge la testa della coda viene resa visibile (postata) nel notification center corrispondente, ed esso comincia a inviare la notifica in modo broadcast (cioè comincia a chiamare i metodi appoisiti dei vari *observers* di cui ha tenuto traccia). Esiste una notification queue associata ad ogni thread;

tutte le queue fanno riferimento al notification center locale all'applicazione, ma comunque è possibile creare center e queue a piacimento.

Nella queue si innesca un meccanismo di selezione, detto *coalescing*, che permette di filtrare le notifiche, sostituendo vecchie notifiche uguali che non sono ancora state postate con una nuova. Ci sono politiche con cui fare il *coalescing*, che potrebbe essere fatto sul nome della notifica, sull'oggetto che l'ha originata o non essere effettuato del tutto.

Le notifiche, dopo essere state accodate, possono venire postate con diverse politiche. Possono essere postate immediatamente (per cui funzionano come le notifiche classiche, tranne che per il fatto che viene effettuato il *coalescing* sulla coda); possono venire postate *ASAP* (As Soon As Possibile) ovvero appena l'iterazione che il run loop[3] sta eseguendo è terminata; oppure possono essere postate quando l'`NSRunLoop` entra in uno stato di tipo *idle*.

2.4 Contact e iOS

Si vedrà ora come implementare alcune specifiche di Contact sulla piattaforma iOS. Questo per preparare il terreno a un generatore di codice da produrre in futuro sulla base di queste considerazioni. Vedremo in particolare quali astrazioni già presenti nella piattaforma suggeriscono una forma di interazione simile a qualcosa che sia già stato introdotto in Contact. In questo modo sarà possibile agevolare una eventuale iniezione di Contact sulla piattaforma utilizzando concetti della piattaforma stessa. Tali astrazioni potrebbero permettere infatti di colmare il gap tra Contact e la piattaforma in modo diretto, anziché fare tutto da zero colmandolo interamente con un'infrastruttura apposita che non tenga conto delle facilitazioni che la piattaforma stessa può offrire.

Request/Response e Method call

Come visto, in iOS la forma preponderante di interazione è costituita dal method call; anche gli eventi, superata la fase in cui essi sono intercettati dalla piatta-

[3] Per la descrizione del concetto di Run Loop si veda Sezione 2.8

forma e accodati nell'applicazione corrente, sono gestiti a method call. Il method call è perciò un modo di fare una `request/response` (ovviamente a livello locale) in Contact. I due `subject` saranno due oggetti qualsiasi, possibilmente che ereditano da `NSObject`, che si conoscono (o quantomeno il primo conosce il secondo). Il messaggio della request corrisponde al selettore del messaggio da inviare al secondo oggetto. Se il secondo oggetto è in grado di rispondere al selettore, farà la `grant` del messaggio (se è in grado di rispondere; altrimenti la piattaforma genererà un errore) e risponderà con una `reply` che contiene il valore di ritorno del metodo.

Notifications e Signal

Un `signal` dovrebbe essere emesso da una sorgente e sentito da una serie di osservatori. Il segnale viene inviato in modo broadcast ai vari osservatori attraverso un certo mezzo di trasmissione. Questo tipo di interazione potrebbe essere implementato in iOS attraverso l'utilizzo di *Notification queue* e *Notification center*. Utilizzando infatti le notifiche asincrone, è possibile stabilire con che politica rendere disponibile il segnale, rappresentato da una notifica, agli osservatori; è anche possibile stabilire come gestire eventuali notifiche duplicate che potrebbero creare problemi. Nel momento in cui una notifica viene accodata, il primo subject non deve più preoccuparsi del destino di essa, quindi ha *emesso* il segnale. Gli osservatori, che ovviamente devono essere registrati al notification center, *sentiranno* il segnale nel momento in cui accade il posting asincrono. I vari osservatori ricevono il segnale (sotto forma di notifica) uno per volta tramite l'invocazione di un metodo apposito. Il mezzo di trasmissione è rappresentato dalla coppia notification queue - notification center. La semantica della `signal` introdotta in Contact prevede però che tutti i segnali inviati siano disponibili per qualsiasi osservatore, anche quelli che si mettano in ascolto in un momento posteriore a quello dell'emissione del segnale. In questo caso il segnale invece viene captato soltanto dagli osservatori già presenti nel sistema nel momento dell'emissione del segnale, quindi si tratta di un'interazione time-uncoupled. Ciò significa che la semantica non è esattamente uguale; si potrebbe definire questo tipo di interazione

come una `signalRightNow`, nel caso in cui la politica di posting della notifica sia immediato. Si potrebbero avere anche altri due tipi di signal, che riguardano le altre due politiche di posting differenti. La prima è `postingWhenIdle`, dove la notifica viene postata nel notification center solo quando il run loop associato ad esso sia inattivo e non stia gestendo altri eventi; in quel caso si avrebbe una `signalWhenIdle`, la cui semantica implica che il segnale venga captato dagli osservatori presenti nel momento in cui il run loop entra in stato di *idle*. Similmente, per l'altra politica di posting asincrono, che prende nome As Soon As Possibile e significa che la notifica viene postata non appena l'iterazione del run loop corrente termina, si avrebbe una `signalASAP`.

Eventi

Quando la piattaforma genera un evento, lo consegna all'applicazione in foreground (e in alcuni casi, anche ad alcune applicazioni in background). In questo caso, l'interazione tra la piattaforma e l'applicazione prende la forma di un `dispatch`, poiché prevede una comunicazione diretta conoscendo il destinatario, con l'aspettativa che esso faccia qualcosa con ciò che ha ricevuto. A quel punto l'applicazione gestirà l'evento cercando un Responder sulla base di una serie di chiamate a metodi.

2.5 Interazioni nella piattaforma Android

Si analizzeranno ora attentamente le interazioni che avvengono durante l'esecuzione di un'applicazione Android.

Oltre al classico method call che nel linguaggio su cui è basata questa piattaforma è ampiamente conosciuto, esiste una nuova forma di interazione costituita dagli Intent. Il meccanismo degli *Intent* si presta ad una serie di considerazioni interessanti.

2.5.1 Intent

Gli Intent sono, per prima cosa, oggetti introdotti nell'ambiente di programmazione Java grazie alle librerie create per lo sviluppo su mobile. Essi rappresentano un'intenzione, o un'azione che deve essere eseguita. Questi oggetti sono di classe `Intent` e contengono il nome di un'azione, il nome della categoria dell'azione ed una serie di attributi opzionali, contenuti in una struttura dati di tipo chiave-valore, che possono essere tipi primitivi o oggetti serializzabili. Questi oggetti sono creati dai componenti dell'applicazione e utilizzati come "messaggi" per comunicare tra i componenti stessi. L'interazione è mediata dalla piattaforma, che riceve l'Intent e dai dati contenuti al suo interno comprende l'intenzione del componente che l'ha inviato. L'intenzione potrebbe essere: l'avvio di una Activity, l'avvio di un Service o la comunicazione via Broadcast verso un Broadcast Receiver. Quindi dei quattro tipi di componenti che possono far parte di un'applicazione Android, solo l'interazione con un Content Provider non utilizza gli Intent. Le interrogazioni su un Content Provider infatti si effettuano utilizzando un *Content Resolver*.

È importante notare che ogni componente ha la proprietà di essere a sé stante con un proprio spazio di memoria. Non è possibile condividere un oggetto tra, ad esempio, Activity se non serializzandolo e inviandolo tramite l'Intent di supporto alla comunicazione tra le due Activity. Ovviamente in questo modo non è possibile condividere la stessa istanza di un oggetto. Perciò componenti diversi non hanno, di fatto, una memoria comune se non quella dell'applicazione, condivisibile attraverso oggetti statici.

Intent e Activity

Quando un'Activity vuole chiamare un'altra Activity, deve utilizzare un Intent e passarlo al metodo `startActivity(Intent)`, che appartiene alla classe `Activity`. Nel far partire una nuova Activity, occorre che l'Activity di origine fornisca alla piattaforma un modo per capire quale Activity deve essere avviata (nel caso in cui non esista già e non debba essere creata nuovamente secondo le politiche dell'applicazione). Ciò può avvenire attraverso l'uso di Intent espliciti,

specificando all'atto di creazione dell'Intent la classe cui appartiene il componente da attivare; oppure attraverso un Intent implicito, cioè specificando ad esempio l'azione che deve essere eseguita con quell'intent. In questo caso, per selezionare quale componente attivare, si utilizzano i *filters*. I filtri sono un modo per specificare per quali azioni, per quali categorie di azioni e su quali dati il componente è in grado di eseguire il suo compito. Ricevuto un Intent che specifica un'azione e/o una categoria e/o dei dati, la piattaforma esamina la lista degli *Intent filters* e stabilisce quale componente attivare, o demanda all'utente la scelta nel caso in cui i componenti che corrispondono siano più di uno. Gli Intent impliciti permettono di attivare un componente che non necessariamente appartiene all'applicazione in esecuzione. Nel caso in cui l'applicazione che contiene il componente di destinazione non sia già in esecuzione, viene fatta partire e le viene allocato un back stack apposito per il proprio task. Nel caso in cui non esista l'Activity specificata o nessuna Activity che abbia dichiarato un filtro corrispondente all'Intent, viene lanciata una `ActivityNotFoundException`.

Oltre a `startActivity(Intent)` esiste anche la variante `startActivityForResult(Intent, int)` dove l'Activity di origine si aspetta che l'Activity in fase di attivazione le restituisca qualcosa. L'Activity attivata deve settare il risultato con `setResult(int, Intent)` prima di terminare, e l'Activity di origine deve implementare il metodo `onActivityResult(int, int, Intent)` per ricevere e gestire il risultato.

Nel momento in cui un'Activity viene fatta partire, l'Activity di origine viene sospesa e non riceve più indietro il controllo finché l'altra Activity non decide di terminare o l'utente non preme il tasto *back*.

Intent e Service

Una Activity può decidere di far partire un Service. Il Service è un task eseguito in background. Background significa non che è eseguito su un thread secondario (se non specificato diversamente, il service è eseguito sul main thread dell'applicazione), ma che è un componente attivabile senza darne diretta comu-

nicazione all'utente (come invece si farebbe attivando un'Activity, che mette in campo la sua interfaccia quando si avvia).

Gli Intent che si utilizzano per avviare un Service possono essere espliciti o impliciti, quindi l'applicazione può utilizzare *Intent filters* ed eventualmente avviare Service che non fanno parte dell'applicazione corrente, allocando un processo dove ospitare l'applicazione se necessario.

Intent e Broadcast Receiver

I Broadcast Receiver sono componenti il cui scopo è la ricezione di appositi messaggi broadcast inviati su tutta la piattaforma. Siccome sono, come gli altri componenti, eseguiti di default sul main thread dell'applicazione, devono reagire a questi messaggi in modo rapido, assolvendo velocemente al compito assegnatogli dallo sviluppatore.

I messaggi sono ovviamente Intent e sono inviati dal componente interessato tramite il metodo `Context.sendBroadcast(Intent)`. Anche però se si utilizza il meccanismo degli Intent, tra utilizzare un Intent per attivare un'Activity e mandarlo via broadcast c'è una grande differenza a livello semantico. Nel primo caso, far partire un'Activity con un Intent significa permettere all'utente di interagire con una schermata diversa. L'utente è consapevole di questa cosa; ciò non è vero per quanto riguarda il broadcast di un Intent, che accade in background.

Nel caso di un normale broadcast, i receiver vengono attivati in modo asincrono e senza un ordine preciso. Esiste anche un tipo di broadcast detto *ordered*, che avviene con il metodo `Context.sendOrderedBroadcast(Intent, String)`. In questo caso i receiver sono attivati uno per volta con un ordine imposto da una priorità, e ognuno di essi può decidere di annullare il broadcast. In caso contrario ogni receiver deve passare il risultato al proprio successivo.

2.6 Contact e Android

Request/Response e Method call

In Android, con il linguaggio Java si trova la classica forma di interazione a method-call che è assolutamente conforme alla chiamata di metodo che tutti conoscono. Quindi anche qui esiste la possibilità di implementare la `request/response` a livello locale. Se gli oggetti (che ovviamente erediteranno da `Object`) sono validi, il primo oggetto può inviare una request al secondo che ne farà sicuramente la `grant` e risponderà (a meno di un eventuale lancio di eccezione, cosa che potrebbe accadere anche su iOS).

Intent

Gli Intent costituiscono una forma di interazione che non si trova in Contact. Infatti, una normale request/response non prevede l'eventuale attivazione del componente di destinazione; la semantica dell'interazione è quindi diversa. Inoltre, la richiesta, oltre che un messaggio, vuole essere la richiesta di un'azione, in particolare quando si parla di Activity e di Service.

Si potrebbe formalizzare questo tipo di interazione con un nuova tipologia di messaggio come `requestForAction`. La semantica di questo tipo di messaggio sarebbe costituita dalla richiesta di un'azione di un certo tipo e prevede anche un trasferimento del controllo, poiché, anche se ogni Activity ha un proprio flusso di controllo all'interno dello stesso thread, un'Activity che ne attivi un'altra resta in attesa finché non le venga restituito il controllo dopo la terminazione della nuova Activity. Oltre a `requestForAction` si avrebbe una specializzazione del tipo `requestForActionWithResult` nel caso la nuova Activity venga attivata con l'intento di ottenere un risultato di qualche tipo. Nel generatore di codice relativo, i due `subject` interagenti sarebbero due Activity (o un'Activity e un Service; in quel caso la semantica sarebbe ancora diversa poiché si tratterebbe della richiesta di un'esecuzione *in background*). La prima Activity eseguirebbe la `requestForAction` inviando un Intent con il metodo `startActivity(Intent)` e specificando nell'Intent stesso l'azione da eseguire. L'azione potrebbe essere

definita nella specifica contact con un nome, una categoria e un insieme di dati che possono poi essere inseriti nel Manifest dell'applicazione utilizzandoli come Intent Filter della seconda Activity.

Signal e Broadcast Receiver

Tenendo conto di questa particolare semantica del mezzo di comunicazione privilegiato su Android, ovvero il meccanismo dell'Intent, si può osservare che il broadcast di un Intent (che è diverso dall'inviare un Intent ad un'Activity) è un modo per implementare una `signal`. Infatti, se un componente dell'applicazione volesse inviare un segnale utilizzando un mezzo broadcast per farlo captare da uno o più osservatori, potrebbe utilizzare il metodo `Context.sendBroadcast(Intent)` specificando nell'Intent quali tipi di osservatori debbano essere attivati. Gli osservatori saranno dei broadcast receiver con un certo tipo di intent filter. Il mezzo broadcast è la piattaforma, che si occupa di inviare gli Intent ai componenti interessati ed eventualmente di attivarli. Anche qui, la semantica dell'interazione non è però quella della `signal` definita in Contact, ma è una `signalRightNow`, poiché il segnale viene inviato soltanto ai ricevitori attualmente presenti nel sistema e non a quelli futuri.

2.7 Richiesta di azioni

In entrambe le piattaforme esiste la possibilità di far eseguire un'operazione di un certo tipo su un thread secondario, che non richieda interazione con il thread originale se non in fase di startup. In iOS questo meccanismo è fornito dalle `NSOperation`. `NSOperation` è una classe astratta la cui implementazione concreta va a specificare quale deve essere il *task* che essa dovrà eseguire. Per avviarne una, deve essere utilizzata una `NSOperationQueue` che permette di mettere le operazioni in coda e di eseguirle una per volta. Su Android esiste un meccanismo simile, che è l'IntentService. L'IntentService è un particolare tipo di Service che incapsula la gestione di un *worker thread* su cui far eseguire un certo compito. Un Service di questo tipo viene creato in occasione della prima richiesta di ese-

cuzione (che avviene attraverso un Intent) e viene distrutto quando l'esecuzione è terminata. Se durante l'esecuzione arrivano altre richieste, vengono messe in coda ed eseguite una per volta dal Service già creato. In entrambi i casi quindi esiste la possibilità di definire un certo tipo di *task* da far eseguire su un thread secondario in modo trasparente, eventualmente accodando richieste multiple senza creare problemi di condivisione delle risorse.

2.8 Il problema dell'allarme

Quando un sistema deve gestire una serie di allarmi, deve essere in grado di reagire prontamente ad un certo tipo di messaggio. Deve essere perciò dotato della capacità di scegliere quali messaggi ricevere o di interrompere ciò che stava facendo per accogliere una richiesta di esecuzione urgente di qualcosa. Si vedrà nel seguito quale è il gap tra le astrazioni fornite dalle due piattaforme e questo specifico problema, in modo da avere già pronta un'impostazione da utilizzare nel caso sia necessario risolvere un problema (o una classe di problemi) in cui emerga la necessità di gestire messaggi con priorità che richiedano immediata reazione, ovvero allarmi.

L'idea di base è costituita da alcuni elementi: per prima cosa ciò di cui si ha bisogno è un componente dotato della capacità di ricevere più messaggi di tipo diverso, in modo che essi siano bufferizzati da qualche parte e non vengano persi nel caso in cui non siano gestiti subito. In secondo luogo occorre che il componente che gestisce le richieste non si blocchi gestendo altre richieste ma resti reattivo per la ricezione di nuovi messaggi.

Un componente simile è realizzabile su entrambe le piattaforme: Android è dotato di un `HandlerThread`, che è un thread che incapsula il meccanismo di gestione di una coda di messaggi, permettendo di specificare nell'implementazione il comportamento a fronte di nuovi messaggi. iOS invece offre *notification centers* e *notification queue*, per gestire code di notifiche. In entrambi i casi è necessario far partire un thread apposito per gestire ogni messaggio/notifica in modo che l'ascoltatore resti reattivo.

In entrambe le piattaforme inoltre esiste il concetto di priorità associato a un thread, ma la cosa viene mappata sul sistema operativo, perciò la cosa non è completamente affidabile (anche se su iOS, dove il sistema operativo e i dispositivi sono sviluppati a braccetto con la piattaforma di sviluppo, è più prevedibile il comportamento piuttosto che sulla molteplicità di dispositivi che montano varie versioni di Android).

Purtroppo in questo modo non è possibile gestire eventi prioritari come gli allarmi. Occorre aggiungere una parte di implementazione dove la coda viene costantemente svuotata e ogni messaggio valutato, bufferizzato e gestito in un secondo momento nel caso in cui non sia relativo ad un allarme. Sarebbe desiderabile l'utilizzo diretto di una coda con priorità.

Un'altra astrazione interessante messa a disposizione dalla piattaforma iOS consiste nel Run Loop, rappresentato dalla classe `NSRunLoop`. Il Run Loop rappresenta il concetto di ascoltatore di eventi che provengono da una o più sorgenti. Ogni applicazione ha associato un Run Loop per gestire gli eventi che provengono dal sistema; essi possono essere sia relativi all'interfaccia sia riguardanti altri tipi di avvisi che possono arrivare dal sistema, come un *memory warning* o il fatto che l'utente abbia ruotato il dispositivo. Questo è il Main Run Loop ed è associato al Main Thread. Questo run loop è impostato in automatico dalla piattaforma e lo sviluppatore raramente avrà bisogno di modificarlo; invece, quando si crea un thread secondario, in alcuni casi sarà necessario creare ed impostare il run loop relativo. Un run loop viene utilizzato quando occorre che un thread sia "svegliato" da un evento di qualche tipo e durante il resto del tempo stia addormentato, in modo che non consumi risorse. Il run loop relativo ad un thread secondario dovrà essere impostato nel caso in cui esso, anziché eseguire una serie di operazioni e poi terminare, debba restare in esecuzione, monitorando l'arrivo di eventi di qualche tipo. Questi eventi possono essere associati a Timer oppure a segnalazioni asincrone relative alla richiesta di eseguire metodi su quel thread (utilizzando il metodo di `NSObject` `performSelector:onThread:withObject:waitUntilDone:`, alla presenza di messaggi su una `Port` (concetto del sistema operativo per indicare un punto di accesso ad un processo con una coda per memorizzare i messaggi

in ingresso), o in generale ad altri tipi di *input source* che possono anche essere definiti dallo sviluppatore. Il run loop ha perciò associati una serie di *input source* e dei timer; inoltre possiede una serie di stati che servono per decidere quali *input source* monitorare in un certo momento. Infine, al run loop sono associati degli osservatori che possono reagire ad eventuali cambiamenti di stato. Il run loop gira per un certo tempo prestabilito, poi se non ci sono eventi da gestire viene addormentato; dopodiché viene risvegliato quando vi sono eventi associati ai propri *input source*.

Configurando un thread in questo modo è possibile implementare la gestione di particolari eventi che richiedano un'esecuzione urgente, impostando un'apposita *input source* per ricevere gli allarmi. Creando il proprio run loop è possibile schedulare con l'ordine desiderato il controllo e la reazione alle varie sorgenti, ed è inoltre possibile reagire in modo appropriato ai cambi di stato del run loop utilizzando gli osservatori, ad esempio per stabilire, nel momento in cui il run loop sia stato svegliato, quale è effettivamente il motivo del risveglio.

L'inconveniente di questa soluzione è che, mentre sarebbe desiderabile l'utilizzo di un solo canale di comunicazione con il gestore di messaggi, è invece necessario configurare una sorgente apposita, che potrebbe essere associata ad una `port` oppure all'esecuzione di un certo metodo sul thread associato al run loop.

Esiste infine la possibilità di utilizzare una `NSOperationQueue` che permette di impostare la priorità delle `NSOperation` che sono pronte nella coda. Gestendo ogni singolo messaggio in un handler e facendo eseguire una Operation per ogni messaggio, si avrebbe la garanzia del superamento in coda delle operazioni con priorità più alta (ovvero quelle associate agli allarmi) e quindi la messa in esecuzione prioritaria di esse.

Capitolo 3

Un caso di studio

In questo capitolo si analizzerà un caso di studio proveniente da una realtà aziendale. Le attività sono partite da un software già esistente e consolidato su iOS. L'obiettivo era ottenere un porting su Android del software, ma si è rivelata un'occasione interessante per un'analisi comparata tra le due piattaforme, per vedere come sia possibile risolvere certi tipi di problemi, con l'obiettivo finale di ottenere uno stack software in grado di produrre codice applicativo equivalente sui due sistemi. Infatti, Order Sender si presta molto allo scopo, perché oltre alla versione originale attualmente in commercio su App Store e Android Market, esistono anche numerose versioni personalizzate per aziende con esigenze particolari. Alcune personalizzazioni sono già state realizzate ma ce ne sono numerose altre che sono state messe in preventivo; inoltre, potenzialmente la richiesta è inesauribile, poiché ogni azienda potrebbe desiderare la propria variante con le caratteristiche specifiche che riguardano il proprio settore di produzione e i propri dati interni.

L'applicazione è stata analizzata partendo dalla versione originale per iPad. Dopo aver studiato i concetti base della piattaforma iOS, è stato ricavato uno schema dell'App costituito dai bulding block e da esso si è studiato un PIM, *platform-independent model* dal quale ricavare un modello per il progetto su Android. In questo capitolo si riporta la parte platform-independent del progetto, ricavata appunto a partire dall'applicazione originale.

3.1 Order Sender

Order Sender è un applicazione per tablet rivolta agli agenti di commercio che vogliano informatizzare il processo di compilazione, memorizzazione ed invio delle commesse d'ordine. Grazie a questa applicazione l'agente può gestire i propri dati e quelli dei propri mandatari e dei loro clienti, insieme al loro listino prodotti ed eventualmente ad una serie di sconti. La funzionalità principale dell'applicazione è però appunto la gestione della commessa, che può essere compilata grazie a una form e i cui dati possono essere successivamente inviati ad un servizio web che si occupa di produrre il PDF corrispondente e di inviarlo per e-mail al cliente, al mandatario e all'agente stesso in copia. La commessa viene poi memorizzata in un database interno. È possibile inoltre, sempre grazie all'interazione con un servizio web, effettuare l'importazione e l'esportazione dei dati dell'azienda come clienti, fornitori e il relativo listino prodotti. Infine l'applicazione contiene il proprio manuale d'uso.

Le funzionalità fin qui descritte fanno parte della prima versione di Order Sender. In versioni più recenti (ovvero da Order Sender 2.0 in poi) sono state aggiunte nuove funzioni al programma per permettere di sincronizzare l'iPad con dati inseriti tramite un servizio web di nome Order Sender Business. Ciò avviene attraverso un sistema di sottoscrizioni; una sottoscrizione identifica un insieme di dati tramite uno username e una password.

I servizi web citati sono forniti e implementati dalla stessa azienda che produce e commercializza l'applicazione.

Nel seguito si analizzeranno i requisiti dell'applicazione.

3.1.1 Descrizione dei requisiti

Si descriveranno nel seguito le funzionalità richieste al programma.

Compilazione dell'ordine

La prima e più importante funzionalità richiesta è la possibilità di compilare una commessa d'ordine. Il programma dovrà presentare un'interfaccia di sem-

plice utilizzo grazie alla quale creare un ordine associandovi i dati di interesse. Si dovrà anche permettere all'utente di inserire in modo rapido dati già presenti nel sistema attraverso un sistema di autocompletamento. Infine, nella schermata dovrà essere presente un meccanismo per il salvataggio dei dati del cliente o del fornitore, o per l'inserimento di un nuovo cliente o di un nuovo fornitore, qualora l'utente abbia inserito dati nuovi e desideri memorizzarli.

Invio dell'ordine

L'ordine, una volta compilato con almeno il cliente, il fornitore e un prodotto, dovrà essere inviato attraverso una richiesta HTTP con metodo POST contenente tutti i dati a un indirizzo specifico che si occupa di elaborare i dati, creare il PDF e inviarlo per e-mail agli indirizzi specificati richiesti all'utente al momento dell'invio e/o ricavati da dati già presenti.

Gestione dei dati

Una sezione del programma è completamente dedicata alla gestione dei dati. Precisamente, dovranno esserci schermate per permettere l'inserimento, la modifica e la cancellazione di clienti, fornitori, prodotti e sconti. In particolare, i prodotti verrano associati a un certo fornitore, e gli sconti a un fornitore, un cliente ed eventualmente ad un prodotto (se non è presente il prodotto, lo sconto è associato all'intero listino del fornitore per quel cliente).

Dovranno essere presenti anche schermate per la consultazione e la manipolazione dello storico degli ordini: dovrà essere possibile visualizzare i 50 ordini più recenti, tutti gli ordini in ordine di data e dovrà anche essere possibile filtrare l'archivio in base al cliente o al fornitore.

Infine l'agente dovrà poter inserire i propri dati tramite un'apposita sezione.

Importazione

Una sezione del programma è dedicata all'importazione dei dati. Questa procedura prevede l'integrazione dei dati dell'applicazione con altri inviati da un ser-

vizio web. L'utente, dopo essersi registrato con i propri dati al sito www.ordersender.com, dovrà seguire una procedura per caricare i dati su web utilizzando il foglio elettronico e specificando quale tipo di dato vuole importare tra clienti, fornitori e prodotti. Al termine del caricamento del foglio elettronico (che dovrà essere conforme a un certo formato) otterrà un codice di 6 lettere. Dovrà successivamente aprire la sezione di importazione dell'App, inserire username, password e codice ottenuto durante la procedura e selezionare il tipo di dato (nel caso dei prodotti dovrà anche selezionare il fornitore a cui associarli). Al termine del trasferimento troverà nell'applicazione i dati come se li avesse inseriti a mano dalla sezione della gestione, con la possibilità di modificarli a piacimento. Per permettere tutto questo, l'applicazione dovrà collegarsi al server inviando una richiesta HTTP GET con i dati dell'utente e otterrà un file di testo contentente le informazioni codificate in maniera opportuna. In questo file ogni cliente, fornitore o prodotto sarà riportato su una riga diversa, e i campi saranno separati dal doppio pipe. L'applicazione dovrà leggere il file, riconoscere i dati e copiarli sul database interno. Il file di testo risiede sul server, ma dovrà essere cancellato dopo essere stato utilizzato per ragioni di privacy, mandando una apposita richiesta HTTP.

Esportazione

La stessa procedura all'inverso dovrà essere implementata per permettere all'utente di salvare i suoi dati su web tramite foglio elettronico. L'utente sceglierà una categoria tra prodotti, fornitori e clienti, dopodiché dovrà inserire username e password. L'applicazione a questo punto dovrà recuperare i dati dal database, creare un file di testo con la sintassi già descritta per l'importazione e inviarlo al server utilizzando una richiesta HTTP POST con allegato. L'utente, accedendo all'area riservata del sito www.ordersender.com, potrà scaricare o cancellare i file di foglio elettronico creati dal servizio web attraverso l'elaborazione dei dati inviati dall'applicazione.

Sincronizzazione

Una sezione del programma aggiunta in tempi più recenti riguarda la sincronizzazione. Il sistema dovrà essere in grado di memorizzare una serie di *sottoscrizioni*. Le sottoscrizioni sono caratterizzate da un nome, uno username, una password e un codice azienda. Il programma dovrà poi verificare se ci sono aggiornamenti rispetto ai dati già presenti nella base dati con una chiamata HTTP apposita e in caso ci siano (la risposta alla chiamata in questo caso sarà `updateok`, altrimenti `updateno`), scaricarli: utilizzando una richiesta HTTP GET si otterrà un file simile a quello dell'importazione, ma contenente tutte le tipologie di dato (clienti, fornitori, prodotti e anche, in questo caso, sconti), le loro relazioni e l'indicazione riguardo se ogni dato debba essere aggiunto, modificato o eliminato nel database. Questo è ciò che avviene durante una sincronizzazione. I dati associati a una sottoscrizione sono trattati in modo diverso dai dati inseriti a mano o importati dall'utente e devono essere riconoscibili in altre parti del programma. Nella gestione questi dati sono visualizzati ma non modificabili (poiché la sincronizzazione è monodirezionale: il flusso va dal web al programma e non viceversa) e lo stesso vale per la schermata dell'ordine, dove i campi di un cliente o di un fornitore associato a una sottoscrizione sono bloccati a seguito dell'autocompletamento.

Manuale

Una parte del programma infine è dedicata alla visualizzazione del manuale d'uso dell'App, in formato PDF.

3.2 Analisi dei requisiti

3.2.1 Glossario

Segue il glossario dei principali termini utilizzati nella descrizione dei requisiti di Order Sender.

Tabella 3.1: Glossario

Termine	Significato
Agente	l'agente è l'utilizzatore principale di OrderSender. La sua figura professionale è definita come Agente di commercio, ed è un persona soggetta a un contratto di lavoro stabile presso un'azienda senza però essere necessariamente un dipendente. L'azienda pagante è detta mandatario dell'agente, il quale è detto monomandatario o plurimandatario a seconda che rappresenti una o più aziende. L'agente si occupa di recarsi presso le aziende clienti e di fare da tramite tra mandatario e cliente vendendo merce del fornitore. La vendita viene registrata con procedure standard come la compilazione di ordini cartacei.
Fornitore	il fornitore consiste nel mandatario dell'agente. È l'azienda che l'agente rappresenta e per conto della quale effettua le vendite e le registra negli ordini.
Cliente	consiste nell'insieme di dati associati a una azienda, o un privato, che compra presso le aziende mandatarie.
Prodotto	ogni azienda mandataria ha il proprio listino prodotti, ovvero una serie di informazioni associate a ciò che può essere venduto.
Sconto	un fornitore può decidere di concedere ad un cliente una riduzione del prezzo su uno o più articoli.
Ordine	l'ordine è un documento con valore legale prodotto dall'agente e può essere sia in formato cartaceo che digitale. L'ordine consiste nell'attestazione di un acquisto effettuato da un particolare cliente effettuando una scelta sul listino prodotti di un certo fornitore, perciò contiene i dati del fornitore, del cliente, dell'agente e dei prodotti coinvolti nell'acquisto, la loro quantità, il loro prezzo e gli eventuali sconti associati.

Termine	Significato
Sottoscrizione	una sottoscrizione identifica un utente iscritto al servizio web Order Sender Business. Permette una gestione dei dati da web le cui modifiche vengono poi trasferite sul programma attraverso un meccanismo di sincronizzazione.

3.2.2 Casi d'uso

L'applicazione dovrà perciò essere in grado di assolvere i seguenti compiti:

- Memorizzare e gestire su supporto una certa quantità di dati.
- Interagire con servizi web tramite richieste HTTP.
- Interpretare una semplice sintassi per la decodifica di dati.
- Presentare all'utente un'interfaccia intuitiva e di semplice utilizzo.

La cosa può essere schematizzata utilizzando i casi d'uso. In Figura 3.1 è riportato il diagramma UML dei casi d'uso. Nel seguito verranno illustrati tre scenari tipici di utilizzo dell'App. Per semplicità di rappresentazione si sceglie di circoscrivere i casi d'uso e la restante trattazione ai casi specifici descritti nei tre scenari.

Primo caso d'uso

Nome: UC1 - Compilazione ordine.

Descrizione: Permette di inserire i dati relativi a un ordine.

Attori: Agente.

Precondizioni: Nessuna.

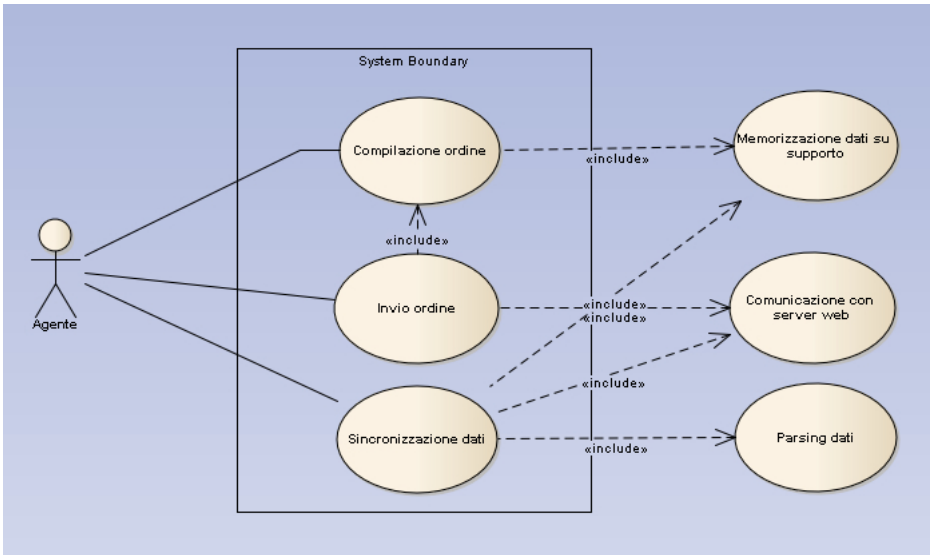


Figura 3.1: Schema UML dei casi d'uso

Scenario principale: L'applicazione presenta una form di compilazione dell'ordine di semplice utilizzo. Grazie alla form è possibile recuperare dati inseriti a mano tramite la gestione, tramite l'importazione, o tramite sincronizzazione. È anche possibile utilizzare dati che non fanno riferimento a ciò che è stato precedentemente inserito. Questi dati sono eventualmente salvati su supporto sempre grazie alla form, per poter essere recuperabili in un secondo momento da gestione, da un altro ordine, o per essere esportati. Dati che non fanno parte di sottoscrizioni ma già presenti su supporto sono eventualmente modificati e salvati permanentemente su supporto grazie alla form. Al termine di ogni aggiornamento della form e all'uscita della schermata l'ordine è automaticamente salvato su supporto in modo che sia recuperabile in un secondo momento.

Scenari alternativi: Nel caso in cui l'App si blocchi, abbia un arresto anomalo o il tablet si spenga, l'ordine è recuperabile da supporto secondo l'ultima configurazione disponibile.

Postcondizioni: L'ordine è salvato su supporto come "non inviato". Alcuni dati

precedentemente salvati su supporto potrebbero essere stati modificati ed altri aggiunti.

Secondo caso d'uso

Nome: UC2 - Invio ordine.

Descrizione: Invio dell'ordine precedentemente compilato via HTTP al server web.

Attori: Agente.

Precondizioni: Compilazione dell'ordine con almeno fornitore, cliente e un prodotto; connessione a internet disponibile.

Scenario principale: Alla pressione del tasto "Invia" sulla form da parte dell'Agente, l'ordine viene inserito nel body di una richiesta HTTP di tipo POST. I campi vengono codificati inserendoli nel body utilizzando coppie chiave-valore secondo un formato riconoscibile dal server. Il server invia una risposta HTTP di tipo `200 OK` contenente la stringa `ok` nel body per indicare che ha correttamente ricevuto la richiesta e che ha inoltrato l'ordine via e-mail ai destinatari specificati.

Scenari alternativi Se non si riceve risposta dal server entro 10 secondi la richiesta viene annullata. Se la risposta HTTP non è `200 OK` o se il body della risposta del server contiene qualcosa di diverso dalla stringa `ok`, l'invio viene considerato fallito. In entrambi i casi l'ordine viene considerato "non inviato" e viene visualizzata una finestra d'errore.

Postcondizioni: L'ordine è salvato su supporto come "inviato".

Terzo caso d'uso

Nome: UC3 - Sincronizzazione dati.

Descrizione: Aggiornamento dei dati inseriti via web. Prevede inserimenti, cancellazioni e modifiche dei dati presenti su supporto associati a una specifica Sottoscrizione.

Attori: Agente.

Precondizioni: Inserimento di una sottoscrizione valida (cioé precedentemente riconosciuta dal servizio web), composta da nome, username, password e codice azienda; connessione a internet disponibile.

Scenario principale : Alla pressione del tasto "Sincronizza" relativo alla sottoscrizione, raggiungibile da due diverse form, verrà inviata una richiesta HTTP, detta *check*, con metodo GET all'indirizzo http://business.ordersender.com/sottoscrizione/check. La risposta indica la presenza o meno di aggiornamenti. Se la risposta HTTP è di tipo `200 OK` e il body è costituito dalla stringa `updateok`, si procede con una seconda chiamata HTTP, detta *sync*, all'indirizzo

```
 http://business.ordersender.com/sottoscrizione/sync/
codice/<codiceazienda>/username/<username>/
password/<password>/lastSync/<dataoraultimasincronizzazione>/
udid/<codicedispositivo>
```

indicando anche la data e l'ora di ultima sincronizzazione secondo il formato `yyyy-MM-dd HH:mm:ss` e il codice univoco che identifica il dispositivo dove l'App gira. Nel caso in cui la richiesta sia andata a buon fine il Body della risposta conterrà un insieme di righe che occorrerà manipolare per ottenere i dati relativi all'importazione. Ogni riga specifica un comando (tra inserimento, aggiornamento e cancellazione) e il dato relativo su cui effettuare il comando. Nelle righe i campi sono separati dal doppio pipe e l'ordine è importante. La riga sarà del tipo:

```
<comando>||<tipo>||<dati>...||<codAzienda>||<idAzienda>
```

`comando` può essere `ins` per inserimento, `upd` per aggiornamento, `del` per cancellazione. `tipo` può essere `fornitore`, `cliente`, `prodotto` o `sconto`. I `dati` variano in numero a seconda del `tipo`. `codAzienda` è l'identificativo dell'azienda che caratterizza anche la sottoscrizione e `idAzienda` è un codice numerico univoco che identifica l'id del dato considerato all'interno del server web. Un esempio di body ottenuto dal server è il seguente:

```
ins||fornitore||nomeF||respF||000111222||333444555||
667788||fornitore@fornitore.it||indirizzoF||cittaF||
azien111||10
ins||cliente||nomeC||respC||00000||11111||222111000||
555444333||887766||cittaC||indirizzoC||noteC||
cliente@cliente.it||55555||CC||99999||azien111||20
upd||prodotto||codP||descP||1||2||azien111||30
del||sconto||1||2||3||15||10||5||azien111||40
```

Occorrerà valutare la sintassi di ogni riga e, nel caso in cui sia valida, ottenere il comando e il dato di cui è rappresentazione. A quel punto il comando dovrà essere eseguito, producendo un inserimento, una modifica o una cancellazione nel database. In questo caso, la prima riga indica di inserire un fornitore di nome nomeF, responsabile respF, telefono 000111222, cellulare 333444555, fax 667788, e-mail fornitore@fornitore.it, indirizzo indirizzoF, città cittaF, codAzienda azien111 e idAzienda 10. La seconda riga contiene un comando di inserimento di un cliente di nome nomeC, responsabile respC, partita Iva 0000, codice fiscale 11111, telefono 222111000, cellulare 555444333, fax 887766, città cittaC, indirizzo indirizzoC, note noteC, e-mail cliente@cliente.it, cap 55555, provincia CC, iban 99999, codice azienda azien111 e idAzienda 20. La terza riga indica di aggiornare il prodotto di idAzienda 30 con i dati codice codP, descrizione descP, quantità minima 1 e prezzo 2. Il codice azienda del prodotto è azien111. La quarta ed ultima riga infine indica di cancellare lo sconto di idAzienda 40 il cui idAzienda del fornitore sia 1, idAzienda del cliente sia 2 e idAzienda del prodotto sia

3. Lo sconto ha inoltre le percentuali di sconto sconto1 pari a 15, sconto 2 pari a 10, sconto3 pari a 5. Il codice azienda è azien111.

Scenari alternativi: In caso di connessione a internet assente viene visualizzato un messaggio d'errore. Se la risposta a una delle due richieste HTTP ha un codice diverso da 200 OK allora c'è stato un problema di comunicazione col server; pertanto l'importazione viene dichiarata fallita. Se non sono presenti aggiornamenti (la risposta HTTP alla chiamata *check* è `200 OK` e il body è composto dalla stringa `updateno`) viene segnalato all'utente che non sono presenti aggiornamenti e la sincronizzazione termina.

Se il body è costituito invece dalla stringa `pwdfail` significa che le credenziali immesse dall'agente sono sbagliate. Se la stringa invece è `deleted` significa che la sottoscrizione non esiste più sul server; la cosa provoca la cancellazione della sottoscrizione e di tutti i dati relativi ad essa. Se il body della richiesta HTTP di *sync*, di tipo `200 OK`, inizia con `***`, allora il server ha restituito un errore, che può essere:

```
***La sottoscrizione non è più attiva oppure il tuo account
non è più valido.
```

oppure

```
*** Password errata
```

. Tutte le situazioni di errore e di malfunzionamento sono notificate all'utente con appositi messaggi.

Postcondizioni: In caso di successo, i dati vengono salvati su supporto e viene salvata come data di ultima sincronizzazione l'orario corrente fornito dal dispositivo.

3.2.3 Modello del dominio

In questa sezione si riporta il modello del dominio. Si è deciso di dividere il modello del dominio in tre parti: il modello del dominio dei dati trattati e memorizzati all'interno dell'applicazione, il modello del dominio dei dati utili alle operazioni da svolgere e il modello del dominio che riguarda la fase operativa della sincronizzazione.

Modello del dominio dei dati

Il modello del dominio dei dati, mostrato in Figura 3.2, parte definendo il modello di un *Elemento* generico dal quale ereditano tutti gli altri modelli di dati, ovvero:

Agente ha un Nome, un Cognome, Telefono, Fax, Cellulare, Email, Indirizzo, Città. Rappresenta l'unico utilizzatore dell'applicazione.

Cliente caratterizzato almeno da una Ragione Sociale. Altri dati aggiuntivi sono: Responsabile, Partita Iva, Codice fiscale, Iban, Telefono, Fax, Cellulare, Email, Indirizzo, Città, Provincia, CAP, Note. Può essere associato ad una Sottoscrizione (quindi è un ElementoSincr) ed è un dato importabile/esportabile (quindi un ElementoImp/Exp).

Fornitore caratterizzato da un Nome e un'Email. Altri campi sono: Responsabile, Telefono, Fax, Cellulare, Città, Indirizzo. Può essere associato ad una Sottoscrizione ed è un dato importabile/esportabile.

Prodotto caratterizzato da un Codice, eventualmente da una Descrizione, un Prezzo e una Quantità Minima. Può essere associato ad una Sottoscrizione ed è un dato importabile/esportabile.

Sconto appartente a un Fornitore, a un Cliente e opzionalmente ad un Prodotto. Ha inoltre associate tre quantità numeriche che identificano tre diversi livelli di sconto. Può essere associato ad una Sottoscrizione.

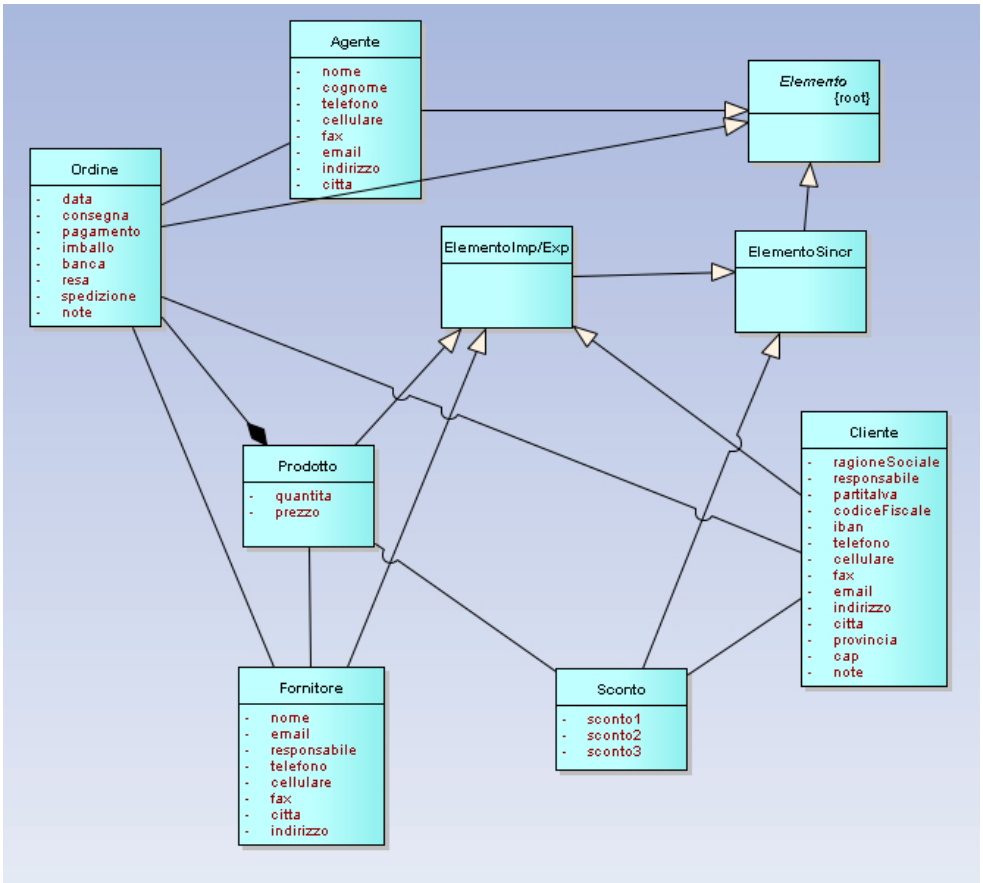


Figura 3.2: Schema UML del modello del dominio dei dati

Ordine ha associato un Cliente, un Fornitore ed almeno un Prodotto ed è effettuato in una certa Data. Ogni prodotto può avere uno Sconto. L'ordine è inoltre associato a un agente. Può contenere altri campi quali Consegna, Pagamento, Banca, Imballo, Resa, Pagamento e Note.

Modello del dominio delle operazioni

Si illustrerà ora il modello del dominio dei dati necessari alle operazioni descritte nei requisiti. Il modello del dominio delle operazioni è mostrato in Figura 3.3:

Utente identifica un utente qualunque, ha uno Username e una Password.

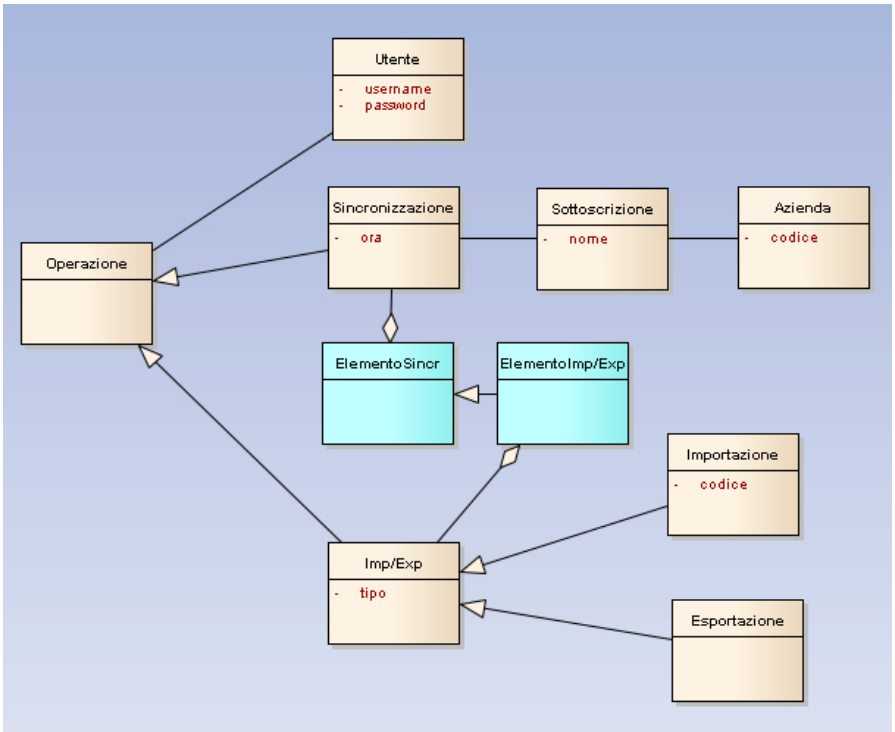


Figura 3.3: Schema UML del modello del dominio delle operazioni

Azienda possiede un identificativo univoco nel sistema. L'azienda identifica il sottoinsieme di dati che occorre sincronizzare.

Importazione è associata ad un Utente e possiede inoltre un Codice e un insieme di dati. I dati associati all'importazione possono essere Fornitori, Clienti o Prodotti.

Esportazione ha un Utente e un insieme di dati (tra Clienti, Fornitori e Prodotti).

Sottoscrizione è caratterizzata da un Nome ed è associata ad un Utente. Inoltre possiede un Codice Azienda. Ha un insieme di dati tra Clienti, Fornitori, Prodotti e Sconti.

Sincronizzazione è associata ad una Sottoscrizione. Possiede il proprio orario di esecuzione e un insieme di dati di vario tipo con le operazioni da eseguire.

Modello del dominio della sincronizzazione

In questa sezione si riporta il modello del dominio dei dati che riguardano la sincronizzazione.

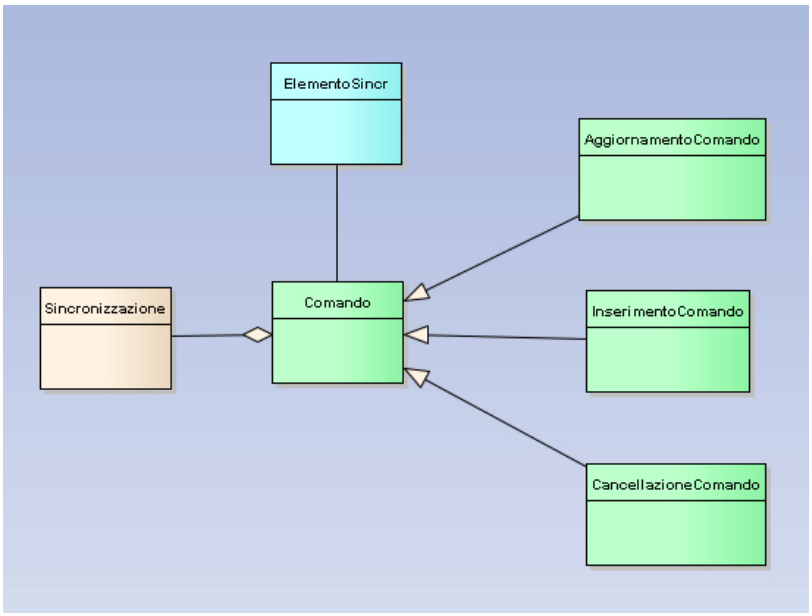


Figura 3.4: Schema UML del modello del dominio della sincronizzazione

ElementoSottoscrizione ogni sincronizzazione ha associati una serie di elementi da cancellare, aggiornare o inserire.

Comando dall'analisi del file ricevuto si otterranno una serie di comandi. Ad ogni comando è associato il dato su cui esso deve essere applicato.

CancellazioneComando un comando che prevece la cancellazione dell'elemento associato.

InserimentoComando un comando che prevede l'inserimento dell'elemento associato.

AggiornamentoComando un comando che aggiorna l'elemento associato.

3.2.4 Requisiti non funzionali

La scelta di implementare l'applicazione su mobile introduce una serie di requisiti non funzionali. Il più immediato è l'*usabilità*. Infatti, occorre che il sistema sia semplice e intuitivo, poiché il target di utilizzatori dell'App (agenti di commercio) non necessariamente saranno esperti di computer; l'introduzione dei tablet, sia per via dell'interfaccia touch sia a causa della possibilità di mostrare una sola schermata per volta, ha permesso e sta permettendo un grande bacino di utenza per via della semplicità e dell'intuitività del sistema stesso. Occorre pertanto che il sistema sia conforme a questo principio, sia nel presentare le interfacce all'utente, sia nel funzionamento complessivo.

L'*efficienza* è un altro requisito importante. Per prima cosa perché, sia per requisiti hardware di memoria e processore (che però sono sempre più potenti), sia, e soprattutto, per risparmiare batteria, occorre non sprecare risorse e lasciare l'applicazione nello stato *idle* il più possibile. In secondo luogo, è necessario che i tempi di risposta dell'applicazione siano rapidi e che i task pesanti siano distribuiti in modo intelligente tra i thread. Infatti, entrambe le piattaforme sono dotate di un sistema che impedisce alle applicazioni di occupare il main thread, quello che si occupa dell'interazione con l'utente, per più di qualche secondo; questo proprio perché è possibile visualizzare una schermata per volta. L'applicazione che occupi lo schermo per più di 10-15 secondi, intasando il main thread e quindi impedendo all'utente di interagire con il device, verrà terminata. Su iOS l'App semplicemente viene spenta e fatta sparire forzatamente dalla schermata; su Android l'utente viene anche avvisato di un errore di tipo ANR, Application Not Responding.

La *scalabilità* è un requisito interessante poiché occorre considerare che il progetto dell'applicazione potrà essere utilizzato come base per altre applicazioni simili. Queste applicazioni sono personalizzazioni per aziende che distribuiscono l'App internamente ai propri rappresentanti. Nei casi più semplici le personalizzazioni prevedono semplicemente la modifica di alcune entità del modello del dominio e una sincronizzazione mirata per quei dati, ma in altri casi può voler dire l'aggiunta di numerose funzionalità, come, ad esempio, la possibilità di sfogliare un catalogo direttamente all'interno dell'App e da esso aggiungere articoli ad

un ordine. Questo requisito porta alla naturale conclusione che siamo sulla strada giusta, nell'intenzione di implementare un generatore di codice a partire da un linguaggio custom derivato dall'esperienza in azienda. In questo senso la soluzione proposta deve anche soddisfare il requisito di *generalità*.

L'App ha anche un costo elevato di *supportabilità*, poiché essendo distribuita su App Store e Android Market senza nessun controllo sugli eventuali compratori, non tutti saranno in grado di utilizzare tutte le funzionalità dell'App appieno al primo tentativo. Esiste un indirizzo e-mail ove poter scrivere per problemi di assistenza. L'app deve essere perciò, come già detto, il più usabile e semplice possibile. Inoltre anche il mantenimento dell'App è un task da non trascurare, poiché, essendo distribuita e utilizzata da un numero sempre crescente di utenti, nuove esigenze e problemi (bug, criticità dovute ad un eventuale cattivo o maldestro utilizzo) sorgono continuamente. Le soluzioni necessitano di essere implementate nel più breve tempo possibile e pubblicate utilizzando il meccanismo degli aggiornamenti che la piattaforma di distribuzione prevede. Inoltre occorre tenere conto che mentre su Android Market gli aggiornamenti vengono pubblicati immediatamente dopo aver caricato con l'apposita procedura il nuovo *apk* dell'applicazione, per pubblicare un aggiornamento su App Store occorre aspettare la fine della procedura di approvazione che porta a un'attesa di circa 5-7 giorni tra il caricamento su App Store e l'effettiva pubblicazione. Infine, ogni applicazione pubblicata su App Store ha associato un certificato di distribuzione (oltre che un certificato per lo sviluppo) che va rinnovato ogni anno, pena la cancellazione delle App associate su App Store. Su Android invece l'applicazione resta valida fino a che lo è il certificato che si è usato per firmare l'*apk*. Una firma derivata da un certificato autoprodotto, ad esempio con Java Keytool, ha validità fino al 2031. Ovviamente se l'applicazione fosse critica e necessitasse di un meccanismo di certificazione più sofisticato i tempi sarebbero ben più brevi, ma su quale certificato appoggiarsi resta comunque una scelta dello sviluppatore.

L'applicazione infine deve avere forti requisiti di *interoperabilità* con il server web con cui deve interfacciarsi, per rendere la comunicazione con un server che può cambiare per evoluzioni della tecnologia web o semplicemente per una

modifica del servizio il più agevole e modulabile possibile.

3.3 Analisi del problema

In questa sezione sarà illustrata l'analisi del problema di Order Sender. In questa analisi si cercherà di astrarre il più possibile dalle tecnologie realizzative e ci si focalizzerà sul problema così come è posto, anziché sulla soluzione.

L'analisi parte formalizzando le entità che fanno parte del problema e le loro interazioni. Si prosegue con la raffinazione dei tre casi d'uso principali, che implicano altrettanti scenari. Si è scelto di utilizzare dapprima il pattern BCE, *Bounday-Control-Entity*, per delineare i componenti e un primo abbozzo delle interazioni per le tre realizzazioni dei casi d'uso. Seguirà una raffinazione dell'aspetto comportamentale del sistema nei tre casi grazie agli *state diagrams*, ed infine verrà utilizzato *Contact* per formalizzare le interazioni tra i componenti principali.

3.3.1 Entità principali del problema

Si andranno ad analizzare le interazioni tra l'App da implementare e l'ambiente esterno. Le due entità principali coinvolte nel problema sono l'App e il server web. Esse comunicano attraverso l'HTTP per vincoli del problema, ma si decide di astrarre dalla specifica implementativa per rendere il progetto più generale possibile e flessibile a cambiamenti del futuro, quali cambi di tecnologia di appoggio, di server, ecc.

Grazie a Contact si mostra l'interazione tra i due componenti. Essi sono formalizzati come due `subject`; le interazioni tra i due avvengono in due dei tre scenari presentati, ovvero per l'invio dell'ordine e per la sincronizzazione dei dati.

Nel primo caso, ovvero nell'invio dell'ordine, l'App invia al server una `request` che contiene i dati dell'ordine. Il server farà la `grant` del messaggio rispondendo con il risultato dell'invio, dopo aver generato il pdf corrispondente ai dati dell'ordine ed aver inviato la e-mail ai destinatari specificati nel messaggio. La specifica Contact è la seguente:

```
ContactSystem InvioSystem;
subject server;
subject app;
Request sendOrderRequest;

app demand sendOrderRequest to server;
server grant sendOrderRequest;
```

Nel caso della sincronizzazione dei dati, l'app invia una `request` al server per verificare se vi sono aggiornamenti riguardo una specifica sottoscrizione. Il server invierà, come risposta, un messaggio in cui indicherà se sono presenti aggiornamenti o meno. Se ci sono aggiornamenti, l'app invierà un'altra `request` con gli stessi dati ma per ottenere i dati relativi alla sincronizzazione da eseguire. La specifica Contact sarà:

```
ContactSystem SincronizzazioneSystem;
subject server;
subject app;
Request checkRequest;
Request syncRequest;

app demand checkRequest to server;
server grant checkRequest;

app demand syncRequest to server;
server grant syncRequest;
```

3.3.2 Raffinazione dei casi d'uso

Si andranno ora ad analizzare struttura, interazione e comportamento delle varie parti dell'App al fine di ottenere l'architettura logica dell'analisi del problema. Ciò verrà ottenuto attraverso un progressivo raffinamento dei casi d'uso, che si concentrano sulle funzionalità principali offerte dall'App e visibili al-

l'utente esteno. Perciò in questa fase vogliamo analizzare l'architettura *interna* all'applicazione.

Compilazione dell'ordine

Si schematizza in Figura 3.5 la struttura, secondo il pattern BCE, della parte che riguarda la compilazione dell'ordine. Come si può vedere, le Entity sono composte dalla Entity che corrisponde all'Ordine che si sta compilando. Il Boundary è la form con cui l'agente interagisce inserendo i dati. Infine, tra i due vi è un Controller, entità di tipo Control, che si occupa di mediare tra la form e il dato Ordine aggiornandolo in base alle informazioni che riceve dall'utente.

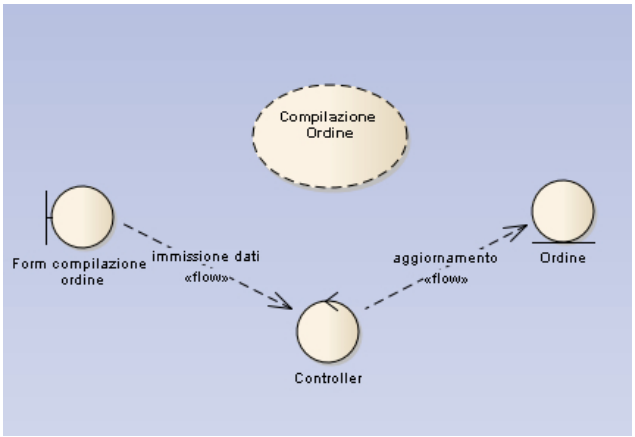


Figura 3.5: Schema UML dei componenti della compilazione di un ordine secondo il pattern BCE

Si vanno ora a definire le interazioni tra questi moduli utilizzando Contact. In questa fase, possiamo considerare il Control e la Entity Ordine come un solo `subject`: il Controller aggiornerà l'Ordine direttamente a mano a mano che i dati giungono dalla form, con, banalmente, un'interazione a method-call. L'interfaccia e il Controller, invece, sono due `subject` distinti e comunicano attraverso la generazione di eventi nel momento in cui nuovi dati sono disponibili e devono essere salvati. La specifica Contact sarà allora:

```
subject controller;
```

```
subject formOrdine;
event datoOrdineDisponibile;

formOrdine raise datoOrdineDisponibile;
controller perceive datoOrdineDisponibile;
```

Si riporta infine in Figura 3.6 lo *state diagram* del comportamento del sistema durante la compilazione di un ordine.

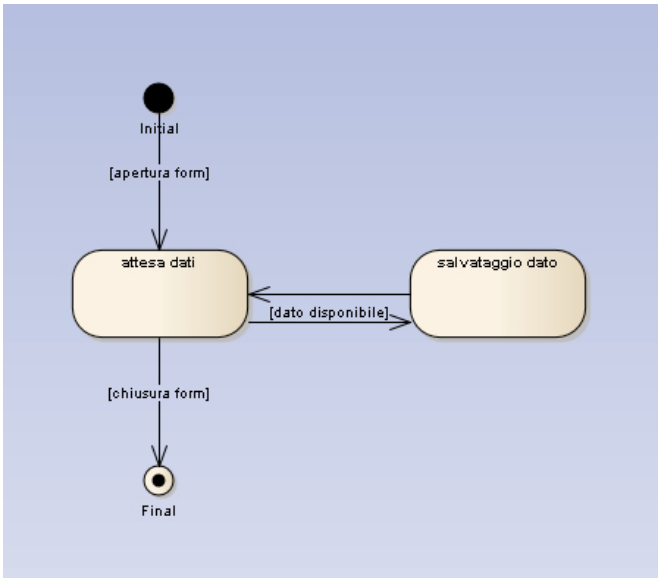


Figura 3.6: State Diagram della compilazione di un ordine

Invio dell'ordine

Si riporta in Figura 3.7 la struttura, secondo il pattern BCE, della parte di invio dell'ordine. L'unica Entity è quella che corrisponde all'Ordine da inviare. Il Boundary è composto da due parti: la form da cui l'agente dà il comando di invio premendo l'apposito pulsante e il gestore delle comunicazioni che manda le richieste al server. In esso è racchiuso il meccanismo di interazione con il server precedentemente illustrato. Di nuovo, a mediare tra Boundary e Entity vi è un

Controller, entità di tipo Control, che aggiorna l'Ordine segnandolo come *inviato* se l'operazione termina correttamente.

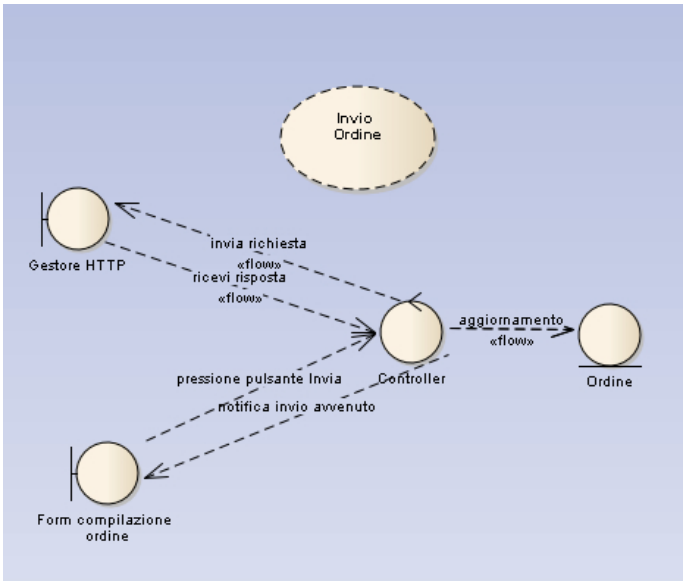


Figura 3.7: Schema UML dei componenti dell'invio di un ordine secondo il pattern BCE

L'interazione tra questi componenti è definita nel seguente modo. Consideriamo nuovamente Controller e Ordine come un solo `subject` poiché Ordine sarà semplicemente un oggetto di cui il Controller potrà disporre. Il GestoreHTTP invece sarà un `subject` a sé e lo stesso vale per l'interfaccia. Nel momento in cui l'utente abbia terminato l'inserimento dei dati e voglia inviare l'ordine, premerà il pulsante Invia della form; in questo caso il subject corrispondente emetterà un evento che verrà captato dal Controller. Il Controller verificherà se l'ordine è valido, dopodiché invierà al `subject` GestoreHTTP una request con i dati dell'ordine e attenderà una response. In base alla `response`, invierà infine un `dispatch` alla form per informare l'utente del risultato dell'invio. Quindi la specifica Contact sarà:

```
subject controller;
subject formOrdine;
```

```
subject gestoreHTTP;
event inviaOrdineComando;
request invioOrdine;
dispatch risultatoInvio;

formOrdine raise inviaOrdineComando;
controller perceive inviaOrdineComando;

controller demand invioOrdine to gestoreHTTP;
gestoreHTTP grant invioOrdine;

controller forward risultatoInvio to formOrdine;
formOrdine serve risultatoInvio;
```

In Figura 3.8 si riporta il diagramma degli stati dell'invio dell'ordine.

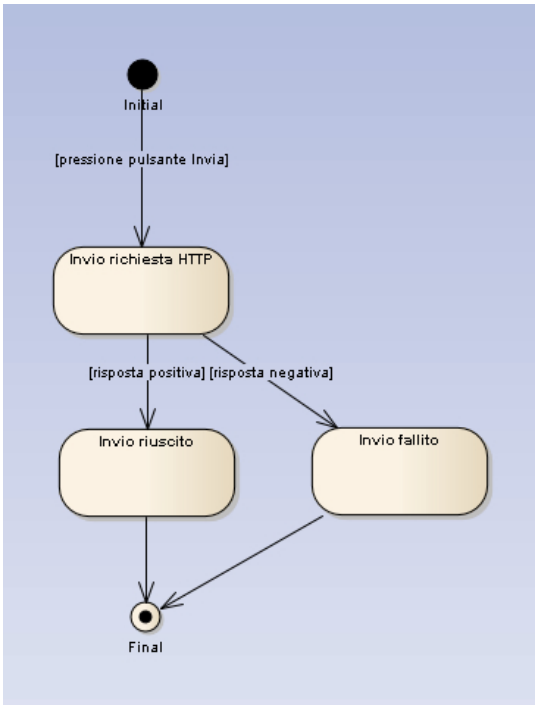


Figura 3.8: State Diagram dell'invio di un ordine

Sincronizzazione dei dati

In Figura 3.9 si può vedere la struttura, secondo il pattern BCE, della sezione dell'App dedicata allo scenario della sincronizzazione dei dati. Le Entity coinvolte sono quelle relative alla sottoscrizione per la quale si stanno cercando gli aggiornamenti, alla sincronizzazione che si eseguirà in caso ci siano effettivamente dati da scaricare e una Entity rappresentativa del tipo di dato che verrà scaricato e introdotto nel programma attraverso il salvataggio su supporto, ovvero la Entity ElementoSincr, che può essere un Fornitore, un Cliente, un Prodotto o uno Sconto. Il boundary è composto nuovamente dalle due parti, una per interagire con l'utente, l'altra per interagire con il server Web. Si trova ancora una volta il Controller per mediare tra le parti, ma le entità Control questa volta sono tre, ovvero un Parser, necessario per interpretare i dati inviati dal server e trasformarli in Entity di tipo Comando, e un Esecutore di comandi, che dato un Comando estrapola il dato contenuto al suo interno ed esegue l'operazione associata sulle Entity ElementoSincr.

L'interazione tra questi componenti è definita circa nello stesso modo degli altri due casi. I `subject` saranno 3, ovvero i due boundary gestoreHTTP e formSincronizzazione e il Controller. Il resto dei componenti vengono assimilati al Controller che interagisce con essi a method call. La form di sincronizzazione emette un evento nel momento in cui l'agente voglia dare il via alla sincronizzazione; il Controller percepisce l'evento e dà il via al procedimento, mandando innanzitutto una request al gestoreHTTP perché esso contatti il server e recuperi i dati; dopodiché, in base alla risposta, eseguirà la procedura interagendo con gli altri oggetti. A mano a mano che la sincronizzazione procede, sarà necessario che l'interfaccia venga aggiornata; la cosa sarà effettuata tramite una serie di `dispatch` che il Controller invierà alla form di sincronizzazione. La specifica Contact che si ottiene perciò è:

```
subject controller;
subject formSincronizzazione;
subject gestoreHTTP;
event sincronizza;
```

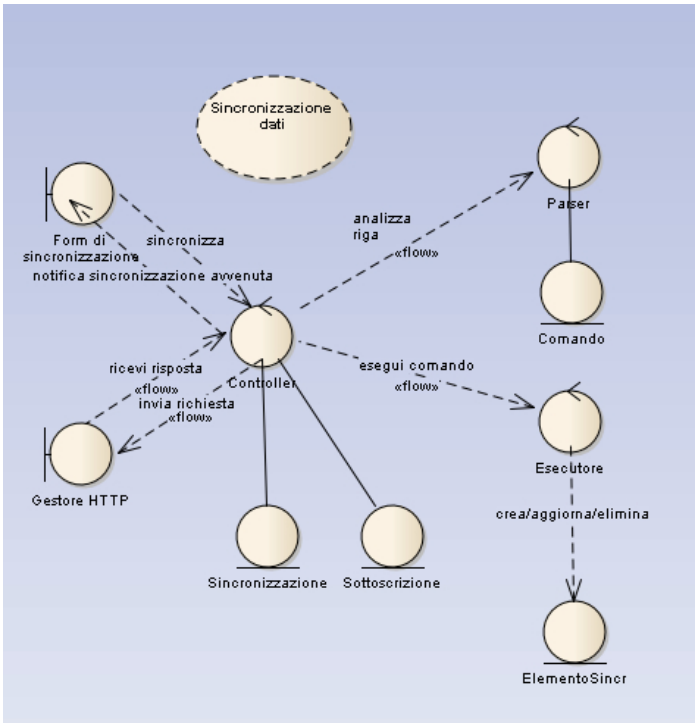


Figura 3.9: Schema UML dei componenti della sincronizzazione secondo il pattern BCE

```
request invioSincronizzazione;
dispatch risultatoIntermedioSincronizzazione;
dispatch risultatoSincronizzazione;

formOrdine raise sincronizza;
controller perceive sincronizza;

controller demand invioSincronizzazione to gestoreHTTP;
gestoreHTTP grant invioSincronizzazione;

controller forward risultatoIntermedioSincronizzazione to
formSincronizzazione;
formSincronizzazione serve risultatoIntermedioSincronizzazione;
```

```
controller forward risultatoSincronizzazione to
formSincronizzazione;
formSincronizzazione serve risultatoSincronizzazione;
```

Si riporta in Figura 3.10 lo *state diagram* della procedura di sincronizzazione.

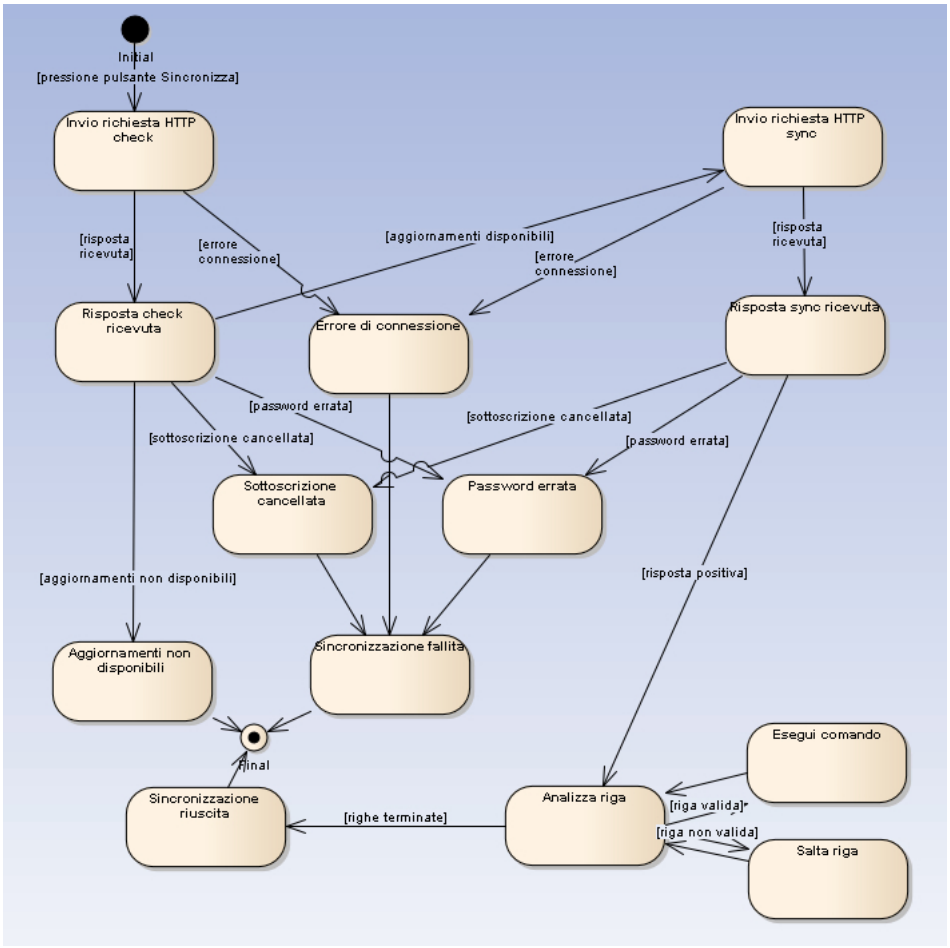


Figura 3.10: State Diagram della sincronizzazione

3.3.3 Architettura logica

Nell'architettura logica troveremo perciò i seguenti moduli, che possiamo considerare come dei Plain Old Object: un insieme di Entity che rappresentano i dati di interesse, quali Ordine, ElementoSincr (e, implicitamente, i 4 tipi di oggetti che ereditano da esso), Sottoscrizione, Sincronizzazione; un GestoreHttp in grado di spedire richieste HTTP e di reagire in modo coerente alla risposta e a problemi di rete; due moduli che interagiscono con l'agente, uno per fargli compilare l'ordine e per inviarlo e una per sincronizzare i dati; un Parser che comprenda la sintassi dei dati che giungono da web e da Stringhe li trasformi in appositi oggetti manipolabili; un Esecutore che sia in grado di eseguire i comandi generati dalla procedura di sincronizzazione; un Controller che si occupi di mediare tra questi elementi. In Figura 3.11 è riportata l'architettura logica secondo il pattern BCE.

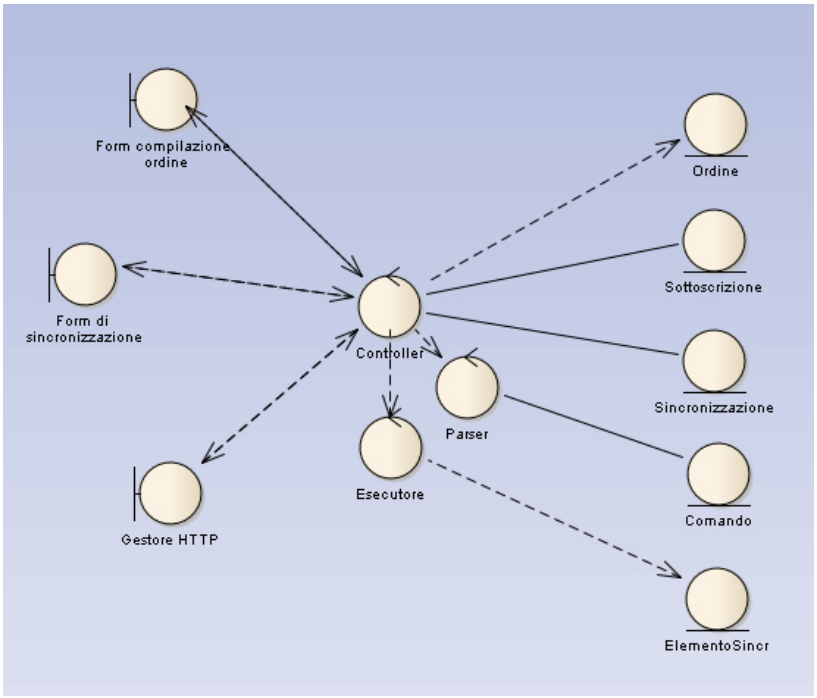


Figura 3.11: Architettura logica dell'analisi secondo il pattern BCE

3.4 Progetto platform-independent

In questa sezione si illustrerà la parte di progettazione platform-independent di Order Sender. Il progetto segue l'analisi del problema estendola e focalizzandosi sulla soluzione ma continuando ad astrarre il più possibile dalle tecnologie realizzative. I grafi (soprattutto quelli di struttura e interazione) vengono estesi grazie ad una serie di considerazioni:

- È necessario salvare i dati su supporto perché siano fruibili in tutta l'applicazione e vengano mantenuti nel tempo; anche in caso di shutdown imprevisto dell'applicazione i dati devono comunque essere memorizzati in modo da poter essere recuperabili in un secondo momento.

- La scelta più intelligente per la memorizzazione dei dati è ovviamente l'utilizzo di un database. In questa fase però si vuole ancora astrarre dal tipo di supporto utilizzato, perché i dati potrebbero essere salvati su file (in caso non si disponga di un database, o ne risulti troppo pesante l'utilizzo, o per qualsiasi altro motivo) o, cosa tipica e interessante in questo dominio applicativo, si voglia utilizzare il *Cloud* per il salvataggio dei dati importanti.

- È possibile affrontare questo problema scegliendo di arricchire l'interfaccia delle Entity utilizzando il pattern *Decorator*, in modo che esse forniscano un metodo per salvare su supporto i propri dati. Questo però mette in discussione il pattern *BCE* che si è scelto di utilizzare finora, andando a inquinare le *Entity* con funzionalità di tipo *Control*.

- Verrà introdotto quindi nell'architettura logica un ulteriore modulo di tipo Control, che sarà chiamato GestoreDati. Il modulo permetterà di incapsulare la logica di salvataggio e quindi di astrarre dallo specifico supporto.

- Ogni volta che sarà necessario salvare i dati rendendoli permanenti, quindi passando dalla memoria virtuale alla memoria fisica, il passaggio per questo modulo sarà obbligato.

- il Controller e l'Esecutore dovranno interagire con questo modulo passandogli i dati da salvare.

Alla luce di queste considerazioni si riporta in Figura 3.12 l'architettura logica di progetto (platform-independent).

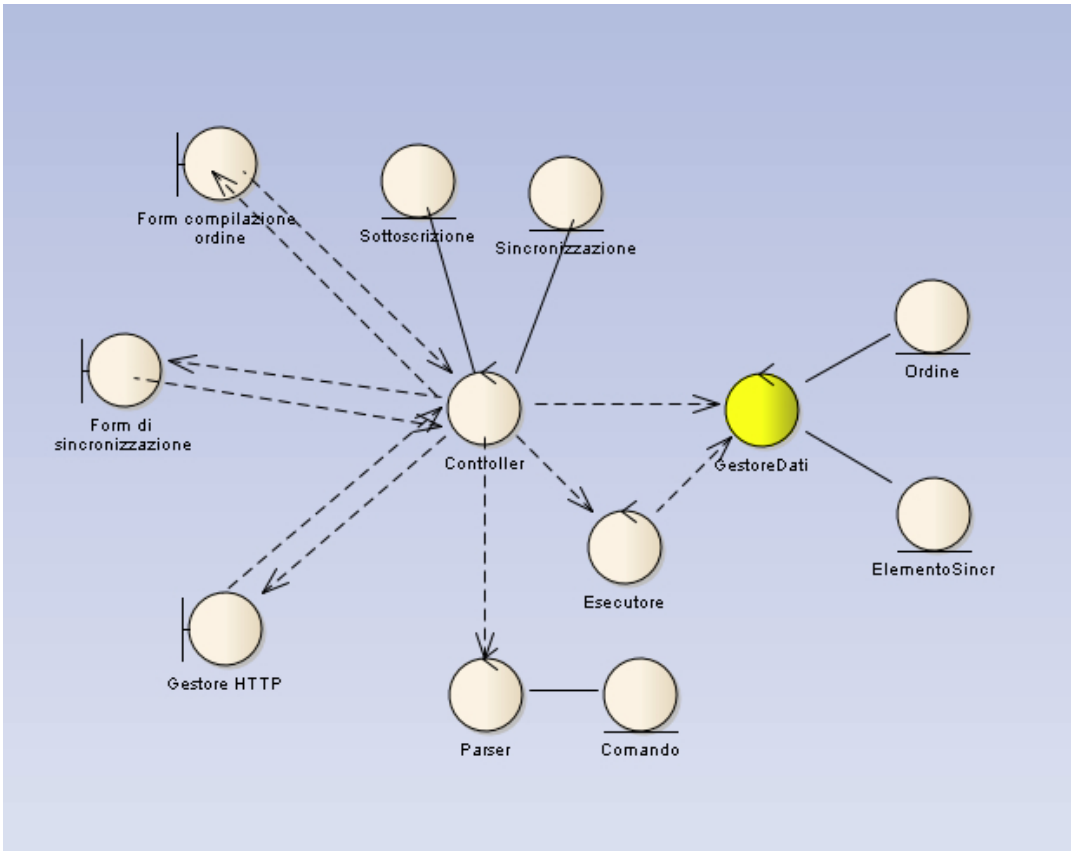


Figura 3.12: Architettura logica del progetto platform-independent secondo il pattern BCE

Capitolo 4

Order Sender su mobile

In questo capitolo si andrà ad analizzare il caso di studio sulle due piattaforme. Come detto in Sezione 1.5, Order Sender è un progetto nato inizialmente su iOS come App per iPad, con l'ausilio di alcuni servizi web creati ad hoc dall'azienda, e successivamente è stata presa la decisione di realizzarne il corrispettivo su Android. Lo sviluppo ha preso dapprima la via del banale *porting* su Android sulla base del software già esistente, per ragioni pratiche sia di tempo, sia di conformità dei due software; successivamente si è sentita la necessità di formalizzare e documentare quanto era stato fatto, producendo ciò che è riportato nel Capitolo 3. Ciò innazitutto perché era necessaria l'implementazione dell'estensione della sincronizzazione anche su Android; poi per un futuro utilizzo su altre piattaforme (Windows Phone, ad esempio), o per l'implementazione di personalizzazioni dell'App stessa, necessità effettivamente presente nell'azienda.

Si analizzerà dapprima l'applicazione di partenza, mostrandone la struttura e vedendo come le varie parti dell'applicazione, formalizzate secondo il metamodello della piattaforma, trovino il corrispettivo nell'architettura logica di progetto platfom-independent ottenuta nel Capitolo 3 come conseguenza dell'analisi del caso di studio. Con questo confronto incrociato sarà possibile ricavare l'architettura logica di progetto platform-dependent su iOS. Poi verrà analizzata la versione per Android partendo invece direttamente dall'architettura di progetto platform-independent, e con le conoscenze di Android sarà ottenuta l'architettura logica di

progetto platform-dependent per questa piattaforma.

4.1 Order Sender per iPad

Order Sender nasce come applicazione per iPad. In questo paragrafo si analizzerà l'applicazione già esistente e, con un processo di reingegnerizzazione, si otterrà l'architettura logica di progetto platform-dependent sfruttando le conoscenze e i concetti illustrati in Appendice A.

4.1.1 Struttura dell'applicazione originale

L'applicazione presenta una serie di viste tra le quali si può effettuare lo switch utilizzando una Tab Bar. Le viste sono: Ordini, Gestione, Dati Personali, Importazione, Esportazione, Manuale e Sincronizzazione. La Tab Bar permette di navigare stack di View Controller diversi mantentendone lo stato. La struttura complessiva dell'applicazione in termini di View Controller è mostrata in Figura 4.1.

In Ordini si trova un Navigation Controller che contiene una serie di View Controller al suo interno. Il primo Controller, che è quello con cui l'applicazione si presenta all'utente al momento del lancio, è un View Controller di un tipo piuttosto comune, di nome `UITableViewController`, che contiene al suo interno le funzionalità per mostrare all'utente una serie di elementi organizzati in una lista (o in una tabella) e i metodi per gestire eventi quali il riempimento di ogni riga e la selezione di una di queste da parte dall'utente. Le righe possono essere riempite in modo statico o dinamico in base ad esempio al risultato di una query. È un modo molto utilizzato per effettuare la navigazione, passando a un successivo View Controller, in base alla riga scelta. È quello che succede in questa schermata, che è riempita con 5 righe statiche che permettono di scegliere tra: creare un nuovo ordine, visualizzare gli ultimi ordini creati, visualizzare l'archivio ordini, visualizzare gli ordini in base al fornitore o in base al cliente. Nel primo caso, si passa direttamente all'OrdineViewController, che permette di inserire i dati relativi all'ordine e di inviarlo; nei due casi successivi si passa a un View Controller

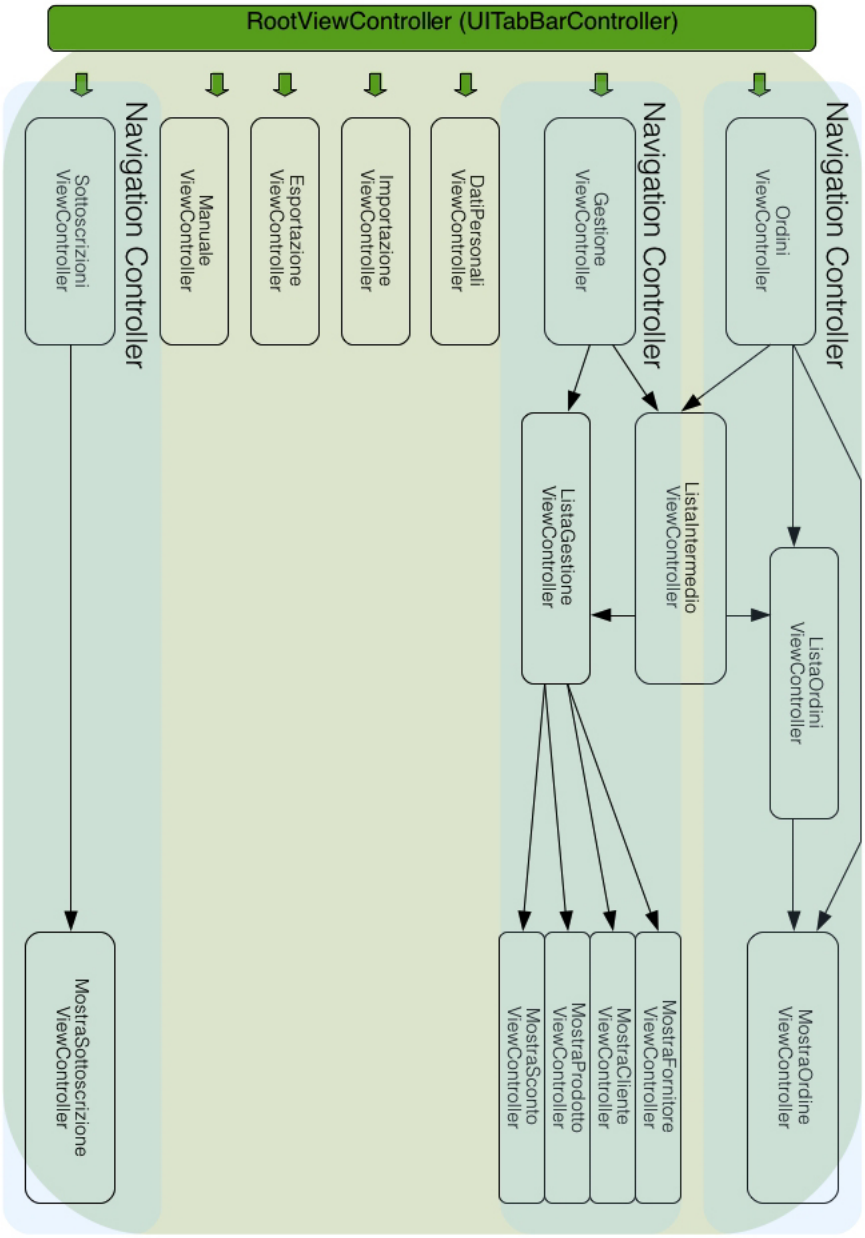


Figura 4.1: Schema della versione iOS dell'App in termini di View Controller

detto ListaOrdiniViewController. Esso è un Table View Controller che visualizza una serie di ordini organizzandoli in righe, mostrando se sono stati inviati o meno e permettendone la cancellazione (tramite bottoni in una barra in basso, inseribili e gestibili grazie alle funzionalità del `UITableViewController`) o la visualizzazione tramite passaggio a un OrdineViewController appropriatamento riempito con i dati dell'ordine selezionato. Gli ultimi due casi infine permettono di visualizzare un View Controller intermedio, dal quale selezionare fornitore o cliente del quale si desiderano visionare gli ordini.

In Gestione troviamo di nuovo un Navigation Controller. Un Table View Controller permette di effettuare la scelta tra Gestione Fornitori, Gestione Clienti, Gestione Prodotti e Gestione Sconti. Ognuna di queste sezioni porta a un altro Table View Controller che mostra tutti gli elementi della categoria (preceduto, nel caso dei Prodotti, da un Table View Controller intermedio per selezionare il fornitore e nel caso degli Sconti per selezionare il cliente). La selezione di un elemento porta al View Controller che contiene la maschera per la modifica dei dati: MostraFornitoriViewController, MostraClientiViewController, MostraProdottiViewController, MostraScontiViewController. Questo View Controller è lo stesso che permette anche l'inserimento di nuovi elementi, visualizzabile tramite il pulsante apposito presente nella lista di elementi. Dalla lista è inoltre possibile eliminare un elemento (come dal View Controller dedicato all'inserimento/modifica) o tutti gli elementi.

In Dati Personali troviamo un View Controller solo, che funge da maschera per l'inserimento e la modifica dei dati relativi all'agente che utilizza l'applicazione.

In Importazione ed Esportazione troviamo due View Controller singoli, ImportazioneViewController ed EsportazioneViewController, con le relative viste che spiegano la procedura di importazione/esportazione e permettono l'inserimento dei dati. Durante l'importazione/esportazione viene messa in campo una nuova vista per monitorare la procedura; la vista però non va a sovrapporsi interamente alla View precedente, quindi non viene effettuata una navigazione nè viene messo in campo una tab bar per spostarsi sulle Viste. Si vedrà più avanti come può essere gestita tale situazione.

Infine in Sincronizzazione c'è di nuovo una navigazione tra un Table View Controller e un MostraSottoscrizioneViewController che permette l'inserimento, modifica e cancellazione di dati relativi a una sottoscrizione. Da entrambi i View Controller è possibile far partire la sincronizzazione, che deve essere monitorata dall'utente mettendo in campo una View simile a quelle che vengono usate per l'importazione/esportazione.

Nell'applicazione inoltre troviamo alcuni dati del modello del dominio definito nella Sezione 3.2.3, in particolare del modello del dominio dei dati. Sottoscrizione è mappato in un'entità Sottoscrizione che racchiude anche i dati dell'Utente e la data di ultima Sincronizzazione. Ordine è mappato in un'entità Ordine che racchiude in sé l'entità Fornitore, Cliente, e i vari Prodotto e Sconto associati. Questi oggetti estendono tutti `NSManagedObject`, sono perciò integrati nella gestione del dato della piattaforma di iOS (si veda Sezione A.6.2): infatti, grazie all'utilizzo di questo genere di oggetti, si mappa direttamente la struttura degli stessi nella corrispettive Entity dello schema Entity-Relationship utilizzato dalla piattaforma per gestire i database e altri supporti (quali, ad esempio, Cloud e file). Per scelta di progetto, questi oggetti incapsulano anche il meccanismo di salvataggio, cancellazione e recupero di questo genere di oggetti (grazie ad una serie di metodi di classe).

Considerando il modello MVC su cui è basata la progammazione su iOS (si veda Sezione A.5 e in particolare Figura A.2), possiamo mappare così i componenti dell'applicazione:

- il *Modello* è composto dalle entità del dominio mappate su oggetti di tipo `NSManagedObjectModel`; esse incapsulano la logica di salvataggio su supporto.

- il *Controller* è formato da tutti i View Controller elencati, più altre eventuali classi di supporto come quella che effettua la sincronizzazione.

- la *View* è costituita da tutte le View a tutto schermo dei View Controller, più le View apposite per sincronizzazione, esportazione, importazione ed invio, che costituiscono fondalmentalmente dei popup.

4.1.2 Scenari

I moduli specificati nell'analisi del problema e nel progetto platform-independent sono facilmente implementabili creando semplici oggetti che estendono `NSObject`. Infatti, considerando i moduli dell'analisi come dei *Plain Old Object*, possiamo creare un parallelismo tra essi e un oggetto `NSObject`, perché le caratteristiche degli oggetti classici accomunano le due piattaforme. L'unica cosa che fa eccezione è il dinamismo di Objective C, che prevede di demandare a tempo di esecuzione il controllo su quali metodi gli oggetti possono effettivamente eseguire, cosa che invece in Java avviene a compile time. La differenza è sottile, perché riguarda un differente comportamento della piattaforma: in Objective C un oggetto potrebbe non rispondere ad un messaggio ma generare un errore, mentre in Java (risolti eventuali errori di compilazione) un oggetto valido risponderà sempre al metodo specificato nel codice. Questa differenza è interessante per quanto riguarda le interazioni, ma per il momento possiamo tralasciarla, e considerare semplicemente un oggetto che estenda `NSObject` come un POO, mappandone il comportamento in modo diretto.

Ciò che è interessante analizzare sono le interazioni tra questi *Plain Old Object* e soprattutto come monitorare il processo di sincronizzazione e invio dell'ordine in modo da dare all'utente un riscontro visuale di ciò che sta accadendo in background.

Per arricchire gli oggetti che sono stati citati nell'analisi del problema della capacità di interagire con il resto del sistema, si può utilizzare il pattern *Decorator*.

Si va ora ad analizzare caso per caso i tre scenari relativi ai tre casi d'uso principali di Order Sender. Per i tre casi si analizzerà la struttura specifica dell'applicazione messa in campo per lo specifico sottoproblema, e creando un parallelismo tra la struttura reale e l'architettura logica si otterrà l'architettura logica platform-dependent relativa al sottoproblema.

Compilazione dell'ordine

Durante la compilazione di un ordine, è messa in campo una View OrdineView e il relativo View Controller MostraOrdineViewController. La View ha una

serie di `UITextField`, che sono oggetti di tipo `UIControl`. I dati vengono resi disponibili dall'interfaccia impostando il View Controller come *delegate* dei vari `UITextField` da *Interface Builder*; inoltre, occorre che il View Controller sia conforme al protocollo `UITextFieldDelegate`, implementando una serie di metodi relativi al cambiamento dello stato dei Text Field. Essi sono referenziati dal View Controller stesso attraverso l'utilizzo di *Outlet*, ovvero mantentendo un riferimento al TextField e contrassegnandolo come `IBOutlet`, per segnalare alla piattaforma che deve collegarlo al corrispettivo elemento nel file *nib* dopo averlo caricato.

Per salvare un nuovo dato nel momento in cui esso è reso disponibile, occorre che sia implementato il metodo `textFieldDidEndEditing:(UITextField *) textField` che viene chiamato dalla piattaforma nel momento in cui un Text Field sia stato riempito con del testo. Il metodo viene chiamato utilizzando il meccanismo target-action: la piattaforma intercetta l'evento, riconosce che è associato ad un Control, identifica il target e gli invia l'Action da eseguire. A questo punto, il View Controller riconosce quale Text Field è stato modificato e aggiorna il relativo dato nell'oggetto Ordine che sarà mantenuto al suo interno. Dopo l'aggiornamento, per rendere permanenti le modifiche, è necessario comandare all'Ordine di salvarsi su database con il metodo apposito. Si riporta in Figura 4.2 una formalizzazione di questo meccanismo. Confrontandola con Figura 3.5, si può vedere che il mapping dei componenti è uno-a-uno. In ognuna delle figure rappresentanti gli scenari su iOS, le frecce tratteggiate rappresentano chiamate a metodi.

Invio dell'ordine

L'invio dell'ordine vede coinvolto lo stesso View Controller ma ha due View da gestire. Una è la form di compilazione (MostraOrdineView) che contiene un pulsante Invia, alla pressione del quale l'ordine viene validato (ovvero, viene verificato che contenga almeno il nome del fornitore, il nome del cliente e un prodotto; in caso contrario la cosa viene segnalata all'agente tramite una finestra di dialogo, ovvero un *alert*, e l'ordine non può essere quindi inviato), dopodiché viene

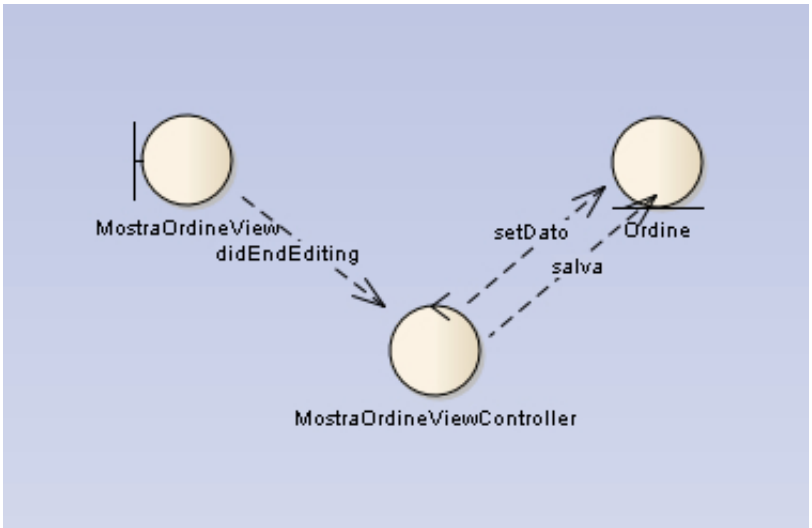


Figura 4.2: Compilazione dell'ordine su iOS

messa in campo un'ulteriore View per far inserire all'agente tre indirizzi e-mail a cui inviare l'ordine. Infine l'agente premerà il pulsante Invia e si darà inizio alla procedura di Invio. Le due View e il View Controller comunicano utilizzando il meccanismo Target-Action (si veda Sezione A.6.5): il bottone Invia, che è un `UIButton` e quindi un `UIControl`, ha una serie di eventi a cui si possono associare un target (che in questo caso è il View Controller) e una action (che nel caso dell'evento di tipo `UIControlEventTouchUpInside` è il metodo del Controller `inviaOrdine`).

Quando l'ordine deve essere inviato, il Controller prepara la richiesta HTTP con tutti i dati necessari e la manda al server. Ci sono due modi per inviare una richiesta HTTP utilizzando le classi della piattaforma: sincrono e asincrono. L'invio sincrono è bloccante sulla chiamata al metodo di classe `sendSynchronousRequest:returningResponse:error:` di `NSURLConnection` sino a che non si riceve una risposta dal server o la richiesta fallisce dopo un certo timeout. Fare la chiamata in questo modo significa bloccare il thread corrente per un tempo imprecisato (che può essere anche di un minuto o più, in condizioni di cattiva connettività). Fare la chiamata sul main thread è rischioso (oltre che sbagliato) perché nel caso in cui, passato un certo tempo senza che la chiamata si

sia sbloccata, l'applicazione viene chiusa forzatamente dalla piattaforma. Occorre perciò mettere in campo un altro thread con cui interagire. Se si utilizza uno dei metodi asincroni di `NSURLConnection`, si lascia fare tutto alla piattaforma. Infatti, la chiamata al metodo genera l'avvio di un nuovo thread in modo implicito. Non è necessario preoccuparsi dell'interazione tra questo e il main thread, poiché è tutto gestito implicitamente dalla classe; è sufficiente che il View Controller venga impostato come delegato dell'operazione. Per monitorare la chiamata, vengono chiamati i metodi del protocollo relativo `NSURLConnectionDelegate`, eseguiti direttamente sul main thread. L'interfaccia perciò rimane reattiva e il main thread libero e in grado, nel frattempo, di gestire altri eventi.

L'interazione nei due casi è diversa. Con la chiamata sincrona, si effettua semplicemente una method-call che restituisce un risultato. Se si utilizza la chiamata asincrona (come è stato fatto, visualizzando nel frattempo un indicatore di avanzamento), è come se si inviasse un messaggio alla classe che gestice la connessione. Essa servirà il messaggio e successivamente chiamerà i metodi della classe per notificare il risultato.

In Figura 4.3 si riporta la schematizzazione di questa parte.

Sincronizzazione

Il popup di sincronizzazione viene visualizzato a partire dalla MostraSottoscrizioneView, quindi il suo Controller è il MostraSottoscrizioneViewController. Al momento di far partire la sincronizzazione, verrà creato (a meno che non sia già presente) un oggetto Sincronizzatore e verrà chiamato il metodo sincronizza, passandogli i dati relativi all'importazione in un oggetto Importazione. A questo punto, il sincronizzatore eseguirà la sincronizzazione passando attraverso una serie di stati (mostrati in Figura 3.10). Sarà necessario comunicare all MostraSincronizzazioneViewController i vari cambiamenti di stato.

Un modo per fare questo è utilizzare la delegazione, imitando l'approccio adottato dalla piattaforma. È possibile creare un protocollo del tipo:

```
@protocol SincronizzazioneProtocol
-(void)onUpdateAvailable;
```

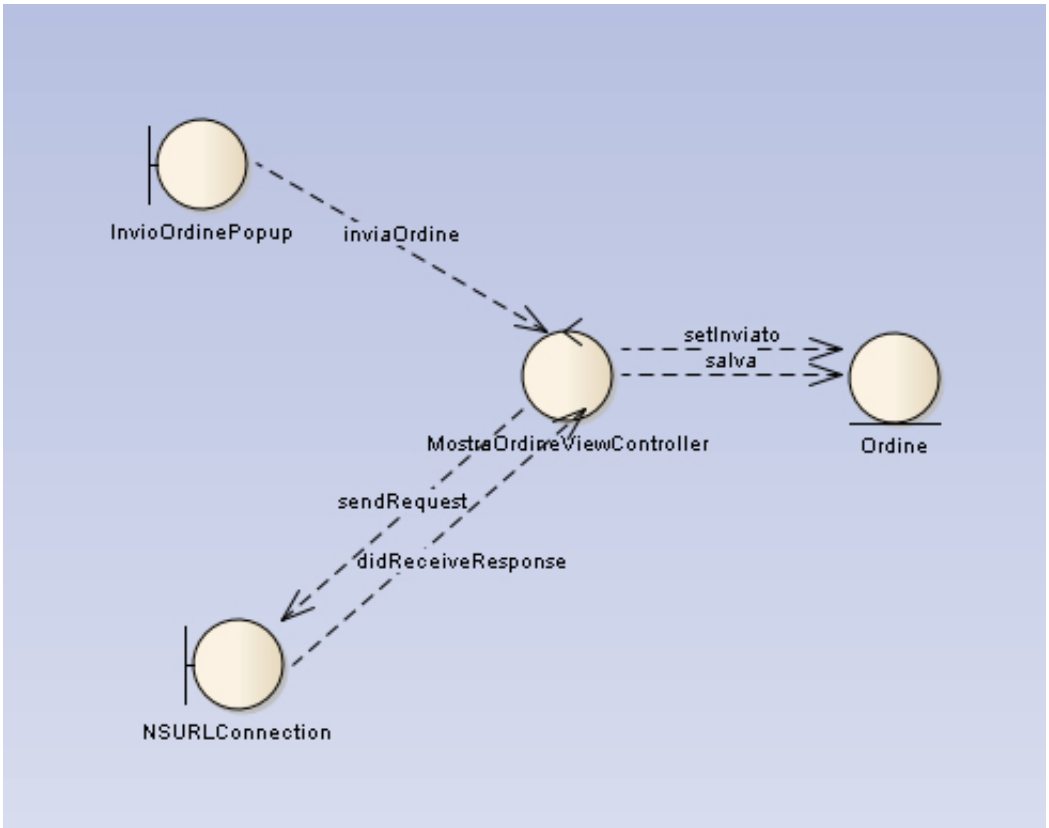


Figura 4.3: Invio dell'ordine su iOS

```
-(void)onUpdateNotAvailable;
-(void)onSubscriptionDeleted;
-(void)onPasswordWrong;
-(void)onConnectionError;
-(void)onElementSynchronized;
-(void)onElementNotSynchronized;
-(void)onSynchronizationFailed;
-(void)onSynchronizationFinished;
@end
```

Se il View Controller adotta questo protocollo, può essere avvisato dell'avvenuto cambiamento di stato all'interno del sincronizzatore. Esso non ha bisogno di sapere qual è l'oggetto da avvisare, gli è sufficiente che sia conforme al

protocollo. L'oggetto può essere passato nella fase di inizializzazione della sincronizzazione insieme ai dati relativi alla sincronizzazione stessa. A quel punto il sincronizzatore dovrà semplicemente mantenere un riferimento all'oggetto delegato e chiamarne i metodi del protocollo al momento giusto. I metodi devono essere eseguiti sul main thread poiché l'esecuzione dovrà necessariamente essere spostata su un thread apposito; la cosa è fattibile semplicemente utilizzando il metodo `performSelectorOnMainThread:withObject:waitUntilDone:` di `NSObject`.

La delegazione funziona bene se l'osservatore interessato è uno solo; in caso contrario, è possibile utilizzare le Notifications, personalizzando oggetti `NSNotification` per rappresentare i vari eventi. In questo modo il disaccoppiamento tra *Observer* e *Observed* è massimo.

Il sincronizzatore conterrà tutta la logica di sincronizzazione che è stata vista nell'analisi del problema sotto forma di normali oggetti. Esso inoltre dovrà anche prevedere l'utilizzo del gestoreHTTP, che in questo caso è `NSURLConnection`; in questo caso le richieste possono essere di tipo sincrono perché la sincronizzazione sarà già stata spostata su un thread apposito.

In Figura 4.4 si riporta una schematizzazione degli aspetti importanti di questa sezione dell'applicazione.

4.1.3 Architettura logica

In Figura 4.5 si riporta l'architettura logica dell'applicazione su iOS. Per semplicità si sono omessi i componenti che fanno parte del sincronizzatore (Parser, ecc.) che vedono comunque una interazione ad oggetti. Nella figura, le frecce tratteggiate indicano interazioni a method-call.

4.2 Order Sender per Android

In questa sezione si analizzeranno gli scenari presentati e alcune possibili soluzioni per l'architettura logica di progetto platform-dependent su Android. Se-

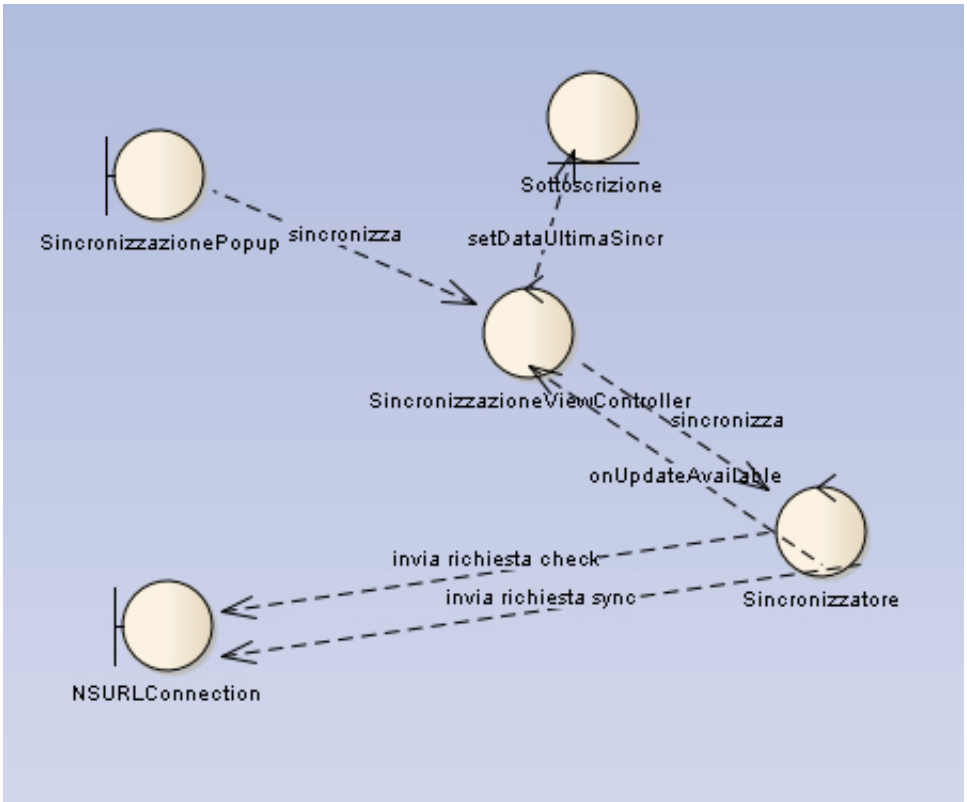


Figura 4.4: Sincronizzazione su iOS

guiranno una serie di considerazioni sulla struttura complessiva dell'applicazione e sulle viste che essa deve presentare per essere conforme al programma originale.

4.2.1 Scenari

Compilazione dell'ordine

La compilazione dell'ordine prevede l'interazione con l'utente, perciò è necessario mettere in campo un'Activity. L'Activity sarà la MostraOrdineActivity, duale del MostraOrdineViewController, con la sua vista definita tramite l'XML e definita chiamando `setContentView()` dal metodo `onCreate()`. Per essere avvisata delle modifiche che avvengono sui widget di tipo Text Field dove l'utente dovrà scrivere, l'Activity dovrà creare e associarvi un listener. Nel momento in cui

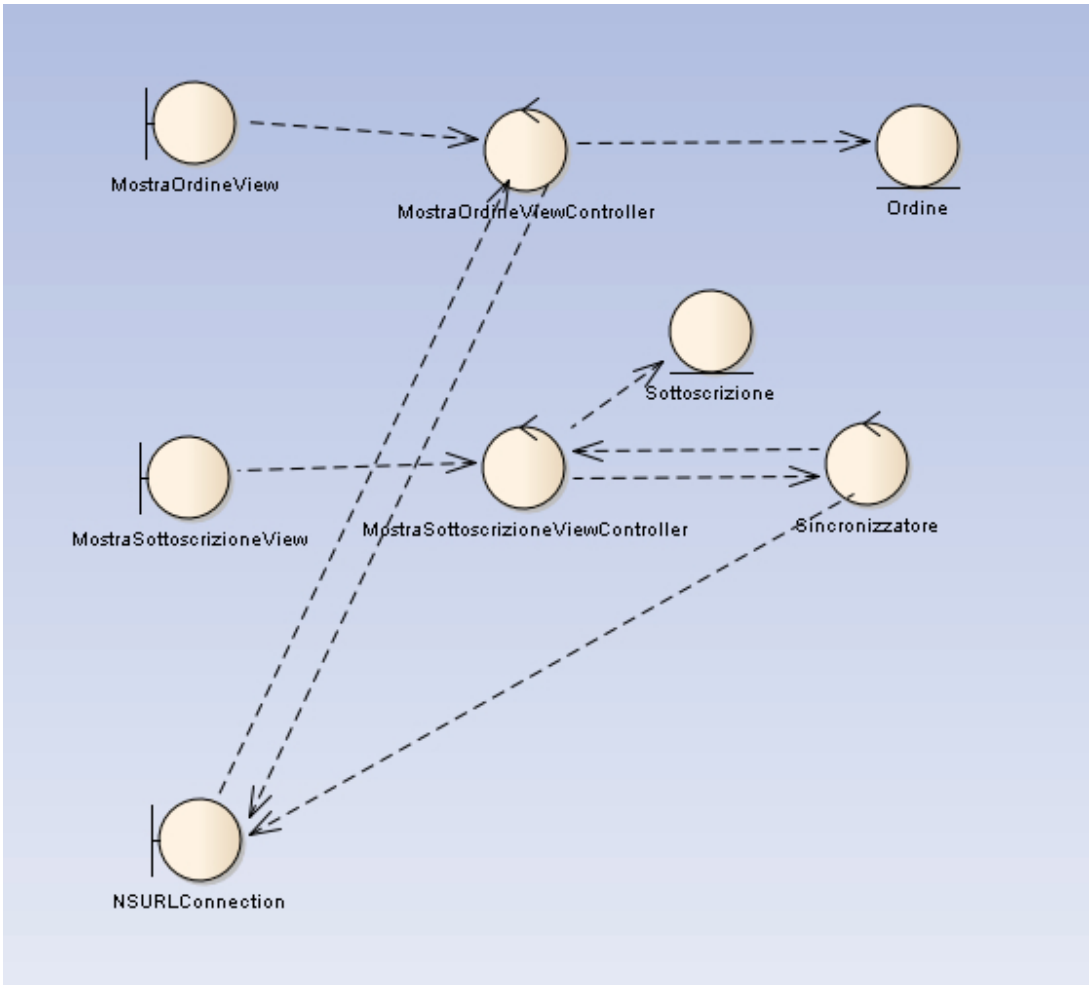


Figura 4.5: Architettura logica su iOS

un nuovo dato sia disponibile, l'Activity dovrà aggiornare la entity corrispondente all'Ordine corrente. Ha senso che l'oggetto che rappresenta l'ordine sia associato all'istanza dell'Activity poiché i dati rappresentati dalla Activity sono proprio quelli di quell'Ordine. L'Activity dovrà poi rendere i cambiamenti permanenti salvando nel Database relativo all'App l'Ordine; ciò può essere fatto utilizzando un Content Provider. Il Content Provider dovrà essere creato come sottoclasse di `ContentProvider`, implementandone i metodi astratti. A quel punto, l'interazione con il Content Provider e quindi con il database non sarà più diretta, ma mediata

dalla piattaforma attraverso l'utilizzo dei metodi della classe `ContentResolver`. Per inserire un dato, occorrerà chiamare un metodo del Content Resolver (la cui istanza è fornita dalla piattaforma usando `getContentResolver()`) e specificando l'URI del Content Provider e gli identificativi di tabella, colonna e/o riga. Anche se l'interazione con questo componente è mediata dalla piattaforma, occorre comunque interagire con il database in modo quasi diretto, come se si utilizzasse un'istanza della classe SQLiteDatabase, che "wrappa" appunto un database. Anche per ottenere i dati da questo componente occorre interagire con un cursore che contiene i dati, in modo diretto. Per ovviare a questo inconveniente che porta la logica di memorizzazione dei dati troppo vicino a quella di interazione con l'utente, si può scegliere o di inserire la logica di salvataggio e recupero dei dati direttamente nella Entity, come è stato fatto su iOS, oppure di creare un vero e proprio Wrapper delle interazioni con il supporto. Si sceglie di utilizzare il pattern decorator per questo, arricchendo gli oggetti della logica di interazione con il database, poiché l'unico modo per creare un Wrapper sarebbe l'utilizzo di metodi statici e/o di un *singleton*, ovvero un'istanza unica in tutta l'applicazione, che per sua natura può portare facilmente a incorrere in errori di progettazione e programmazione.

In Figura 4.6 si riporta una schematizzazione degli elementi principali illustrati.

Invio dell'ordine

Anche in questo caso è necessaria una forma di interazione con l'utente perciò sarà necessario coinvolgere nella progettazione una Activity, che nella fattispecie sarà costituita dalla stessa MostraOrdineActivity già utilizzata. Ci sarà un bottone con un listener associato all'interfaccia; nel momento in cui esso venga premuto, l'interfaccia dovrà essere bloccata (mostrando ad esempio un indicatore di avanzamento) in attesa che l'ordine sia inviato. Per fare ciò, occorre spostare l'esecuzione del task di invio in un thread secondario, o ci si troverà molto facilmente in una situazione di Application Not Responding. Innanzitutto sarà necessario un modulo in grado di gestire le richieste HTTP, come nell'architettura di progetto

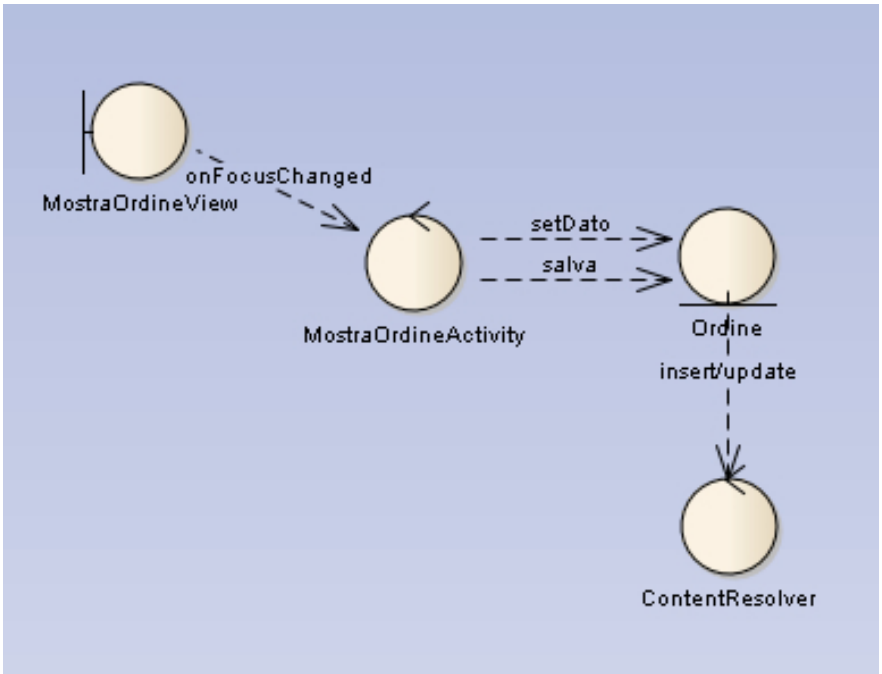


Figura 4.6: Compilazione dell'ordine su Android

platform-independent. Android permette solo di fare chiamate sincrone, perciò che si bloccano in attesa della risposta. In questo caso il gestoreHTTP è un normale oggetto che effettua la chiamata dentro un metodo e restituisce la risposta. Ovviamente questo metodo deve essere eseguito su un worker thread. Questo può essere fatto utilizzando un Service, in particolare un IntentService, che gestisce automaticamente un worker thread su cui far eseguire il lavoro presente nel metodo `OnHandleIntent(Intent)`, serializzando le richieste di esecuzione. La comunicazione tra i due avviene attraverso un Intent, che potrebbe contenere negli extra i dati relativi all'ordine da inviare. Siccome i dati possono essere una certa quantità, si potrebbe utilizzare un *singleton* o un metodo statico (ad esempio collocato nella mostraOrdineActivity) per ottenere tutte le informazioni del caso. Quando un IntentService finisce il suo lavoro, sul main thread viene chiamato il metodo `onDestroy()`. Dentro a questo metodo è possibile interagire con l'interfaccia, per sbloccarla ed eliminare l'indicatore di attività. Ciò deve essere fatto dalla Activity; occorre che il Service, terminato il suo lavoro, le no-

tifichi l'avvenuta esecuzione, comunicandole il risultato. Questo può essere fatto con un Intent, tra i cui extra vi sia il risultato dell'operazione, inviato dal Service alla MostraOrdineActivity, che per ricevere correttamente l'Intent senza essere nuovamente instanziata deve essere contrassegnata nel Manifest con l'indicazione `android:launchmode` come `singleTop`, che permette all'istanza dell'Activity che è correntemente sulla cima del back stack di gestire un Intent in arrivo con il metodo `onNewIntent(Intent)`. In base al risultato che essa riceverà con questo Intent, l'Activity constrassegnerà l'Ordine come "inviato", salvandolo sul Content Provider, oppure segnalerà all'utente il mancato invio dell'ordine.

Si riporta in Figura 4.6 lo schema di quanto si è detto. In ognuna delle figure rappresentanti gli scenari su Android, le frecce tratteggiate rappresentano chiamate a metodi, e le frecce piene rappresentano l'invio di Intent.

Sincronizzazione

In Android, si possono direttamente creare i POJO ricavati dall'analisi del problema. Una volta costruito l'oggetto Sincronizzazione, occorre capire come metterlo in comunicazione con il resto del sistema.

All'utente, nel momento in cui sia necessario far partire la sincronizzazione con i dati validi inseriti dall'utente, verrà mostrato un Dialog. Un Dialog è una classe Android che permette di associare e gestire una View che non copre interamente lo schermo. Come per le Activity, la View corrispondente viene associata con il metodo `setContentView`. Questo nel caso in cui sia necessario personalizzare il Dialog e la visualizzazione all'utente dello stesso, ma spesso possono essere utilizzati dei formati già presenti nel sistema. In questo caso, sarà necessario estendere la classe Dialog per creare un SincronizzazioneDialog con una View personalizzata dove mostrare l'andamento della sincronizzazione. Questo avviene, come già visto, attraverso messaggi di qualche tipo inviati dal Sincronizzatore nel momento in cui esso cambi stato.

È l'Activity che crea il Dialog e lo elimina quando non serve più, quindi un modo per gestire l'interazione tra un oggetto Sincronizzatore indipendente e le View è l'utilizzo di Intent che contengano dati customizzati ad hoc per questo tipo

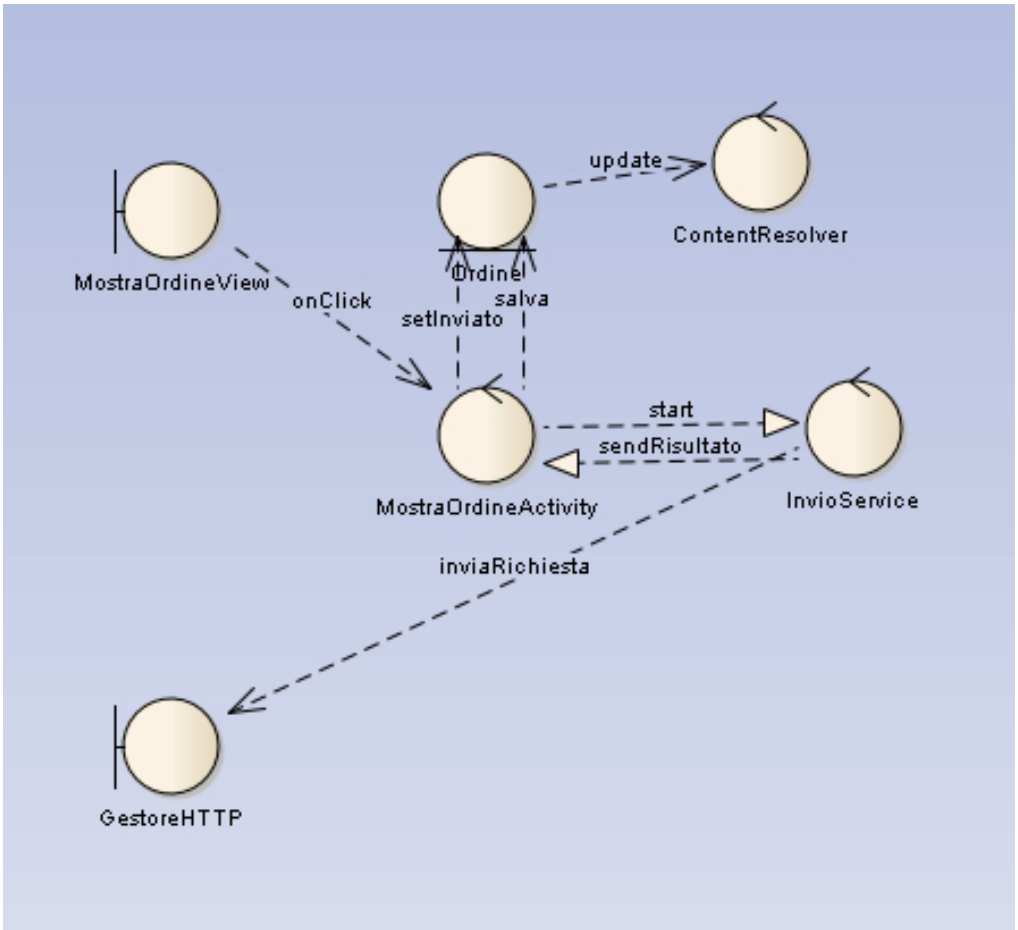


Figura 4.7: Invio dell'ordine su Android

di comunicazione, diretti all'Activity corrente, che sarà la MostraSottoscrizioneActivity. In questo caso occorre considerare che si utilizza la piattaforma come mediatore, rendendo l'interazione più pesante. Inoltre occorre modificare le politiche di istanziazione classiche della Activity poiché normalmente l'invio di Intent a una Activity ne provoca la creazione anche se essa esiste già, constrassegnandola con `launchmode "singleTop"` nel Manifest.

Un altro modo per gestire tutto questo è l'utilizzo di un *Bounded Service*. Un Bounded Service permette di utilizzare un Service con un rapporto client/server. Il Service mette a disposizione un interfaccia (IBinder) grazie alla quale gli altri componenti possono comunicare con esso. Si potrebbe pensare, ad esempio, di

creare un Binder che permette all'utente di monitorare lo stato dell'importazione con richieste dirette. Questo ovviamente costringe il Dialog ad essere sempre attivo, facendo una sorta di *polling*. Il Service è comunque un buon modo per isolare la sincronizzazione dal resto del sistema, utilizzando l'interfaccia per controllarne il flusso di esecuzione (per avviare la sincronizzazione, per annullarla, ecc).

In tutti questi casi occorre ovviamente gestire a mano l'esecuzione dell'importazione su un Thread apposito. L'utilizzo di AsyncTask[1] permette di utilizzare un Worker Thread e di pubblicare le modifiche sull'interfaccia in modo trasparente. Infatti, su Android il tentativo di aggiornare l'interfaccia da un Thread secondario produce una eccezione, al contrario di Java dove, nonostante possa produrre malfunzionamenti, è possibile farlo, anche se concettualmente sbagliato: le modifiche dell'interfaccia infatti vanno tutte eseguite sul Thread Swing, cosa che da Thread secondari è possibile utilizzando i metodi della classe SwingUtilities. In Android la cosa non è, giustamente, proprio possibile. La classe astratta AsyncTask permette di eseguire task in modo asincrono; in automatico essa crea un thread secondario dove far eseguire le operazioni contenute nel metodo `doInBackground()`. Le operazioni di preparazione che avvengono sul Main Thread sono specificate nel metodo `onPreExecute()` e quelle finali nel metodo `onPostExecute()`. Durante l'esecuzione in background è possibile pubblicare a mano a mano ciò che avviene utilizzando il metodo `publishProgress()`, che provoca l'esecuzione delle operazioni contenute in `onProgressUpdate()`. AsyncTask permette anche di gestire la cancellazione del task in background. Tutti i parametri che vengono passati ai metodi sono personalizzabili, in base ai tipi con cui si parametrizza la classe. Il primo tipo parametrico è quello che viene passato attraverso il metodo `doInBackground()`; il secondo rappresenta gli oggetti che possono essere passati all'`onProgressUpdate()`; il terzo è il tipo di ritorno del metodo `doInBackground()` e rappresenta anche ciò che viene passato al `onPostExecute()`. I primi due tipi possono essere più di uno nel momento in cui vengono passati (ovvero, le signature dei due metodi `doInBackground()` e `onProgressUpdate()` prevedono un numero variabile di parametri di quel tipo,

[1] AsyncTask è presente da API Level 3

utilizzando la sintassi `type ... variableName`), mentre l'ultimo è uno soltanto. Ciò significa che è possibile utilizzare oggetti per descrivere lo stato del Sincronizzatore e reagire, nei metodi preposti, di conseguenza. Il Sincronizzatore potrebbe pertanto essere incapsulato o incapsulare un AsyncTask, permettendo così il monitoraggio dello stato interno.

A fronte di queste considerazioni, la soluzione scelta è quella di utilizzare un Bounded Service per contenere il Sincronizzatore. Sarà necessario interagire con l'interfaccia molto spesso, inviando Intent all'Activity corrente, e la cosa è fattibile solo da main thread; ma il sincronizzatore dovrà necessariamente essere eseguito in un thread secondario; perciò conviene utilizzare un AsyncTask eseguito all'interno del Service. Questa soluzione risolve anche la problematica relativa all'annullamento della sincronizzazione in corso; sarà infatti sufficiente implementare un `IBinder` che fornisca anche un metodo per annullare la sincronizzazione.

In Figura 4.8 si riportano i principali elementi derivati da questa analisi.

4.2.2 Activity e le viste

Dalla struttura per iPad, che costituisce un vincolo poiché è già esistente e l'obiettivo era di replicare nel modo più simile possibile il sistema su Android, possiamo ottenere abbastanza facilmente l'insieme di Activity che dobbiamo mettere in campo per duplicare l'interfaccia. La struttura complessiva dell'applicazione come insieme di Activity è mostrata in Figura 4.9.

Per prima cosa, ciò che permette di navigare su iPad i vari View Controller è una Tab Bar. Una struttura del genere esiste anche su Android, ma in quel caso si perdono i vantaggi della navigazione a stack intrinseca nella struttura del sistema operativo poiché ad ogni tab corrisponde una sola Activity, a meno di non utilizzare l'escamotage dell'Activity Group, un modo per raggruppare le Activity e farle risultare come una sola.

Siccome invece in Android esistono i menu, si è scelto di utilizzarli. I menu, quando definiti, sono raggiungibili da qualunque Activity attraverso l'apposito tasto presente in tutti i dispositivi. Ogni Activity definisce un proprio menu, ma

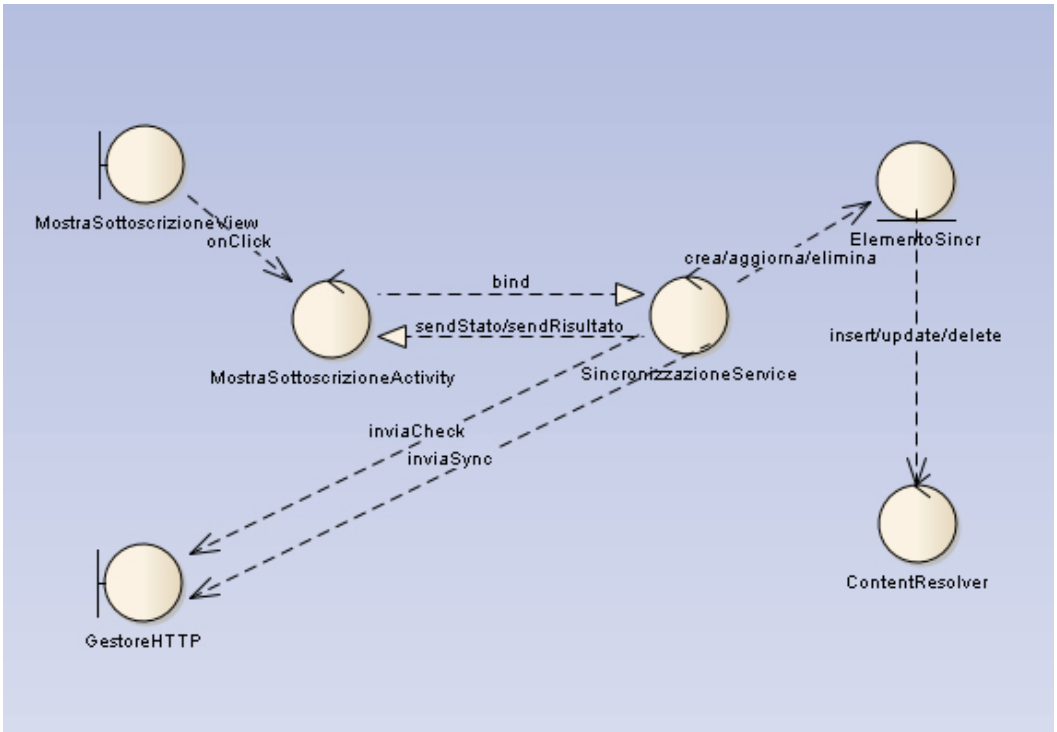


Figura 4.8: Sincronizzazione su Android

per avere un menu tutto uguale ovunque nell'applicazione è sufficiente definirlo in una Activity ad hoc e poi creare tutte le Activity dell'applicazione facendo il subclassing della MenuActivity.

Questo approccio introduce una differenza tra le due versioni dell'applicazione: se da menu si seleziona una funzionalità come la Gestione e poi si torna, sempre utilizzando il menu, alla funzionalità precedente, ovvero gli Ordini, ciò provoca l'inserimento di due istanze di Activity diverse nel back stack, cosa che si può vedere navigando indietro con il tasto back: dalla nuova Activity degli ordini, si trova l'Activity della Gestione e poi un'altra Activity degli ordini. Tenendo conto che premendo il tasto back la cima del back stack viene estratta e distrutta, si può capire che le Activity sono duplicate. Questo è un aspetto che non introduce problematicità funzionali all'applicazione che può andare bene così, ma è una cosa di cui comunque occorre tenere conto quando si progettano e codificano le

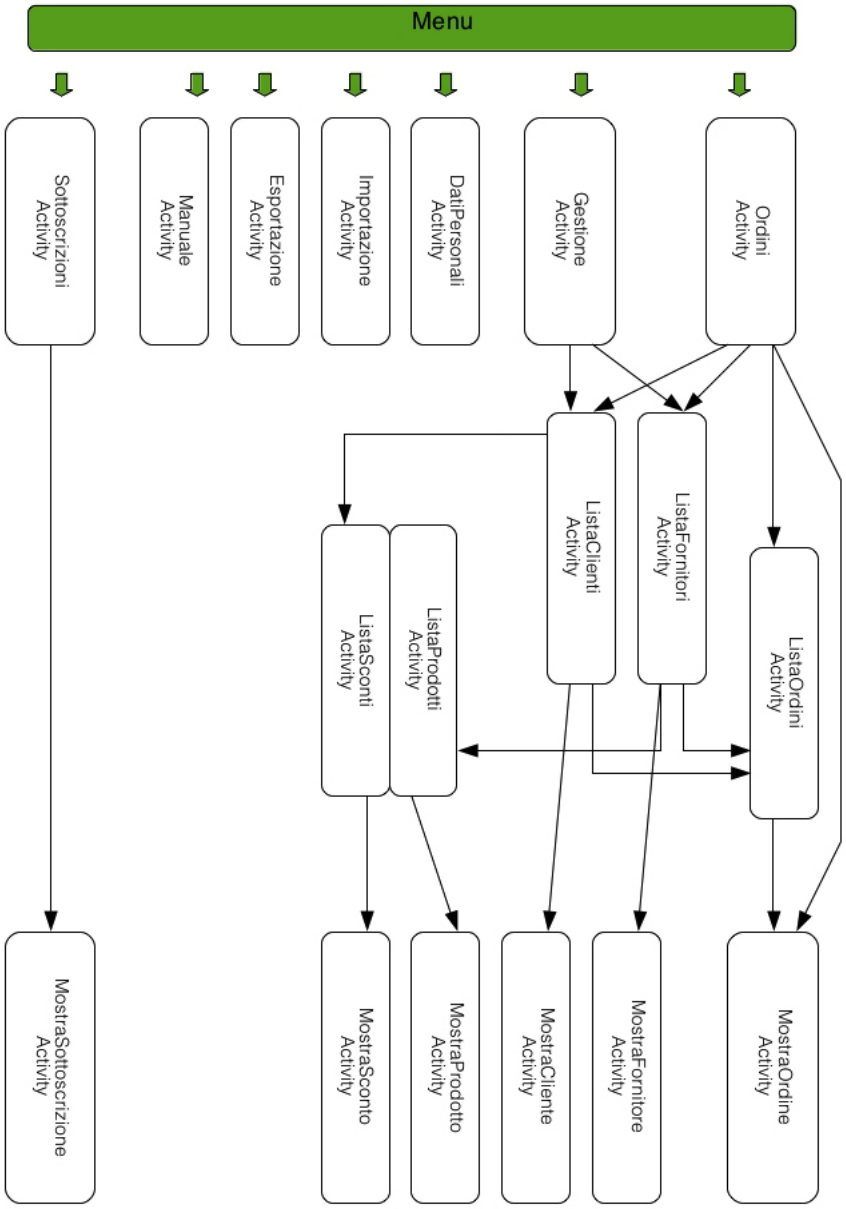


Figura 4.9: Schema della versione Android dell'App in termini di Activity

Activity. Questo comportamento di default può essere modificato attraverso gli opportuni flag da specificare nel file Manifest in congiunzione con l'Activity.

Dai 6 elementi del menu quindi partono altrettante Activity. Quasi tutte danno poi il via alla visualizzazione di ulteriori Activity. Avremo una OrdiniActivity, una GestioneActivity, una DatiPersonaliActivity, una ImportazioneActivity, una EsportazioneActivity, una ManualeActivity e infine una SottoscrizioneActivity. Dalla OrdiniActivity possono partire una MostraOrdini o una ListaOrdini che discende da una Activity ILista; dalla gestione una qualunque altra Activity ILista (eventualmente due, per sconti e prodotti) che porti poi ad una Activity MostraCliente, MostraFornitore, MostraProdotto o MostraSconto, infine la SottoscrizioniActivity, che è anch'essa una ILista, porta ad una Activity MostraSottoscrizione.

4.2.3 Architettura logica

In Figura 4.10 si riporta l'architettura logica dell'applicazione su iOS. Anche qui, per semplicità si sono omessi i componenti che fanno parte del sincronizzatore (Parser, ecc.) che vedono comunque una interazione ad oggetti e non sono differenti, come realizzazione, dall'architettura per iPad. Nella figura, le frecce vuote indicano interazioni a method-call e le frecce piene indicano l'invio di Intente.

4.3 Confronto tra le due soluzioni

Dalle considerazioni di questo capitolo si possono ricavare una serie di conclusioni riguardo alla risoluzione del problema di Order Sender sulle due piattaforme e, in generale, alla risoluzione di problemi generici che si possono facilmente incontrare durante la progettazione di sistemi di qualunque tipo.

Lo sviluppo di applicazioni su iOS, basata sul pattern MVC, è prevalentemente orientato agli oggetti e pertanto si presta abbastanza facilmente ad una progettazione classica object-oriented, utilizzando ad esempio tool come UML. Ovviamente questo non vale per tutti i problemi e per tutti gli aspetti del problema

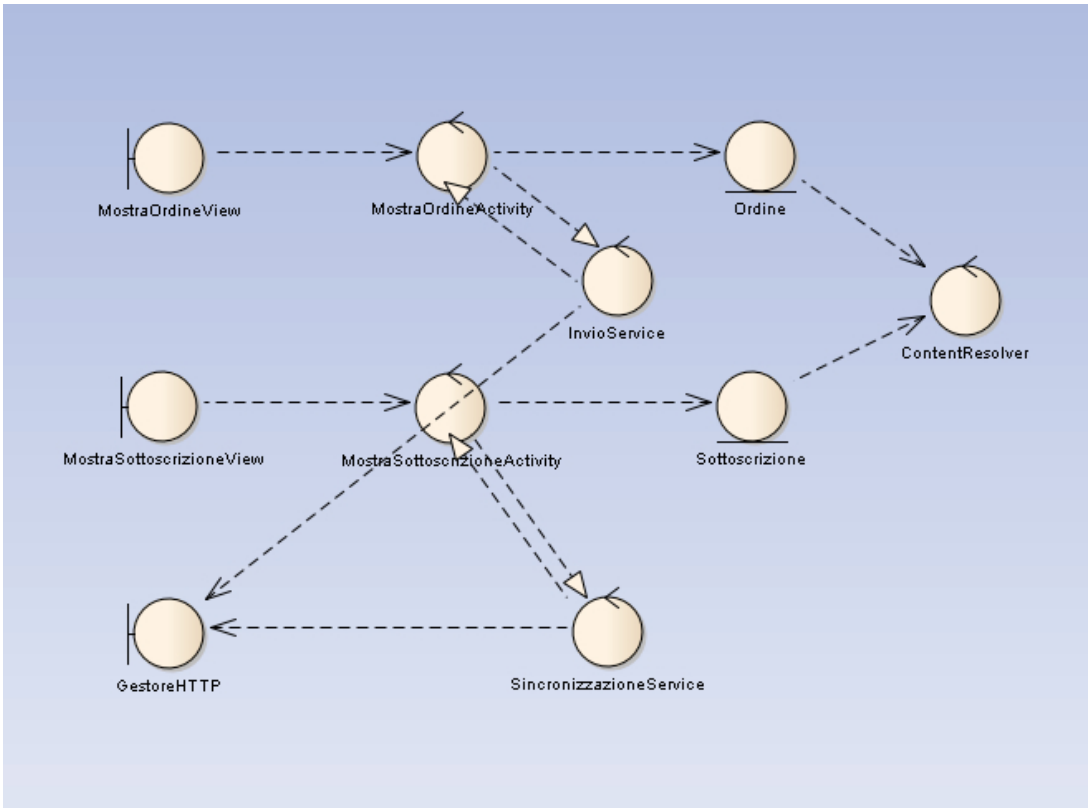


Figura 4.10: Architettura logica su Android

presentato, poiché la piattaforma offre anche alcune astrazioni interessanti come la delegazione, che sono però comunque implementate a livello di method call. Altre astrazioni presenti invece sono realizzate utilizzando la piattaforma come mediatore e non sono direttamente riconducibili alla classica chiamata a metodi, come le Operation, le Notification Queue o i Run Loop.

Rispetto all'architettura logica platform-independent troviamo che il rapporto tra i componenti messi in campo nell'architettura logica platform-dependent è pressoché uno-a-uno, eccezion fatta per il Controller originale che si spezza in più componenti, ovvero in due View Controller relativi alle View messe in campo nel momento in cui si rende necessaria l'interazione con l'utente, e poi in un'ulteriore componente Sincronizzatore, poiché si è reso necessario, a fronte di un Controller così relativo alle viste, isolare la funzionalità specifica della sincronizzazione in

un componente a sé stante.

Le interazioni indicate nell'analisi del problema tra i componenti diventano su iOS interamente a method call (cosa possibile essendo in un ambiente concentrato). Gli eventi generati dalla piattaforma vengono inviati all'applicazione e successivamente consegnati al componente in grado di gestirli con una chiamata a metodo. L'interazione sincrona con il modulo che comunica con la rete è nuovamente a method-call e lo stesso vale con gli oggetti di tipo `NSManagedObject` con la loro logica di salvataggio su supporto. Le comunicazioni tra il View Controller e il Sincronizzatore o l'interazione asincrona con il modulo che comunica con la rete avvengono anch'esse a method call, ma mettendo in campo la delegazione si trova un design pattern (che nel secondo caso è implementato dalla piattaforma e nel primo caso si è scelto di imitarlo) che permette di rendere modulare l'accoppiamento tra i due oggetti che la utilizzano. Infatti la delegazione si avvale del concetto di interfaccia (che su iOS diventa *protocollo*) per disaccoppiare il delegato dal delegante, che non deve preoccuparsi di qual è l'oggetto a cui fa eseguire i compiti. In questo caso specifico in cui la delegazione è utilizzata più che altro per segnalare qualcosa, normalmente un cambio di stato, il delegato non necessariamente deve fare qualcosa con queste "segnalazioni" che il delegante gli consegna invocando i suoi metodi, quindi il metodo deve essere implementato ma non necessariamente deve contenere qualche istruzione. In questo senso la delegazione può assomigliare al `dispatch` che abbiamo appunto scelto nell'analisi del problema come modalità di interazione tra il componente che effettua la sincronizzazione e la vista. In questo modo il delegante esegue i suoi compiti chiamando i metodi del delegato senza preoccuparsi di cosa effettivamente l'utente visualizzerà alla fine. L'importante è definire bene il protocollo in fase di progetto, in modo da includere tutti gli eventi e i cambi di stato degni di nota.

Lo sviluppo di applicazioni su Android invece si basa su un'architettura a componenti che interagiscono a message-passing, perciò si discosta in modo deciso dalla classica progettazione a oggetti; o meglio, essa può essere utilizzata soltanto all'interno dello stesso componente, mentre invece l'architettura complessiva del sistema va pensata in termini dei componenti forniti dalle librerie con

cui si è arricchito Java per sviluppare su dispositivi mobili. Questi componenti hanno inoltre una precisa semantica. Le Activity incapsulano il comportamento di una funzionalità che richieda l'interazione con l'utente, i Service sono componenti che eseguono compiti in background e i Broadcast Receiver sono fatti per ricevere messaggi broadcast dalla piattaforma e per reagirvi in modo coerente. Infine i Content Provider incapsulano l'accesso ai dati, anche se l'accesso a questi funziona comunque a method call anche se in modo mediato dalla piattaforma.

Le interazioni previste nell'analisi del problema si riducono tutte a Intent o a chiamata a metodi. Il `dispatch` previsto per l'aggiornamento della View prende la forma di un Intent inviato da parte del Service. L'intent può essere inviato all'Activity in modo esplicito ed essa può eventualmente decidere di ignorarlo (in modo da ricalcare lo stesso ragionamento fatto per la delegazione) oppure si possono utilizzare intent impliciti con categorie e azioni definite ad hoc che poi l'Activity interessata potrà ricevere o meno in base ai filtri che vengono impostati nel Manifest.

La diversa semantica di Activity e Service porta a spezzare, anche qui, il controller originale in più parti. Infatti occorre definire una parte di Controller più vicina all'interazione all'utente e un'altra che rappresenti l'attività in background.

Confrontando le due architetture di progetto viene naturale accostare le Activity e i View Controller, perché svolgono lo stesso ruolo e costringono allo stesso tipo di ragionamento per quanto riguarda la modularità del controllo. Occorre tenere presente però che i concetti non sono esattamente uguali e le interazioni tra essi e il resto del sistema sono diverse.

Impostando perciò l'architettura logica platform-dependent di iOS in questo modo, isolando le funzionalità dalle viste, risulta quasi più agevole progettare su Android a partire dal modello di iOS che dal modello platform-dependent, con le dovute precauzioni e considerando che non è corretto impostare una progettazione a partire dalle viste. La cosa inoltre vale solo per questo tipo di applicazioni, ma non è detto che sia lo stesso anche per altri classi di problemi.

In questo senso si nota che la programmazione su mobile è in generale molto orientata alle viste e all'interazione con l'utente. Android, nel caso in cui si

vogliano effettuare funzionalità a sé stanti, offre il concetto di Service (e quello di Content Provider se si vogliono isolare le funzionalità relative ai dati) proprio a livello di progettazione. La progettazione su iOS invece costringe a fare uno sforzo per isolare le funzionalità dai View Controller.

Per concludere, si è visto che utilizzando Contact come linguaggio per la rappresentazione delle interazioni, si è ricavato come implementare alcune astrazioni adattandole alle piattaforme di interesse. Il problema presentava poche e semplici interazioni (ovvero `request/response` e `dispatch`), che sono state adattate prevalentemente con method-call. Su iOS si è anche utilizzato il design-pattern della delegazione mentre su Android si sono usati gli Intent. Siccome però Contact prevede anche altre forme di interazione che potrebbero emergere da altri tipi di problemi, per una trattazione approfondita di questo aspetto si rimanda al Capitolo 2.

Conclusioni e sviluppi futuri

Lo spunto per questa tesi è stato dato da una esigenza di un'azienda di sviluppare un'applicazione per Android a partire dalla versione già esistente su iPad, di cui era disponibile soltanto il codice.

Per effettuare questo tipo di porting si è reso necessario lo studio delle due piattaforme per capire quali astrazioni esse mettessero a disposizione dello sviluppatore. Nell'approfondire i due stili architetturali, si è visto che erano molto diversi e che non era possibile effettuare semplicemente un passaggio diretto da una forma all'altra, ma che era necessario capire quale fosse il problema a cui era sottoposto lo sviluppatore e di conseguenza ricavare un modello che lo rappresentasse. Si è scelto perciò di effettuare un reverse-engineering a partire dal codice già implementato per iPad in modo da ricavare analisi dei requisiti, analisi del problema e una parte di progetto platform-independent.

Siccome si è scelto di utilizzare UML per rappresentare struttura e comportamento e il linguaggio custom Contact per le interazioni nell'analisi del problema, si è potuto osservare come queste forme di rappresentazione si trasformassero sulle due piattaforme, in particolare l'aspetto dell'interazione. Dal problema però emergevano solo alcune astrazioni di Contact; si sono perciò anche approfondite le forme di interazione delle piattaforme per vedere quali astrazioni del linguaggio Contact fossero paragonabili a quelle presenti nelle architetture e quali invece portassero a riflettere sulle forme di interazione già presenti in esso.

Le due soluzioni sono state messe a confronto e si è potuto osservare che:

- La progettazione su iOS non si discosta molto dalla normale progettazione ad oggetti, anzi costringe ad utilizzare il pattern MVC, rendendo l'approc-

cio più ordinato. La programmazione sulla piattaforma però offre anche numerose astrazioni, oltre a quella ad oggetti, che si possono prestare ad altri scopi.

- La progettazione su Android offre una serie di nuove astrazioni, quali Activity, Service, Intent e Broadcast Receiver, che prevedono un'interazione a message-passing tra componenti che possiedono una specifica semantica. Ciò implica che anche per applicazioni concentrate è necessario progettare le varie parti in modo che comunichino a message passing e occorre perciò decidere come separare le funzionalità per inserirle nel componente che possiede la semantica adatta.

Da questa analisi comparata è possibile ricavare spunti per la realizzazione di una software factory orientata allo sviluppo per mobile. Infatti, a partire dall'analisi delle piattaforme soprattutto in relazione al linguaggio *Contact* si potrebbe estendere il linguaggio stesso per abbracciare anche le forme di interazione suggerite dalle piattaforme. Si potrebbe pensare inoltre di sviluppare un generatore di codice che utilizzi questo o altri linguaggi custom per ottenere codice applicativo già pronto a partire da una certa specifica. Generando codice a partire da un Contact esteso, si potrebbe ottenere un generatore di codice che si occupi già di definire struttura e interazioni nel sistema, lasciando al programmatore solamente l'onere di definire il comportamento dei singoli componenti. Alternativamente potrebbe essere possibile, per una certa classe di problemi, effettuare una prima fase di progetto generico che porti a capire come l'applicazione deve essere strutturata e deve comportarsi, per poi sviluppare una serie di generatori che a partire dalla specifica del sistema lo implementino direttamente sulle due piattaforme. L'applicazione Order Sender potrebbe essere un esempio di generatore da sviluppare a partire dall'analisi eseguita in questa tesi.

Appendice A

Lo sviluppo di applicazioni su iOS

iOS[1] è un sistema operativo proprietario della Apple Inc., sviluppato dalla Apple stessa per i propri dispositivi con interfaccia touch. Questi dispositivi sono iPod touch, iPhone e iPad. La versione corrente di iOS è la 5.1.

Creare applicazioni, ovvero App, per ambiente iOS significa sviluppare per un target di dispositivi noti e poco numerosi. Infatti, iOS è distribuito e utilizzato soltanto sulle varie versioni dei dispositivi citati. Inoltre, tutti questi dispositivi sono Apple, quindi il produttore del sistema operativo e della piattaforma è lo stesso produttore dei dispositivi.

A fine sviluppo, le App vengono pubblicate su App Store, che è il canale preferenziale (e pressoché unico) per la distribuzione di applicazioni per dispositivi touch Apple. Le App, prima di essere visibili sullo Store, passano attraverso un processo di validazione e approvazione da parte della Apple stessa, per verificare che siano conformi agli standard e non dannose.

Per la realizzazione di applicazioni in questo ambiente è necessario l'utilizzo di uno specifico strumento di sviluppo, ovvero *XCode*, disponibile soltanto per i sistemi operativi *Mac OS*. XCode fornisce, tra le altre cose, i compilatori per il linguaggio Objective-C, l'ambiente runtime per eseguire programmi scritti in tale linguaggio, strumenti di analisi delle prestazioni e della memoria, simulatori delle varie piattaforme esistenti, uno strumento per creare per via grafica le interfacce e

[1] Le informazioni e le immagini di questo capitolo sono tratte da [2] e [4]

un piattaforma per la pubblicazione su App Store o per la distribuzione delle App su altri canali, insieme alla gestione dei certificati che lo rendono possibile.

A.1 iOS

iOS (fino al 2010 chiamato iPhone OS) è il sistema operativo dei dispositivi touch Apple, ed è stato sviluppato dalla Apple stessa. È un sistema UNIX, ed è basato sulla stessa tecnologia su cui è basato Mac OS X, ovvero su un kernel Mach basato su Darwin OS e su interfacce BSD[2].

L'architettura del sistema operativo è mostrata in Figura A.1. Come si può vedere, iOS ha quattro livelli: Core OS layer, Core Services layer, Media layer e Cocoa Touch layer.

Una caratteristica importante di iOS è che la memoria virtuale non supporta il paging su disco rigido; invece, si limita a eliminare le pagine in eccesso anziché effettuarne lo swapping su disco. Ciò implica che la memoria a disposizione per le applicazioni è scarsa e che, nel caso in cui si renda necessario liberarne una parte, le applicazioni vengono terminate direttamente dal sistema operativo. La terminazione avviene dopo una prima fase di segnalazione alle App, per avvisarle che la memoria sta finendo. Quanto più una applicazione occupa memoria tanto più è probabile che venga terminata. Le App vengono stoppate e rimosse dalla memoria a partire dalla più lontana dall'utente; perciò si parte da quelle in background fino, eventualmente, a terminare l'unica applicazione in foreground.

Il sistema operativo è anche responsabile dell'uso della batteria, perciò esso è anche fornito del meccanismo per lo spegnimento dello schermo, passato un certo tempo senza che l'utente vi abbia interagito. Per applicazioni particolari questa funzionalità può però essere disabilitata.

Dalla versione 4 in poi, iOS supporta anche il multitasking e quindi le applicazioni in background. Precedentemente le app venivano terminate direttamente nel momento in cui uscivano dallo stato di foreground.

[2]Si veda [4], pag. 115

Per garantire la sicurezza e l'affidabilità, ogni applicazione viene eseguita in uno spazio dedicato soggetto a restrizioni, detto *sandbox*. Ogni App ha la propria directory dove è contenuta insieme alle risorse che le sono necessarie.

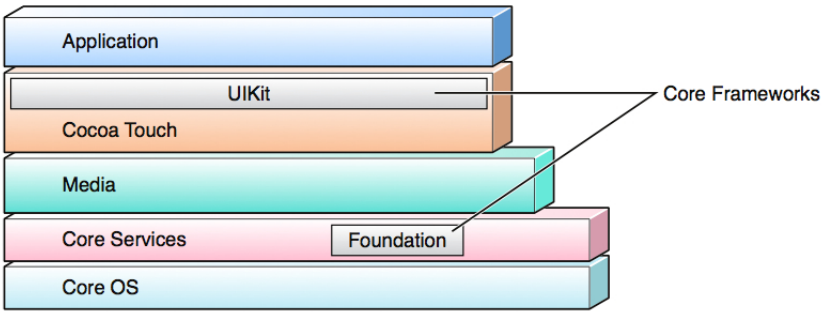


Figura A.1: L'architettura di iOS e i framework di Cocoa

A.2 L'ambiente di sviluppo Cocoa

La programmazione su iOS è basata su Cocoa. Cocoa è l'ambiente di programmazione privilegiato per lo sviluppo di applicazioni su Mac OS X e l'unico disponibile per applicazioni su dispositivi con interfacce touch[3]. È possibile utilizzare diversi linguaggi ma il principale e più importante è Objective-C. Objective-C è un linguaggio a oggetti che deriva da un'estensione di ANSI-C. È possibile alternare C e Objective-C a piacimento dentro il codice.

Cocoa è una piattaforma object-oriented e include una serie di classi scritte in Objective-C. Queste classi sono raccolte in due librerie principali, dette framework. I framework principali su iOS sono:

Foundation contiene tutti i meccanismi di base per la programmazione. Tutte le classi Cocoa che non appartengono all'interfaccia grafica o che non sono usate esclusivamente per l'interazione con l'utente appartengono a questo framework.

[3] Si veda [2], pag.13

UIKit contiene le classi per l'interfacciamento con l'utente.

Questi due framework costituiscono il Core di Cocoa.

Cocoa Touch è il layer di iOS che offre l'accesso alle funzionalità del sistema operativo alle applicazioni. In questo livello è presente l'UIKit, che appunto offre l'accesso alle classi utilizzate per creare le interfacce grafiche. Foundation invece risiede nel livello Core Services, che si interfaccia direttamente al Core del sistema operativo (si veda Figura A.1).

Cocoa offre una serie di funzionalità allo sviluppatore che voglia cimentarsi nella progettazione di applicazioni su iOS, tra cui: l'infrastruttura per gestire eventi e in generale per costruire le applicazioni; elementi per creare e gestire interfacce grafiche; la possibilità di gestire immagini, video e audio, testi, comunicazioni di rete; strumenti per gestire il multithreading al fine di aumentare le prestazioni; la possibilità di interagire con il sistema e con le altre applicazioni; la gestione dell'internalizzazione.

A.3 Il linguaggio Objective-C

Objective-C è, come detto, il linguaggio preminente nell'ambiente Cocoa per lo sviluppo di applicazioni. Questo linguaggio è un'estensione di ANSI-C in cui sono state introdotte caratteristiche sintattiche e semantiche[4] che derivano da Smalltalk per includere nel linguaggio le astrazioni dell'object-oriented programming.

Tra le caratteristiche tipiche dei linguaggi di programmazione a oggetti esso presenta l'incapsulamento, l'ereditarietà, la riusabilità e il polimorfismo, ma non sono disponibili l'ereditarietà multipla, i template e l'overloading degli operatori (che invece sono presenti ad esempio in C++).

Un altra peculiarità di Objective C è il dinamismo. Per quanto possibile, il linguaggio cerca di rinviare ogni decisione tipica della compilazione e del linking al runtime. Quindi al linguaggio non occorre soltanto un compilatore ma anche un

[4] Si veda [2], pag.14

ambiente di runtime[5]. La determinazione della classe a cui appartiene un oggetto, ovvero il typing, avviene a runtime, così come il binding, ovvero la decisione di quale metodo eseguire, e il caricamento di moduli. In particolare, il typing può avvenire a runtime utilizzando il tipo di dato `id`, descritto in Sezione A.3.1, per definire un generico oggetto la cui classe non sia ancora nota; ma il typing può essere anche statico, specificando direttamente il tipo che la variabile dovrà assumere. In questo modo, al compilatore è concesso fare qualche controllo, come verificare se una classe implementa un certo metodo oppure no. Il compilatore però segnala questi eventuali problemi solo tramite warning.

A.3.1 Gli oggetti in Objective-C

In Objective-C ogni oggetto è composto da dati e da operazioni che possono essere utilizzate per modificare questi dati. Essi vengono detti *variabili di istanza* (*instance variables*) e le operazioni sono dette *metodi* (*di istanza*). Esistono inoltre i *metodi di classe*, che sono metodi non relativi ad una istanza specifica ma all'intera classe.

Le variabili di istanza non sono normalmente visibili all'esterno della classe, perciò è necessario implementare dei metodi di accesso ad esse utilizzando le proprietà[6]. Di default, il livello di visibilità delle variabili di istanza è `@protected`, perciò è visibile dalla classe stessa e da quelle che la estendono, ma è possibile specificare anche altri livelli quali `@private`, `@public` e `@package`. Quest'ultimo livello equivale a una visibilità pari a `@public` dentro allo scope dell'immagine eseguibile della classe.

Per condividere variabili tra classi, occorre dichiararle fuori da una classe. Utilizzando la parola chiave `static`, è possibile legare una variabile a una classe e utiilizzare dei metodi di classe per manipolarla. Una variabile statica non viene ereditata ed è legata solo alla parte di classe definita nel file in cui la variabile è dichiarata.

[5] Si veda [5], pag. 13

[6] Si veda Sezione A.3.2

Ogni classe viene dichiarata specificando un'interfaccia in un file del tipo `ClassName.h` e un'implementazione normalmente in un file `ClassName.m`.

Ogni oggetto mantiene un puntatore a se stesso di nome `self`. `super`, invece, è un flag che segnala di cercare l'implementazione dei metodi nella superclasse.

All'interno dell'ambiente Cocoa, ogni oggetto discende dalla classe `NSObject`, formando una gerarchia. Questo vale sia per le classi definite dallo sviluppatore, sia per quelle presenti nei due framework di Cocoa, UIKit e Foundation. `NSObject` definisce l'interfaccia di base per ogni oggetto del linguaggio. A questo scopo, `NSObject` oltre che una classe è anche un protocollo, per permettere di realizzare gerarchie di classi indipendenti ma che comunque siano conformi allo standard degli oggetti di questo framework. Ovviamente è più semplice mantenere la classe `NSObject` come radice della propria gerarchia di classi, poiché essa contiene già tutti i meccanismi per il comportamento di un oggetto in quanto tale.

I metodi principali di questa classe sono:

alloc questo metodo alloca la memoria per l'oggetto e lo fa puntare alla classe di appartenenza.

init inizializza le variabili di istanza della classe.

class, superclass restituiscono un oggetto di tipo `Class` che identifica rispettivamente la classe di appartenenza e la superclasse dalla quale eredita la classe di cui l'oggetto è istanza.

isKindOfClass, isMemberOfClass, isSubclassOfClass permettono di identificare a che classe appartiene l'oggetto in esame.

conformsToProtocol permette di verificare se una classe è conforme a un certo protocollo[7].

isEqual è utilizzato per effettuare confronti tra oggetti.

description è usato per associare una stringa di descrizione dell'oggetto.

[7] Si veda Sezione A.3.2

forwardInvocation permette di inoltrare un messaggio ricevuto verso un altro oggetto.

performSelector è un metodo utilizzato per inviare messaggi dopo un certo intervallo di tempo, e per scambiare messaggi tra thread.

Quando una nuova classe viene creata, essa deve essere definita come sottoclasse di un'altra classe esistente, a meno che non si stia definendo la classe radice di una nuova gerarchia.

In questo linguaggio, `NSObject` è un esempio di classe astratta, ma in realtà non esistono sintassi particolari per indicare che una classe non è concreta, né impedisce la creazione di un'istanza di una classe astratta.

Ogni oggetto contiene una struttura dati. Il principale dato contenuto in esso è il puntatore `isa`, che definisce di quale classe l'oggetto è istanza. Esso collega l'oggetto alla classe di appartenenza puntando all'oggetto che definisce la classe, essendo la classe essa stessa un oggetto.

Le interazioni tra oggetti sono definite da uno scambio di messaggi tra gli stessi. Un messaggio specifica il nome di un selettore, il quale serve per selezionare il metodo dell'oggetto da eseguire. Il concetto di nome di selettore, spesso chiamato per brevità selettore esso stesso, è un tipo del linguaggio che si specifica con la parola chiave `SEL`. Oltre al selettore, il messaggio che viene inviato da un oggetto a un altro normalmente contiene anche una serie di parametri che si specificano insieme alla parola chiave che li definisce. Il tutto è racchiuso tra parentesi quadre, con una struttura del tipo:

```
NSClassName *variable =
[ receiver selectorName:param1 keyword2:param2 ...]
```

In questo caso si prevede anche un valore di ritorno di tipo `NSClassName`, ma non è necessario; il tipo di ritorno può anche essere `void`.

Il selettore completo di questo messaggio sarà `selectorName:keyword2:...` e così via, poiché il nome del selettore comprende anche i nomi dei parametri e il simbolo : che separa il nome del parametro dal parametro stesso. Il nome del primo parametro è sottinteso e segue direttamente il nome del metodo. Se il metodo

non deve ricevere alcun parametro in ingresso, il nome completo del selettore sarà semplicemente `selectorName`. Si parla perciò di scambio di messaggi tra oggetti per indicare la chiamata a metodi.

Altri tipi di dato peculiari del linguaggio sono:

id è un tipo di dato dinamico che può contenere il riferimento a un oggetto di qualunque classe, in virtù del tipying dinamico del linguaggio.

Class è il tipo di dato che rappresenta una classe. Ogni classe è istanza di tipo `Class`.

A.3.2 Estensioni in Cocoa

In Cocoa, il linguaggio presenta quattro tipi di estensioni che ne aumentano la potenza e la flessibilità.

Categories le categorie permettono di aggiungere metodi ad una classe senza farne il subclassing, è perciò un metodo alternativo all'ereditarietà. Inoltre permettono di raggruppare i metodi in base al loro utilizzo, o per esigenze di sviluppo. Tutti i metodi aggiunti attraverso le categorie sono automaticamente ereditati dalle sottoclassi. Le categorie però non permettono di aggiungere variabili di istanza alla classe.

Protocols i protocolli sono un modo per definire una lista di metodi e proprietà che un metodo può scegliere o meno di implementare; in questo senso assomigliano alle interfacce del linguaggio Java. Anche questo metodo può essere un'alternativa al subclassing, permettendo di ovviare alla mancanza dell'ereditarietà multipla, poiché una classe può adottare più protocolli senza perdere la sua identità. I protocolli possono essere formali o informali. Nel primo caso la classe che adotta il protocollo deve obbligatoriamente implementare tutti i metodi definiti nel protocollo stesso; nel secondo caso può scegliere quali metodi implementare. I protocolli permettono l'utilizzo di oggetti anonimi, rendendo effettivamente utile il typing dinamico, oltre ad

aggiungere la possibilità di mandare messaggi a oggetti di altre applicazioni, detti *oggetti remoti*.

Declared Properties l'utilizzo delle proprietà permette di definire in automatico i metodi che forniscono l'accesso alle variabili di istanza delle classi[8], ovvero setter e getter.

Fast Enumeration L'enumerazione rapida definisce una sintassi per il ciclo for apposita per l'iterazione di strutture dati quali array (`NSArray`), set (`NSSet`), ecc[9].

A.4 Gestione della memoria

Nell'ambiente Cocoa è possibile gestire la memoria in due modi: con un garbage collector che si occupi di liberare la memoria occupata da oggetti che non sono più referenziati da nessuno, oppure gestendo i riferimenti agli oggetti creati. Il garbage collector non è disponibile in iOS; la gestione dei riferimenti può essere automatica o manuale. La gestione automatica permette di lasciare gestire il ciclo di vita degli oggetti al compilatore, ma la gestione manuale della memoria resta comunque un'alternativa percorribile e altamente consigliabile in ambienti che normalmente hanno vincoli hardware stringenti.

Ogni oggetto mantiene il proprio *retain count*, ovvero il numero di oggetti che possiedono un riferimento all'oggetto stesso. Perché l'oggetto venga correttemente deallocato e la memoria liberata, il *retain count* deve raggiungere lo zero. L'oggetto che crea un altro oggetto con la `alloc` è responsabile di rilasciarlo nel momento in cui non serva più. Se un altro oggetto ottiene un riferimento all'oggetto allocato tramite ad esempio una factory, non è quindi responsabile del ciclo di vita di quella variabile, ma nel caso in cui voglia essere certo che l'oggetto non sia deallocato prima di un successivo utilizzo, deve aumentare il *retain count* e poi decrementarlo quando l'oggetto non gli serve più.

[8] Le proprietà sono definite in Objective-C 2.0

[9] L'enumerazione rapida è definita in Objective-C 2.0

Il rilascio dell'oggetto può essere anche eseguito in modo ritardato rispetto all'effettivo utilizzo dello stesso, mandando un messaggio `autorelease` all'oggetto. In questo caso, il *retain count* dell'oggetto viene decrementato quando l'`NSAutoreleasePool` a cui appartiene l'oggetto viene rilasciato. Un `NSAutoreleasePool` è un tipo di dato che identifica un insieme di oggetti definiti all'interno di uno scope. In applicazioni normali, l'`NSAutoreleasePool` è uno solo e relativo al main thread, fornito e gestito direttamente dal framework, perciò di fatto l'oggetto a cui venga inviato un messaggio di tipo `autorelease` viene rilasciato soltanto alla chiusura dell'applicazione. Un `NSAutoreleasePool` può però essere creato manualmente in altri punti dell'applicazione, ed è obbligatorio crearne uno al momento della creazione di un thread. In iOS, l'approccio dell'`autorelease` è normalmene sconsigliato, poiché generalmente significa non rilasciare la memoria in un tempo ragionevole per ambienti con poche risorse.

Per gestire questo aspetto, sono definiti nella classe `NSObject` (e quindi ereditati da qualunque oggetto la estenda) i seguenti metodi:

alloc questo metodo alloca la memoria per l'oggetto e imposta il puntatore `isa` dell'oggetto perché punti alla classe di appartenenza (a runtime) e inizializza il *retain count* a 1.

retain incrementa il *retain count*.

release decrementa il *retain count*.

autorelease decrementa il *retain count* al momento del rilascio dell'`NSAutoreleasePool` che contiene l'oggetto.

retainCount permette di ottenere il valore corrente del *retain count*.

dealloc è un metodo che occorre implementare per rilasciare le variabili di istanza della classe che non sono tipi primitivi, quindi che necessitano del decremento del *retain count*. Terminato il rilascio delle variabili di istanza della classe occorre chiamare il metodo `dealloc` della superclasse. Questo metodo viene chiamato in automatico dal gestore della memoria quando il *retain count* si azzera.

A.5 La progettazione di applicazioni

Un'App che gira su iOS perciò dovrebbe essere scritta in Objective-C (per la maggior parte), utilizzare le librerie Foundation e UIKit di Cocoa e tenere conto dell'ambiente vincolato in cui deve essere eseguita. Successivamente l'App così creata può essere pubblicata su App Store o distribuita in altro modo, ma sempre attraverso certificati e autorizzazioni approvati dalla Apple stessa.

A.5.1 Struttura di un'applicazione

La Figura A.2 mostra la struttura di una generica applicazione progettata per l'esecuzione su iOS.

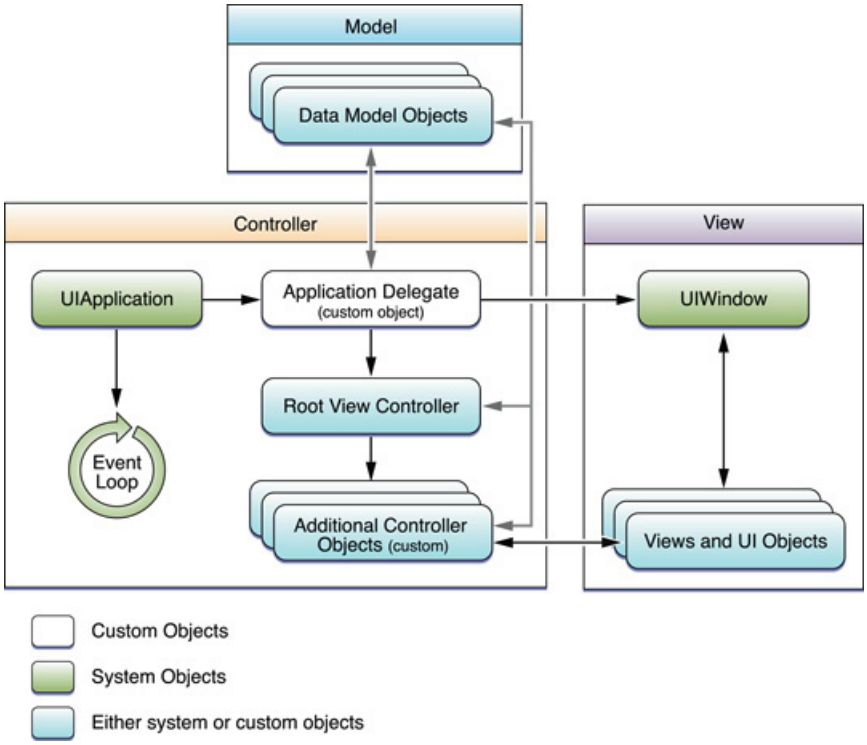


Figura A.2: Struttura di una generica applicazione su iOS

Come si può vedere, il pattern Model-View-Controller domina la scena, al punto da figurare nella struttura complessiva di qualunque App.

UIKit è la libreria responsabile di fornire i componenti base per la costruzione di applicazioni. Gli oggetti presenti nel framework permettono di gestire eventi, costruire interfacce e interagire con il resto del sistema.

Gli oggetti definiti dall'utente vanno a inserirsi in una struttura ben precisa. Alcuni componenti sono oggetti definiti direttamente dal sistema e non modificabili (in Figura A.2, in verde), e sono:

UIApplication rappresenta l'applicazione. Si occupa di gestire gli eventi e le interazioni tra componenti di alto livello.

UIWindow esiste una sola `UIWindow` per ogni schermo occupato o occupabile dall'App. Contiene tutte le View.

Alcuni componenti possono essere invece forniti da UIKit o creati dall'utente, normalmente facendo il subclassing di altri oggetti che fanno parte del framework.

Application Delegate il delegate è un oggetto creato dal programmatore che viene caricato nella funzione `UIApplicationMain`[10]. Questo oggetto si occupa di gestire i cambi di transizione dell'App, ovvero, ad esempio, del passaggio da background a foreground. Esso è una sottoclasse di `UIResponder`[11]. Se la UIApplication non è in grado di gestire un evento, esso viene inviato all'App Delegate.

View Le View sono tutte istanze della classe `UIView`. Una View può essere una schermata mostrata all'utente o una porzione di essa. Una View è sempre contenuta in una `UIWindow` e può essere padre o figlia di una ulteriore View, formando una gerarchia. Una View normalmente è contenuta in un file, detto *nib*, che viene caricato dal View Controller a cui è associata. Esistono particolari tipi di View, chiamati *Control*, che generano eventi specifici gestiti secondo il meccanismo Target-Action[12].

[10] Si veda Sezione A.5.2
[11] Si veda Sezione A.6
[12] Si veda Sezione A.6

View Controller Un View Controller normalmente è associato a una View e si occupa di gestire le interazioni tra essa e il resto del sistema. I View Controller sono istanze della classe `UIViewController`. Un View Controller è responsabile della creazione e distruzione delle View che possiede e ne gestisce la presentazione sullo schermo.

Data Model Utilizzando le classi del framework che fanno parte di questo blocco è possibile rappresentare svariati tipi di dati, di rappresentazioni e di memorizzazione degli stessi (database, file system, ...).

A.5.2 Esecuzione di un'applicazione

Un'applicazione può trovarsi in più stati diversi e deve comportarsi di conseguenza. Essendo in un ambiente fortemente vincolato, l'App deve risparmiare più risorse possibile, pena la terminazione.

L'App, in un qualsiasi momento, può essere in uno dei seguenti stati:

Non in esecuzione quando non è stata ancora lanciata.

Non attiva quando l'App è in foreground ma non sta ricevendo eventi.

Attiva se l'App è in foreground e riceve eventi.

In background nel momento in cui l'App non è in foreground ma sta eseguendo codice.

Sospesa se l'App è in background ma non esegue codice. Se l'App è in questo stato, può essere terminata dal sistema operativo in qualsiasi momento nel caso in cui occorra fare pulizia della memoria, senza preavviso.

In Figura A.3 è mostrata la schematizzazione del lancio di un'applicazione. Subito dopo il lancio, l'App va in foreground. L'avvio corrisponde alla transizione tra gli stati *Non in esecuzione* e *Attiva*.

Come accade normalmente in un ambiente C o simili, il punto d'ingresso dell'applicazione è la funzione `main()`. Il `main` di una qualunque App eseguita su

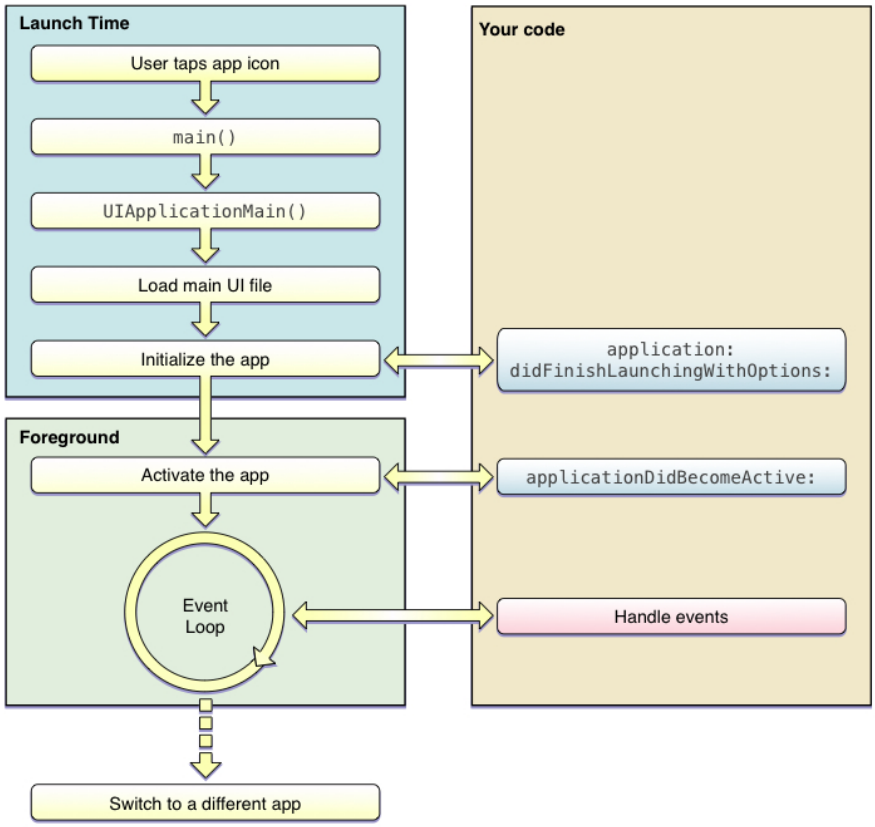


Figura A.3: Lancio ed esecuzione di un'applicazione. Fonte: iOS App Programming Guide

iOS crea l'`NSAutoReleasePool` dell'applicazione e lancia la funzione `UIApplicationMain` della UIApplication, specificando come parametro anche quale è la classe a cui appartiene l'App Delegate. `UIApplicationMain` inizializza l'applicazione, caricando ad esempio l'interfaccia principale. Terminato il caricamento viene chiamato l'apposito metodo dell'App Delegate, personalizzabile dallo sviluppatore per inserire una serie di operazioni da eseguire terminato il lancio dell'applicazione. A questo punto, l'App è in foreground e si mette in attesa di eventi, siano essi input provenienti dall'utente o dal sistema. L'App può poi cambiare stato e/o essere terminata, sia dal sistema per liberare memoria, sia da un utente che abbia la possibilità di interagire con il pannello di gestione delle

applicazioni attive. È possibile personalizzare un metodo dell'App Delegate per stabilire cosa fare prima che l'App venga terminata.

A.5.3 Interfacce Grafiche

Le interfacce grafiche possono essere costruite da codice, ma l'ambiente di sviluppo XCode mette a disposizione uno strumento, chiamato *Interface Builder*, che permette di realizzare le interfacce utilizzando strumenti grafici. Esso permette inoltre di realizzare automaticamente i collegamenti con il codice, associando ai Control gli eventi da gestire, i metodi da eseguire in risposta a qualche evento[13], i *delegate* e i *View Controller*. Il risultato di questa costruzione viene salvato in un file detto *nib*, di tipo *.xib*, che viene caricato da programma al momento in cui si renda necessario il disegno dell'interfaccia grafica associata.

A.5.4 Gestione degli eventi

Subito dopo l'avvio, un'applicazione entra in un *Run Loop*, detto *Main Run Loop* (si veda Figura A.4), responsabile della ricezione e relativo dispatching di eventi. Il *Main Run Loop* viene eseguito sul thread principale dell'App, in modo tale da garantire il processamento degli eventi in ordine di arrivo.

Gli eventi, a mano a mano che arrivano, vengono inoltrati al componente che è in grado di gestirli.

Questi eventi sono sottoclassi di `UIEvent`. Essi possono essere raggruppati nelle seguenti categorie principali:

Multi-Touch Event generati a fronte del tocco dell'utente sullo schermo. Il tocco dell'utente può anche coinvolgere più di un dito alla volta, perciò gli eventi sono detti Multi-Touch. In questi casi esistono degli strumenti di ausilio del framework per riconoscere e gestire diversi tipi di movimenti delle dita sullo schermo.

[13] Utilizzando il meccanismo *Target-Action*, si veda Sezione A.6

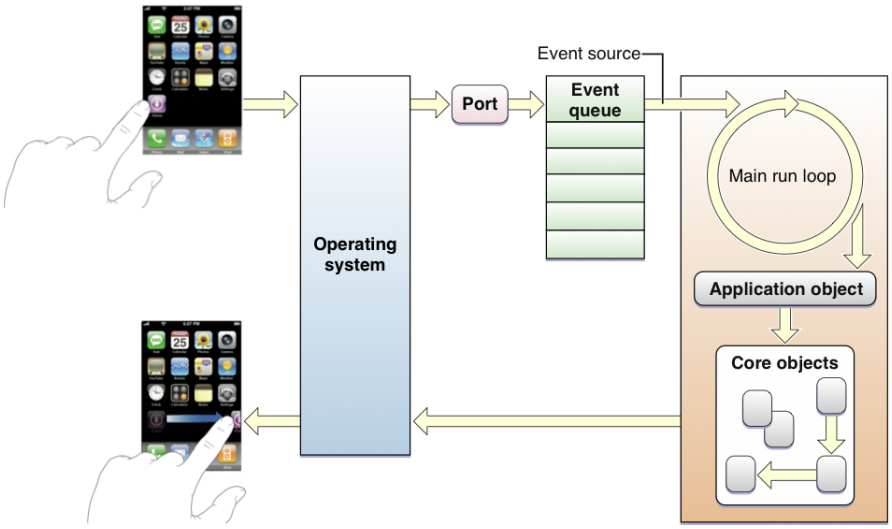


Figura A.4: Gestione degli eventi

Motion Event generati a fronte del movimento del device. Questi tipi di eventi sono generati dagli accelerometri e dal giroscopio.

Remote Event generati a fronte di eventi relativi a periferiche collegate al dispositivo, quali ad esempio il collegamento delle cuffie.

Quando un evento accade, è la piattaforma a gestirne la trasformazione in dati comprensibili dalle applicazioni (ovvero un oggetto `UIEvent`) e a consegnarlo all'applicazione che è attualmente in foreground. Gli eventi più interessanti sono quelli di tipo Touch. Quando un utente tocca lo schermo, un apposito oggetto viene creato e passato all'`UIApplication`, la quale lo passa alla `UIWindow` corrente. Per stabilire qual'è la View che l'utente ha virtualmente toccato, viene effettuato un test geometrico (detto *Hit-test*) ricorsivo sull'albero di View che compone la Window fino a che non si trova la View che contiene la porzione di schermo toccata dall'utente. Tutto ciò accade attraverso l'invocazione di appositi metodi.

A.6 Astrazioni e Design Pattern

Analizziamo ora nel dettaglio i design pattern e le astrazioni presenti, esplicitamente e implicitamente, nel linguaggio Objective-C, nel framework Cocoa e nell'ambiente di sviluppo per iOS. Ciò al fine di estrarre informazioni interessanti per lo sviluppo platform-independent di applicazioni.

A.6.1 Model View Controller

Come detto, il Model View Controller è un design pattern pervasivo all'interno dell'ambiente di programmazione di iOS, al punto da caratterizzare la struttura complessiva di qualunque applicazione. I building block con cui si costruisce un'App fanno parte di una delle tre categorie Model, View e Controller, e le interazioni tra di esse sono regolate grazie all'utilizzo di questo pattern. Come visto in Figura A.2, quindi, la struttura di una generica applicazione è divisa nelle tre parti Model (azzurro), View (viola) e Controller (arancione). Le tre parti sono concepite per separare, sia concettualmente che praticamente, i dati e la loro visualizzazione all'utente.

Model

La parte Model dell'applicazione rappresenta i dati con cui essa lavora. In iOS, ci sono oggetti con cui si può lavorare (Core Data), già concepiti per essere salvati su un qualche tipo di supporto, e a seconda del tipo di supporto la piattaforma fornisce già una serie di funzionalità per il salvataggio, modifica, ecc[14]. Gli oggetti vengono perciò già in questa fase modellati insieme alla logica di salvataggio su disco fisso, database, rete, ecc. In questo modo, nel Model abbiamo già un livello di astrazione più alto del classico Model-View-Controller, che permette di alleggerire il Controller da funzioni quali la gestione di file, database...

I building block che fanno parte di questa sezione sono:

[14] Si veda Sezione A.6.2

- Data Model Objects: permettono di modellare dati generici (quali stringhe, numeri...). Questi oggetti possono anche essere strutturati ed avere associata una generica logica di salvataggio su supporto.
- Document Objects: modellano il contenuto di file e la loro gestione in modo trasparente.

View

La View è la parte di applicazione che si occupa di interagire con l'utente, normalmente mostrandogli dati e permettendogli di modificarli. In iOS questa sezione contiene tutti i building block fatti per essere mostrati all'utente. Le View in questo caso non sono componenti completamente passivi, ma possono essere in grado di rispondere ad eventi.

- `UIWindow`: come già detto, una `UIWindow` è unica per ogni schermo che l'applicazione può occupare (quindi generalmente è unica). Contiene tutte le View che appartengono all'applicazione.
- `UIView`: sono componenti attivi e passivi in grado di interagire con l'utente, dai bottoni alle schermate.

Controller

- UIApplication: rappresenta l'applicazione e attende gli eventi in input.
- Application Delegate: il *delegate* dell'applicazione, gestisce le transizioni di stato.
- View Controllers: componenti associati a, e responsabili di, una o più view, ne gestiscono le interazioni con il resto del sistema.

In alcuni casi, la visualizzazione all'utente può, e deve, essere dipendente dai dati, ma il Controller in questo caso funziona comunque da intercapedine.

Il concetto di View Controller unisce in realtà View e Controller, poiché è fortemente dipendente dalla View ma svolge comunque una funzione di controllo.

In questa parte di applicazione è ovviamente possibile aggiungere altri oggetti per realizzare un vero e proprio controller centralizzato, utilizzando ad esempio il concetto di *Singleton* o utilizzando un oggetto condiviso raggiungibile attraverso l'*Application Delegate*.

A.6.2 Object Modeling

Il concetto di Object Modeling permette di realizzare un modello dei dati coerente con la sua stessa memorizzazione su supporto, qualunque esso sia (file, cloud, database...). All'interno del framework Cocoa esiste una tecnologia di nome *Core Data*, che permette la creazione di un grafo strutturato di dati e di gestirne il salvataggio su e il recupero da file, database o rete in maniera trasparente. Questo grafo è composto da oggetti di nome *Entity*, contententi degli *Attributes*. Le *Entity* sono collegate tra di loro da *Relationships*.

A.6.3 Delegate

Il Delegate è un meccanismo ampiamente usato in svariate occasioni nel framework Cocoa. L'adozione del Delegate prevede che un oggetto ne encapsuli un altro allo scopo di utilizzarlo per eseguire determinate operazioni. Normalmente l'oggetto contenente non è responsabile della creazione dell'oggetto a cui delegare i compiti, ma si limita a mantenerne un riferimento, senza conoscerne i dettagli. Per la realizzazione del meccanismo, è sufficiente che l'oggetto sia conforme al protocollo adatto. Al momento opportuno, l'oggetto delegante manda un messaggio conforme al protocollo del Delegate all'oggetto delegato e attende che l'operazione venga eseguita.

A.6.4 Responder Chain

All'interno della piattaforma viene utilizzata la cosiddetta catena di responsabilità per la gestione di eventi. In questo meccanismo si hanno una serie di oggetti collegati tra di loro tramite l'UIKit. La piattaforma manda al primo oggetto della catena, detto normalmente *First Responder*, un messaggio perché esegua una

certa operazione. Se l'oggetto è in grado di rispondere al messaggio, esegue l'operazione, altrimenti la piattaforma inoltra il messaggio al successivo elemento della catena, e così via fino alla fine.

Gli oggetti che fanno parte della catena sono sottoclassi della classe `UIResponder` del framework Cocoa, e sono utilizzati per rispondere agli eventi generati dall'interfaccia. Ad esempio, `UIWindow` e `UIApplication` ereditano da `UIResponder`, e lo stesso vale per le View. Queste classi, che fanno parte del framework, possiedono un metodo `nextResponder()` che è in grado di determinare l'elemento successivo della catena.

La *Responder Chain*, in occasione ad esempio di un evento di tipo Touch, è formata dalle `UIView` coinvolte nell'evento, e il messaggio che viene inoltrato lungo la Responder Chain contiene l'oggetto `UIEvent` associato all'evento stesso. La prima View è quella che è stata riconosciuta come quella direttamente coinvolta nel tocco dell'utente, seguita dalle View che la contengono con gli eventuali View Controller associati, arrivando alla `UIWIndow` e infine alla `UIApplication`, secondo il seguente schema[15]:

- Si comincia dalla View che è stata riconosciuta come quella che contiene il tocco dell'utente e dal suo View Controller, se presente;
- Si risale la gerarchia delle View con gli eventuali View Controller;
- Si prova con la `UIWindow` che contiene la View che ha generato l'evento e il suo *delegate*;
- Se la Window corrente non è quella principale si giunge alla *Main Window* e al suo *delegate*;
- Si prova infine con l'oggetto `UIApplication` e con l'*Application Delegate*.

Se nessuno di questi elementi è in grado di gestire il messaggio associato all'evento, non viene eseguito alcun pezzo di codice e l'evento viene di fatto ignorato. In generale, il primo oggetto (di tipo `UIResponder`, normalmente una View) che

[15] Si veda [2], pag. 182

dovrebbe essere in grado di reagire ad un evento secondo la piattaforma, e da cui parte la *Responder Chain*, è detto *First Responder*. Nel caso degli eventi Touch questo *First Responder* non esiste e l'oggetto da cui parte la *Responder Chain* è la View che contiene il tocco dell'utente, determinata grazie all'*Hit-test*.

A.6.5 Target-Action

Il meccanismo Target-Action è responsabile del collegamento tra un evento generato dall'interfaccia e le istruzioni che il programmatore vuole siano eseguite quando quell'evento si verifica. Esso è usato dal framework Cocoa per la comunicazione tra un Control che riceve l'evento e un altro oggetto che normalmente contiene il codice da eseguire. L'oggetto è detto *Target* e il messaggio che gli viene inviato dal Control è l'*Action*. Normalmente l'oggetto *Target* è contenuto nel Control, ma può essere anche determinato dinamicamente settando il *Target* a `nil` e utilizzando a runtime il meccanismo della Responder Chain. L'azione consiste nel selettore da inviare all'oggetto *Target*, ed è perciò il nome del metodo che esso realizza per rispondere alla *Action*. Il metodo ha normalmente un solo parametro di tipo `id` che identifica colui che ha inviato il messaggio, ovvero il *sender*. Il metodo non ha tipo di ritorno, ma a volte può essere utilizzata la keyword `IBAction` per segnalare a XCode che quello è il metodo che si è indicato nell'Interface Builder come metodo da eseguire in risposta a un certo evento.

Tutto questo è formalizzato nella classe `UIControl`. Ogni oggetto dell'interfaccia grafica in grado di generare eventi è sottoclasse di `UIControl`. Il ruolo di questa classe è di fornire l'interfaccia e l'implementazione di base del meccanismo *Target-Action*, perciò permette di mantenere traccia dei *Target* e delle *Action* associati a diversi eventi[16]. Assegnare una coppia Target-Action per un certo evento è possibile sia da *Interface Builder*, che permette di salvare le associazioni dentro un file nib, sia da codice utilizzando il metodo con selettore `addTarget:action:forControlEvents:`.

[16]UIControl Class Reference, http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIControl_Class/Reference/Reference.html

Quando l'utente tocca un Control, ad esempio un bottone, la piattaforma genera l'evento come oggetto di classe `UIEvent`, e l'applicazione sa già che l'evento deve essere inviato al Control in questione. Questo Control invierà l'apposita Action al Target designato per l'evento.

A.6.6 Notifications

Le notifiche sono un design pattern molto utilizzato in Cocoa e in pratica è un'implementazione del pattern *Observer*. Esso prevede un broadcasting di messaggi nel caso in cui si verifichi un certo evento. Gli *observers* si registrano ad un *Notification center* per un evento di interesse. Il *Notification center* si occupa poi di notificare l'avvenuta occorrenza dell'evento in modo sincrono (approccio classico, poiché prevede che ad ogni *observer* venga inviato un certo messaggio che scatena l'esecuzione di un metodo) oppure in modo asincrono (tramite l'utilizzo delle *Notification queues*, che prevede diverse modalità di segnalazione delle notifiche, anziché del *Notification center*). Lo schema prevede che a fronte di una notifica inviata da una certa sorgente, l'`NSNotificationCenter` invii degli oggetti `NSNotification` agli *observers* utilizzando il metodo indicato dagli observer stessi all'atto dell'iscrizione al Notification Center.

Appendice B

Lo sviluppo di applicazioni su Android

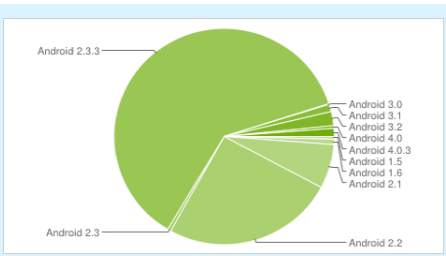
Android è una piattaforma software sviluppato da Google Inc. che include un sistema operativo, un middleware per l'integrazione sui dispositivi e alcune applicazioni di base[1]. Esistono diverse versioni di Android e la più recente è la 4.0, Ice Cream Sandwich. Le varie versioni coesistono normalmente sul mercato e sui dispositivi in uso e alcune sono open source.

Sul mercato sono disponibili numerosi dispositivi che montano un firmware Android. I dispositivi sono di diversi produttori quali Samsung, Motorola ecc. I vari produttori di dispositivi mobili, nel momento in cui scelgono di adottare Android come sistema operativo, devono anche scegliere una versione del sistema operativo da supportare attraverso il proprio firmware.

Quella di implementare una versione del firmware che supporti una versione più recente del sistema operativo è una scelta del produttore del dispositivo. Un cellulare perciò potrebbe montare Android 1.5 anche se è recentemente uscita la versione 4.0 della piattaforma. Occorre tenere conto di questa caratteristica nello sviluppo di applicazioni, se si vuole avere il bacino di utenza più ampio possibile.

Infatti, le App vengono distribuite secondo una precisa disciplina. L'Android Market filtra automaticamente le App in base alle caratteristiche del dispositivo su

[1] Le informazioni e le immagini di questo capitolo sono tratte da [1]



Platform	Codename	API Level	Distribution
Android 1.5	Cupcake	3	0.4%
Android 1.6	Donut	4	0.8%
Android 2.1	Eclair	7	6.6%
Android 2.2	Froyo	8	25.3%
Android 2.3 - Android 2.3.2	Gingerbread	9	0.5%
Android 2.3.3 - Android 2.3.7		10	61.5%
Android 3.0	Honeycomb	11	0.1%
Android 3.1		12	1.1%
Android 3.2		13	2.1%
Android 4.0 - Android 4.0.2	Ice Cream Sandwich	14	0.4%
Android 4.0.3		15	1.2%

Data collected during a 14-day period ending on March 5, 2012

Figura B.1: Distribuzione delle versioni di Android al 5 marzo 2012

cui è installata l'App del Market stesso. Android Market è il mezzo privilegiato per regolarizzare la distribuzione delle App nel mondo Android, che però possono essere installate in un dispositivo anche utilizzando direttamente il file con estensione *.apk* che costituisce il pacchetto (firmato o meno) che contiene interamente l'applicazione. Se si utilizza Android Market direttamente tramite la App dedicata preinstallata su tutti i dispositivi Android, si potranno visualizzare soltanto le applicazioni compatibili con il proprio cellulare o tablet, in termini principalmente di versione del sistema operativo e di dimensione dello schermo ma anche di altri requisiti hardware più o meno vincolanti.

Per quanto riguarda lo sviluppo di applicazioni, dal sito *android.developer.com* è possibile scaricare liberamente il pacchetto che contiene tutti gli strumenti, ovvero l'Android SDK. Per rendere più semplice l'utilizzo di questi strumenti è possibile integrarli nell'IDE Eclipse attraverso un plugin apposito[2]. Grazie a questo ambiente di programmazione integrato è possibile utilizzare le librerie aggiuntive di Android per lo sviluppo in Java, utilizzare simulatori con svariate caratteristiche hardware e software per il testing, impacchettare e firmare le applicazioni.

[2]Utilizzando come Location l'indirizzo https://dl-ssl.google.com/android/eclipse/ nell'area Repository di Eclipse

B.1 Il sistema operativo

Android è basato sul kernel Linux versione 2.6. Il sistema operativo è multi-utente e ogni applicazione rappresenta un utente.

Siccome il sistema è basato sul linguaggio Java, i programmi sono eseguiti su una macchina virtuale. La macchina virtuale che viene utilizzata su Android non è la classica Java Virtual Machine, ma una macchina virtuale ottimizzata per eseguire su dispositivi con scarse risorse hardware. Questa è detta *Dalvik Virtual Machine*. La Dalvik VM è dotata di un garbage collector perciò la gestione della memoria non deve essere direttamente gestita dal programmatore.

Ogni applicazione viene eseguita in un processo a sé stante, con una propria istanza della Dalvik VM. Ciò permette di implementare il meccanismo del minimo privilegio, così che le applicazioni abbiano accesso solo alle risorse che gli sono esplicitamente concesse.

La struttura della piattaforma è riportata in Figura B.2.

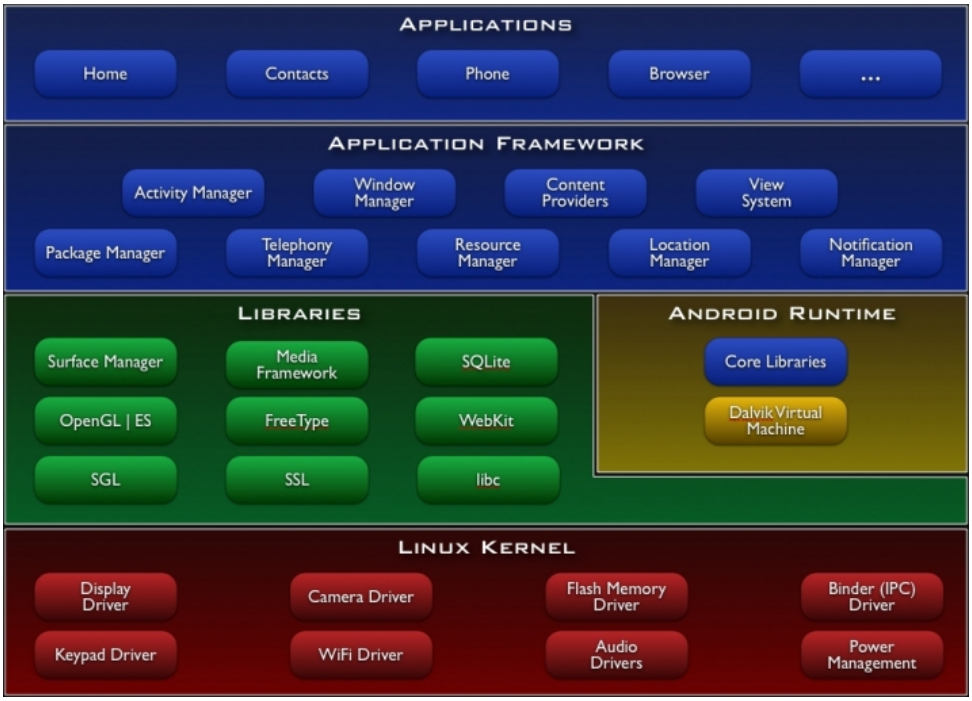


Figura B.2: L'architettura di Android

Il livello più basso è costituito dal kernel Linux che offre anche i driver per l'hardware sottostante, permettendo di astrarre dalle caratteristiche specifiche del dispositivo.

Le librerie sono a disposizione del programmatore e permettono di sfruttare le caratteristiche della piattaforma, quali l'uso di SQLite, le OpenGL, supporto per i file multimediali, l'uso della telefonia e di tutti i componenti hardware di cui il dispositivo può essere dotato.

L'ambiente runtime di Android permette di implementare il meccanismo per il quale ogni nuova applicazione viene avviata in un processo a parte con la propria istanza della Dalvik Virtual Machine.

L'application framework contiene tutti i componenti di base per lo sviluppo di applicazioni.

Infine nell'ultimo livello abbiamo le applicazioni, che possono essere già presenti nella piattaforma o sviluppate successivamente da terzi.

B.1.1 Versioni

Come mostrato in Figura B.1, ci sono diverse versioni di Android in circolazione. Si riportano brevemente le varie versioni della piattaforma esistenti, con i corrispondenti *API level*, ovvero l'intero che la identifica univocamente. Esso viene utilizzato all'interno delle applicazioni per indicare quale è la piattaforma su cui un'App è stata progettata per funzionare.

Android 1.0 (Base) API level 1 - ottobre 2008. La piattaforma di base. Tutte le successive release di Android sono incrementali a partire da questa e aggiungono nuove caratteristiche al sistema.

Android 1.1 (Base_1_1) API level 2 - febbraio 2009. Versione minore.

Android 1.5 (Cupcake) API level 3 - maggio 2009. È la versione di base per lo sviluppo e la versione minima in circolazione.

Android 1.6 (Donut) API level 4 - settembre 2009. Versione minore. Prevede, tra le altre cose, un aggiornamento del kernel.

Android 2.0 (Eclair) API level 5 - novembre 2009.

Android 2.0.1 (Eclair_0_1) API level 6 - dicembre 2009. Versione minore.

Android 2.1 (Eclair MR1) API level 7 - gennaio 2010. Versione minore.

Android 2.2 (Froyo) API level 8 - giugno 2010. Versione minore.

Android 2.3 (Gingerbread) API level 9 - novembre 2010. Prevede, tra le altre cose, un aggiornamento del kernel.

Android 2.3.3 (Gingerbread MR1) API level 10 - febbraio 2011. Versione minore.

Android 2.3.4 API level 10 - maggio 2011. Versione di manutenzione. Non ci sono cambiamenti nelle API rispetto alla versione 2.3.3.

Android 3.0 (Honeycomb) API level 11 - febbraio 2011. Questa versione è stata pensata appositamente per i tablet. Introduce diversi cambiamenti nella piattaforma, come l'aggiunta di *Fragments* e dell'*Action Bar*.

Android 3.1 (Honeycomb MR1) API level 12 - maggio 2011.

Android 3.2 (Honeycomb MR2) API level 13 - luglio 2011.

Android 4.0 (Ice Cream Sandwich) API level 14 - ottobre 2011. Questa versione della piattaforma, che esiste nelle versioni 4.0, 4.0.2 e 4.0.3, introduce numerosi cambiamenti ed è sviluppata per integrare le funzionalità offerte nello stesso modo su smartphone e tablet.

Android 4.0.3 (Ice Cream Sandwich MR1) API level 15 - dicembre 2011.

B.2 Java, Android e le applicazioni

Per creare applicazioni su Android si parte dal linguaggio Java, che viene poi esteso con una serie di librerie apposite per la programmazione su mobile

e per gestire una serie di aspetti, quali ad esempio dispositivi hardware integrati e telefonia.

Un'applicazione Android è formata da una certo numero di componenti. I componenti sono 4, e sono:

Activity è un pezzo di interfaccia con il relativo codice applicativo.

Service rappresenta un servizio eseguito in background.

Content Provider contiene dati e fornisce l'interfaccia per accedervi.

Broadcast Receiver riceve notifiche di sistema e di altre applicazioni.

Essi costituiscono i nuovi concetti introdotti nella piattaforma. Questi concetti non fanno parte nativamente del linguaggio Java ma sono inclusi tramite librerie. Un ulteriore concetto è quello dell'*Intent*, che serve come mezzo di comunicazione tra Activity, Service e Broadcast Receiver.

I vari componenti scelti per la propria applicazione devono essere inseriti nel file *Manifest* che viene poi incluso nell'apk e che contiene le informazioni che costituiscono l'interfaccia pubblica dell'App.

Qualunque applicazione può far partire un componente di un'altra applicazione o dell'applicazione stessa. Ciò accade attraverso gli Intent. Gli Intent sono una forma di messaggio che viene recapitato al sistema per segnalare che si vuole far partire un certo componente. Infatti, siccome le applicazioni vengono eseguite in processi separati e non possono uscire dalla loro macchina virtuale Dalvik, non è possibile far partire un componente di un'altra applicazione o comunque recapitargli un messaggio direttamente. Il sistema viene usato quindi come intermediario. Esso si occupa di far partire il processo dell'applicazione corrispondente, se essa non è già in esecuzione, dopodiché istanzia un oggetto della classe relativa al componente e lo avvia.

Una diretta conseguenza di questa caratteristica è che un'applicazione non ha un unico entry point, come una funzione *main()*, ma ne ha tanti quanti sono i componenti che dichiara come pubblici, ovvero che hanno degli *Intent filters*[3]. Se

[3] Si veda Sezione B.2.5

l'applicazione viene fatta però partire dall'utente tramite l'icona dell'App, l'entry point è rappresentato dalla Main Activity dell'applicazione, contrassegnata così tramite il file Manifest attraverso particolari *Intent Filters*.

Come detto, di default ogni applicazione è ospitata da un processo a sè stante con una propria istanza della Dalvik Machine. In questo processo viene anche eseguito il codice di tutti i componenti che la compongono. Ogni pezzo di codice è eseguito nello stesso thread, a meno che non sia avviato esplicitamente un altro thread utilizzando i meccanismi nativi di Java per la gestione dei thread, oppure usando classi per gestire in modo automatico le interazioni tra main thread e thread secondari, come ad esempio *AsyncTask* o *IntentService*.

Il sistema può decidere di uccidere un processo, distruggendo tutti i suoi componenti, rispettando un ordine di priorità:

- I primi ad essere eliminati sono i processi vuoti, che sono considerati tali se sono attivi senza che sia attivo alcun componente. Ciò significa che per eseguire codice in background senza correre il rischio che l'operazione venga interrotta, occorre che esso sia legato in qualche modo a un Broadcast Receiver o a un Service.

- Dopo si passa ad eliminare i processi che contengono le Activity in background[4].

- A seguire si uccide il processo dell'Activity visibile ma non completamente in foreground. Ciò accade di rado e in situazioni di gravi scarsità di memoria.

- L'ultimo processo che viene distrutto è quello dell'Activity in foreground, considerata la più importante. Essa viene distrutta solo nel caso in cui occupi più memoria di quella effettivamente disponibile nel dispositivo. A quel punto l'uccisione del processo è necessaria per mantenere il sistema ragionevolmente veloce poiché il sistema sta rallentando a causa del paging su disco.

[4] Si veda Sezione B.2.1

B.2.1 Activity

L'Activity rappresenta una schermata mostrata all'utente. Ogni Activity deve essere sottoclasse della classe `Activity`, fornita dalla piattaforma.

La schermata è contenuta in una finestra che è assegnata all'Activity e che essa si occupa di riempire utilizzando il metodo `setContentView(View)`. Grazie a questo metodo è possibile sia creare un'interfaccia composta da una gerarchia di *View*, oggetti di classe (o sottoclassi di) `View`, racchiusi in un *ViewGroup* che normalmente è un layout, ovvero una direttiva sull'ordine e la disposizione delle View in esso contenute. L'interfaccia può essere specificata sia tramite un file XML che contiene la descrizione della schermata, sia creando ViewGroup e View da codice.

Ogni activity viene fatta partire dal sistema grazie a un Intent. Per gestire il ciclo di vita dell'Activity (si veda Figura B.3) è necessario implementare alcuni metodi che vengono chiamati dalla piattaforma durante l'evolversi dei cambiamenti di stato, quali `onCreate()`, `onPause()`, ecc.

Un'activity può essere in uno dei seguenti stati:

Resumed/Running è lo stato in cui l'Activity è in foreground e pronta a ricevere eventi da parte dell'utente.

Paused in questo caso, l'Activity è ancora in foreground ma è parzialmente coperta da un'altra Activity che non copre tutto lo schermo, lasciandola in parte visibile.

Stopped l'Activity è stata rimossa dal foreground ed è in background. Di fatto è ancora viva ma non è collegata al gestore delle finestre.

Quando un'Activity è *Paused* o *Stopped* può essere distrutta da parte del sistema per ragioni di risparmio di memoria. In ogni caso è possibile salvare la condizione dell'Activity al momento della terminazione utilizzando il metodo `onSaveInstanceState()`, che comunque è già implementato nella classe originale per il salvataggio di base (ad esempio, quando viene ruotato lo schermo

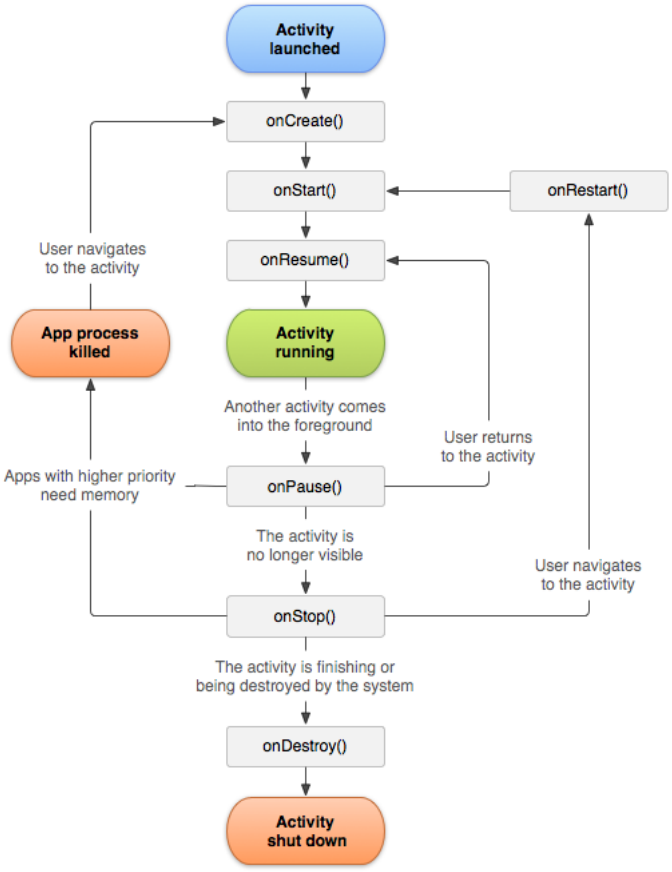


Figura B.3: Ciclo di vita di una Activity

l'Activity viene distrutta e ricreata e in automatico sono salvati, ad esempio, i contenuti delle *Text Field*).

In Figura B.4 è mostrata la gestione delle Activity. Quando un'applicazione viene fatta partire, il sistema le riserva uno stack detto *back stack* che permette di tenere traccia delle Activity e di navigare all'indietro tramite il tasto *back*. Il comportamento di default prevede che un Intent diretto a una Activity ne provochi l'instanziazione e la porti in foreground; a questo punto essa viene inserita nello stack della propria applicazione con una *push*. Se l'Activity ne fa partire un'altra, quest'ultima viene collocata in cima allo stack, facendo passare la precedente in background ma mantenendone lo stato. Alla pressione del tasto back, l'Activity

in foreground viene distrutta e ne viene fatta la *pop* dallo stack. A questo punto l'Activity non è più recuperabile e viene portata in foreground l'Activity che si trova in cima allo stack.

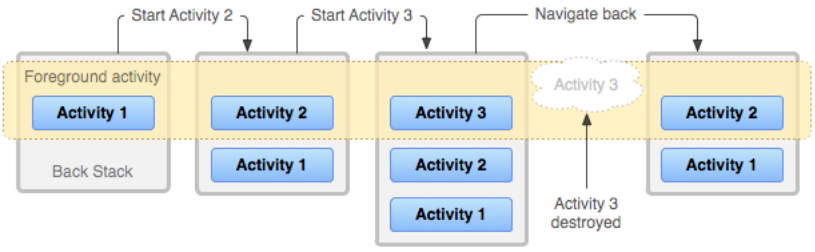


Figura B.4: Activity e Task

B.2.2 Service

Un servizio è un componente che si occupa di eseguire operazioni lunghe in background. Di default è eseguito sul main thread, per cui a meno di non usare interazioni particolari, descritte a breve, occorre gestire un eventuale multithreading.

I servizi si dividono in *Started Service* (o *Unbounded Service* o semplicemente *Service*) e *Bounded Service*. In Figura B.5 sono mostrate i cicli di vita di questi due tipi di servizi.

Un Service normale non è necessariamente o Bound o Unbound, ma può essere entrambi contemporaneamente.

Started Service

Un normale Service viene fatto partire chiamando `startService()`, utilizzando un Intent. Con questo tipo di approccio, non è possibile interagire con il servizio dopo averlo avviato, perciò non si possono ottenere dati in risposta, ma solo effettuare delle operazioni.

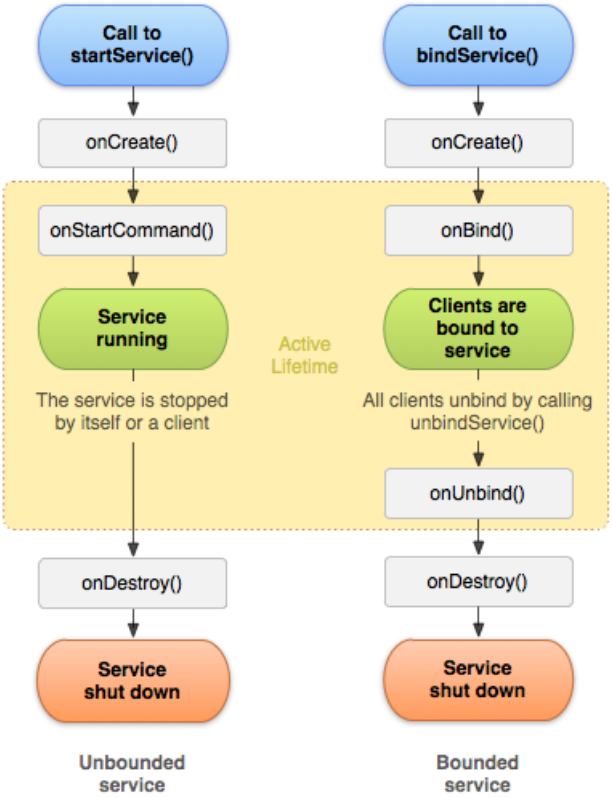


Figura B.5: Cicli di vita dei Service

Un servizio di questo tipo, dopo essere stato fatto partire, può rimanere in esecuzione indefinitamente, perciò per non sprecare risorse occorre che sia il servizio stesso a stopparsi quando ha portato a termine il suo compito.

Per creare un Service di questo tipo occorre effettuare il subclassing della classe `Service` e implementare il metodo `onBind()` restituendo un oggetto nullo.

Nel caso in cui l'operazione che il servizio deve eseguire sia onerosa in termini di CPU e/o tempo, occorre che essa sia spostata su un thread diverso dal main thread. Per fare questo ci sono due modi:

- Utilizzare una sottoclasse di *IntentService*[5]. Nel creare un IntentService, la piattaforma mette a sua disposizione un worker thread su cui eseguire il

[5]Disponibile dalla versione 1.5 in poi

proprio compito. In automatico il sistema mette in coda nuove richieste per quel particolare servizio ed esse vengono eseguite una per volta.

- Nel caso in cui questo approccio fosse troppo limitante, come ad esempio nel caso in cui si vogliano gestire correttamente chiamate multiple con un normale approccio multithreading, si può fare una sottoclasse di un normale Service che gestisca la creazione di thread.

Bounded Service

Un Bounded Service è un servizio che offre un'interfaccia di tipo client/server e come tale permette lo scambio di dati. Esso si ottiene come sottoclasse di un normale Service che però implementa il metodo `onBind()` restituendo un oggetto di tipo `Binder`. Questo oggetto definisce l'interfaccia di comunicazione tra il Service e il resto dell'applicazione.

Il Binder non è l'unico modo per definire l'interfaccia del servizio ma è corretto per l'uso di Service all'interno della stessa applicazione. Se si vuole rendere disponibile il Service alle altre applicazioni (settando correttamente il Manifest) si possono usare altri metodi, quali l'uso di un *Messenger*, che permette di inviare messaggi al Service, oppure l'AIDL (Android Interface Definition Language) che definisce un linguaggio apposito per l'utilizzo di un Service tramite un file pubblico. Entrambi questi casi sono modi per inviare dati serializzandoli, utilizzando una sorta di RPC (Remote Procedure Call).

B.2.3 Content Provider

Un Content Provider si occupa di gestire un insieme di dati condiviso. Questi dati possono trovarsi su un database, su file system, su internet o in un qualsiasi altro tipo di supporto di memoria.

I database che possono essere utilizzati sono di tipo SQLite.

I Content Provider sono necessari per condividere dati tra applicazioni, ma sono utili anche per dati che devono rimanere privati all'interno di un'App. È

possibile sia creare un proprio Content Provider per i propri dati sia utilizzarne uno già esistente, dopo aver ottenuto i dovuti permessi.

Per creare un Content Provider occorre fare una sottoclasse della classe `ContentProvider`. Occorre perciò implementare una serie di metodi che fanno parte dell'interfaccia:

- query()
- insert()
- update()
- delete()
- getType()
- onCreate()

Implementati questi metodi e definiti i vari identificativi per l'accesso ai dati, non è più necessario lavorare direttamente con il Content Provider. Infatti, i contenuti dei Content Provider possono essere ottenuti utilizzando un Content Resolver e specificando l'identificativo dei dati che si vogliono ottenere. I dati sono identificati da un URI che identifica ad esempio il database, la tabella e l'id del dato.

B.2.4 Broadcast Receiver

I componenti di tipo Broadcast Receiver sono sottoclassi di `BroadcastReceiver` e servono per ricevere e reagire a determinati eventi, chiamati appunto *Broadcast*, che possono essere originati dal sistema (ad esempio, per avvisare l'utente che la batteria è quasi scarica) o dalle applicazioni. I Broadcast Receiver sono abilitati a ricevere Intent inviati con appositi metodi e lo scope dell'Intent può essere limitato all'applicazione stessa, oppure tutto il sistema può riceverlo.

Un Broadcast Receiver si limita a implementare il metodo `onReceive()` che viene chiamato dal sistema al momento della ricezione di un broadcast. Terminato il metodo, il componente non viene più considerato attivo.

B.2.5 Intent

Gli Intent permettono ai componenti di un'applicazione di comunicare tra di loro. È una forma di comunicazione indiretta poiché è mediata dalla piattaforma sottostante. Ogni componente (esclusi i Content Providers) può far partire un altro componente tra Activity, Service e Broadcast Receivers inviando l'apposito Intent.

L'Intent può essere esplicito o implicito. Nel caso in cui sia esplicito, il componente specifica direttamente il nome della classe di cui vuole che il componente sia istanza. Questo metodo funziona per Intent all'interno della stessa applicazione. Nel caso degli Intent impliciti, possono essere specificate alcune informazioni quali l'azione che deve essere eseguita, la categoria dell'azione e i dati da utilizzare. In questo caso, il sistema verifica quali applicazioni presentano nel proprio Manifest componenti in grado di rispondere a questo tipo di Intent e, nel caso ce ne sia più di uno, il componente viene fatto scegliere all'utente.

Per questo tipo di selezione, per ogni componente nel Manifest devono essere specificati degli *Intent Filters*, che servono a indicare a quali tipi di Intent i componenti sono in grado di rispondere. Un filtro particolare è quello utilizzato per specificare che una certa Activity è la Main Activity di un'applicazione; in quel caso occorre dichiarare che l'Activity risponde ad una azione di tipo `MAIN` della categoria `LAUNCHER`. Se non si vuole che i componenti siano accessibili fuori dallo scope dell'applicazione, basta non specificare alcun Intent Filter.

Nel caso degli Intent impliciti, potrebbe essere che l'applicazione che contiene il componente target non sia attiva e in quel caso viene fatta partire, perciò l'Intent non provoca solo la consegna di un messaggio ma anche la creazione di un nuovo processo e l'istanziazione del componente interessato.

Un Intent può contenere anche degli *extra*. Gli extra sono dati inclusi nell'Intent che vengono serializzati e inviati verso il componente target.

B.2.6 View

Le interfacce grafiche sono formate da una gerarchia (Figura B.6) di *View* e *View Group*. I View Group sono istanze della classe `ViewGroup` e sono oggetti atti a contenere degli oggetti View, per cui costituiscono layout e contenitori. Le View, dette anche *widget*, sono istanze di classe `View` e possono essere di vario tipo, dai bottoni ai selettori di date. È ovviamente possibile fare una sottoclasse sia di una View che di un widget già presente per creare componenti personalizzati.

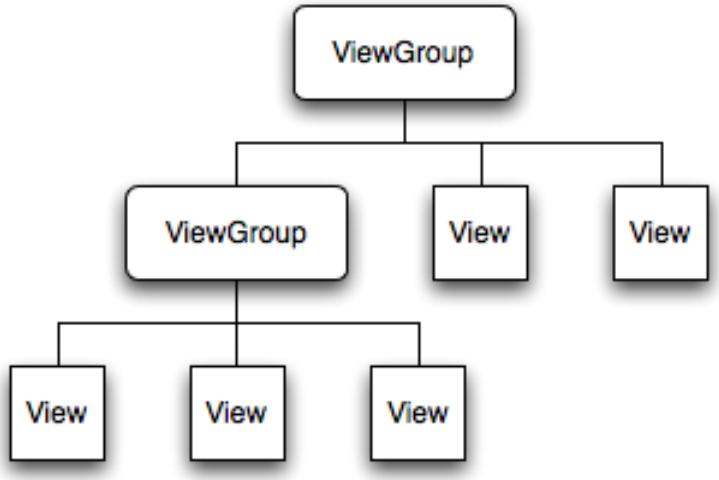


Figura B.6: Gerarchia di View

La classe View, oltre a racchiudere le caratteristiche di un componente che deve essere disegnato su schermo e in grado di interagire con l'utente, include anche una serie di interfacce per la gestione degli eventi. Per ricevere un evento in un'Activity, occorre associarle un ascoltatore per l'evento specifico; l'ascoltatore deve estendere l'apposita interfaccia e implementare il metodo associato all'evento.

Esistono particolari tipi di ViewGroup, detti *Adapter View*, che si occupano di effettuare il collegamento tra un'insieme di View e dei dati, che possono essere ad esempio array (`ArrayAdapter`) o provenire da database tramite curso-

ri (`CursorAdapter`). Gli Adapter recuperano i dati dalla sorgente indicata e si occupano di costruire un numero congruo di View di conseguenza.

Appendice C

Un esempio applicativo

In questa appendice si metteranno a confronto le due piattaforme grazie a un caso pratico molto semplice.

L'esempio prevede la creazione di due schermate e il passaggio da una all'altra tramite un bottone.

C.1 Sviluppo su Android

Su Android possiamo creare questa applicazione da Eclipse, con l'apposito plugin. Saranno necessarie due Activity, ovvero *FirstActivity* e *SecondActivity*, e due schermate semplici create con l'editor visuale, generando così due file *first.xml* e *second.xml*. Infine dovrà essere settato il file Manifest per notificare che l'applicazione contiene due Activity.

Il codice della prima Activity sarà:

```
public class FirstActivity extends Activity {
 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.first);
  Button button = (Button) findViewById(R.id.button1);
  button.setOnClickListener(new OnClickListener() {
```

```
    @Override
    public void onClick(View v) {
     showSecondView();
    }
   });
  }

  public void showSecondView() {
   Intent intent = new Intent(getApplicationContext(),
          SecondActivity.class);
   startActivity(intent);
  }
}
```

Come si può vedere, per passare alla schermata successiva è necessario far partire l'activity SecondActivity, che viene avviata con un Intent esplicito. Per fare ciò si associa un listener creato al volo al bottone.

Il codice della seconda Activity sarà semplicemente:

```
public class SecondActivity extends Activity {
 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.second);
 }
}
```

La seconda Activity non fa nulla, a parte impostare la View da mostrare.

Infine il Manifest sarà:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res
      /android"
 package="esempi.esempio1" android:versionCode="1" android:
       versionName="1.0">
 <uses-sdk android:minSdkVersion="2" />
```

```
<application android:icon="@drawable/icon" android:label="
      @string/app_name">
 <activity android:name=".FirstActivity" android:label="
       @string/app_name">
  <intent-filter>
   <action android:name="android.intent.action.MAIN" />
   <category android:name="android.intent.category.
         LAUNCHER" />
  </intent-filter>
 </activity>

 <activity android:name=".SecondActivity">
 </activity>

</application>
</manifest>
```

L'intent filter della prima Activity indica alla piattaforma che essa è l'Activity da lanciare al momento del caricamento dell'applicazione. La seconda Activity non ha filtri, quindi non è raggiungibile se non con un Intent esplicito dall'interno dell'applicazione.

C.2 Sviluppo su iOS

Si parte creando un progetto vuoto in XCode, che permette di avere già l'App-Delegate e la Window. A questo punto occorrerà aggiungere due View Controller e due View. Ci sono diversi modi per passare da un View Controller ad un altro, ma il più simile alla filosofia Android è quello che utilizza il Navigation Controller. Questo è anche il modo normalmente più usato nelle App per iPhone, dove il passaggio da una View all'altra avviene attraverso un metodo di navigazione gerarchico delle funzionalità. Utilizzando un Navigation Controller abbiamo sempre in alto una barra che permette di controllare lo spostamento e che possiede un ti-

tolo e un tasto Back per tornare indietro. Il Navigation Controller fornisce uno stack di View Controller navigabili e in questo senso è molto simile al Back Task di Android.

Si definiscono quindi i due View Controller con le rispettive View: *FirstViewController* e *SecondViewController*. Dopo aver impostato dal Delegate il Navigation View Controller come Root Controller della Window principale, occorrerà settare FirstViewController come View Controller iniziale da mostrare all'interno del Navigation View Controller. Terminata questa procedura, nella View del FirstViewController si creerà un semplice bottone che inneschi il passaggio al secondo View Controller. Da Interface Builder occorrerà collegare all'evento Touch Up Inside il metodo
`showSecondView:(id)sender` contrassegnato come `(IBAction)` per essere correttamente riconosciuto da XCode come collegabile ad un evento con il meccanismo Target-Action. In questo modo, il Target è il View Controller e la Action riporterà il selettore `showSecondView:(id)sender`; l'evento corrisponderà a quello del tocco del bottone.

Il codice del metodo sarà il seguente:

```
-(IBAction) showSecondView:(id) sender{
    SecondViewController* controller = [[
          SecondViewController alloc]
          initWithNibName:@"SecondViewController" bundle:
                nil];
    [self.navigationController
          pushViewController:controller animated:YES];
    [controller release];
}
```

Il View Controller deve essere rilasciato al termine del metodo perché all'oggetto in questione non occorre più il riferimento ad esso e quindi il retain count deve essere decrementato. `self.navigationController` è una proprietà di tutti gli `UIViewController`, pertanto non occorre mantenere il riferimento al Navigation Controller creato presso l'AppDelegate.

Elenco delle figure

Bibliografia

[1] http://developer.android.com/

[2] Cocoa Fundamentals Guide, Apple Inc., disponibile all'indirizzo https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html

[3] Event Handling Guide for iOS, Apple Inc., disponibile all'indirizzo https://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/Introduction/Introduction.html

[4] iOS App Programming Guide, Apple Inc., disponibile all'indirizzo https://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html

[5] The Objective-C Programming Language, Apple Inc., disponibile all'indirizzo https://developer.apple.com/library/ios//#/library/mac/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html

[6] Software system specifications in Contact, Antonio Natali