Università degli studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA

PORTING DI APPLICAZIONI MOBILE DI TIPO "MISSION CRITICAL" DA JAVA ME A GOOGLE ANDROID

RELATORE: Prof. Sergio Congiu

CORRELATORI: Ing Alessandro Costacurta, Ing. Davide Baracco

LAUREANDO: Federico Bortolomiol

A.A. 2012-2013

Sommario

Questa attività di tesi è stata svolta in ambito di stage presso MBM Italia S.r.l, un'azienda di Padova che si occupa di progettare e realizzare sistemi informativi business; in modo particolare ERP. Lo stage ha avuto una durata di 6 mesi ed è stato svolto interamente in azienda. Mi sono occupato sostanzialmente di porre le basi per il porting di applicazioni business di tipo "mission critical" sviluppate in Java ME su Google Android.

Indice

Capitolo 1

Introduzione

Questo capitolo ho lo scopo di introdurre il tipo di tesi svolta, gli obiettivi, le motivazioni e il metodo di lavoro. Sarà fatta anche una panoramica sull'architettura del sistema client-server a cui fa riferimento il mio progetto.

1.1 Obiettivi e motivazioni

Come accennato nel sommario la tesi è stata svolta presso MBM Italia S.r.l, un'azienda che progetta e realizza sistemi informativi business. Essa si concentra in modo particolare nel settore industriale, fornendo sistemi informativi integrati per il controllo dei flussi della logistica aziendale, sia interna che esterna (supply chain management).

Tra gli altri l'azienda sviluppa sistemi che prevedono l'utilizzo di dispositivi mobile (PDA) per la rilevazione di dati sul campo. Un esempio in questo senso è *WCube*, un sistema per la gestione del magazzino (WMS - Warehouse Management System), in cui vi sono due tipologie di utenti: i gestori, che operano da una postazione fissa e si occupano della parte gestionale; e gli operatori di magazzino, dotati di PDA per interagire con il sistema centrale.

Tecnicamente *WCube* è un sistema client-server sviluppato con tecnologia Java, in cui i client fissi (gestori) accedono al server utilizzando una web application, mentre i client mobile (operatori) vi accedono utilizzando un'applicazione realizzata nella piattaforma Java ME. L'applicazione mobile è fruita da PDA dotati di SO Windows Mobile. L'architettura di *WCube* è adottata anche in altri sistemi sviluppati da MBM che prevedono applica-

zioni mobile, io mi riferirò sempre a questo progetto poiché è quello che mi è stato sottoposto come caso di studio.

Se il mercato smartphone è ormai per due terzi equipaggiato dal SO Android lo stesso non si può dire per i PDA, dove va per la maggiore ancora Windows Mobile. Solamente negli ultimi mesi nel mercato si cominciano a trovare i primi dispositivi di questo tipo con Android. La tendenza dei costruttori per il futuro è comunque quella di orientarsi verso sistemi open, come Android.

MBM ha quindi deciso di intraprendere un processo di ristrutturazione che prevede l'abbandono della piattaforma Java ME a favore di Android. La scelta è ricaduta su Android per una serie di ragioni che saranno chiarite nei prossimi capitoli e che quì accenno solamente: il fatto che Android è open, l'utilizzo di JAVA come linguaggio di sviluppo, la diffusione del SO. In azienda nasce quindi la necessità di effettuare il porting delle attuali applicazioni realizzate sulla piattaforma Java ME su Android.

L'obiettivo del mio stage era acquisire tutte le conoscenze preliminari sulla nuova piattaforma per verificare la fattibilità e la mole di lavoro necessaria per poter fare il porting delle applicazioni mobile esistenti. L'obiettivo finale era quello di realizzare un'applicazione prototipo, sulla base di *WCube*, per testare l'iterazione con il resto del sistema e avere un'applicazione campione da mostrare qualora i clienti lo richiedessero. Questa applicazione prototipo rappresenta quindi parte del porting dell'applicazione *WCube* esistente.

Un aspetto importante è il fatto che *WCube* è un'applicazione di tipo "mission critical", ovvero gli aspetti di sicurezza, "data loss", e "disaster recovery" assumono una certa rilevanza. In MBM è stato fatto un lungo lavoro di ricerca che ha portato a realizzare applicazioni che soddisfano questi requisiti. Per questo motivo, e per garantire la retrocompatibilità con il resto del sistema, nel effettuare il porting la logica architetturale e di comunicazione con il server doveva necessariamente rimanere inalterata.

In figura 1.1 è data una rappresentazione schematica del sistema *WCube*.

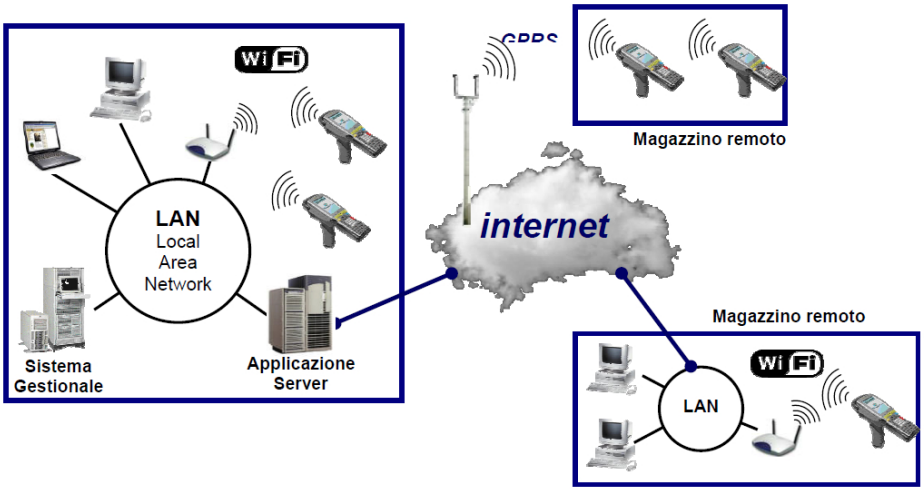

Figura 1.1: Sistema *WCube*

1.2 Architettura di WCube

L'architettura del sistema *WCube* è di tipo SOA (service-oriented architecture) e quindi l'iterazione tra client e server è resa possibile grazie a dei Web Service. In questo modo è possibile ottenere un forte disaccoppiamento tra client e server, nonostante non ce ne fosse la reale necessità, poiché entrambi sono implementati utilizzando la medesima tecnologia Java. A suo tempo però l'azienda ha preferito adottare questa soluzione, rispetto ad altre nativamente distribuite per l'ambiente Java, come poteva essere Java RMI. Il web service utilizza gli standard WSDL (per descrivere l'interfaccia pubblica del web service) e SOAP (per lo scambio di messaggi). In particolare, in MBM, il formato dei messaggi SOAP di comunicazione è stato standardizzato in una tecnologia interna chiamata ECO.

Per garantire la persistenza dei dati rilevati sul campo le applicazioni client mobile utilizzano un database locale. In questo modo anche a fronte di anomalie nell'applicazione (client o server), o nella comunicazione, i dati fino a quel momento rilevati non saranno persi. Il vantaggio di utilizzare questo approccio risiede anche nel fatto che l'applicazione può lavorare in modalità "off-line": è infatti sufficiente scaricare le consegne dal server una prima volta, eseguendo un'operazione di allineamento, per garantire l'operabilità minima dell'applicazione per lungo tempo, anche in assenza in copertura WiFi o GPRS.

Le consegne, nella logica operativa di *WCube* sono racchiuse nel concetto di "missione". Una missione di magazzino è un insieme di movimentazioni da effettuare, possibilmente in sequenza, a fronte di una richiesta del sistema informativo aziendale o di esigenze specifiche del magazzino. Riguarda sempre un unico magazzino e viene di norma assegnata ad un operatore. La missione può essere di prelievo, di versamento o di trasferimento tra locazioni. L'imputazione dei dati (codici, locazioni, ecc.) può avvenire tramite lettura di codici a barre o digitazione da tastiera.

Lo stato delle missioni è rappresentato da una LUW, cioè da una Logical Unit of Work. Questi oggetti, salvati anch'essi in modo persistente, inglobano tutte le informazioni necessarie al server per processare, quando richiesto, le varie missioni scaricate sul dispositivo. La sincronizzazione tra client e server sullo stato delle missioni viene garantita da un servizio in background, che, all'occorrenza, invia al server tutte le LUW presenti nel database.

Le applicazioni client di MBM realizzate nella piattaforma Java ME, tra cui *WCube*, sono realizzate a partire da una libreria di supporto che funge da framework, denominata *MbmPalm*. Tale libreria ha sostanzialmente lo scopo di mettere a disposizione componenti e funzionalità comuni ad ogni applicazione mobile nel contesto di MBM. In questo modo si rendono più modulari le applicazioni con tutti i vantaggi di gestione e riutilizzo del codice che ne derivano.

1.3 Metodo di lavoro

In questa sezione presento le tappe che mi hanno portato a realizzare l'applicazione prototipo, denominata nel resto della relazione *WCubeApp*.

Ad inizio stage mi è stata presentata una lista di funzionalità implementative sulle quali fare ricerca in ambiente Android. Queste funzionalità costituivano sostanzialmente i moduli più rilevanti del framework *MbmPalm*. Non avendo alcun background sulla programmazione Android, la primissima fase, quantificata in circa un mese di lavoro, ha previsto uno studio delle API fornite dal SDK e dei pattern di programmazione. Terminato questo primo lavoro di know-how mi è stata presentata l'applicazione *WCube* esistente nella piattaforma Java ME, denominata nel resto della relazione come *WCubePalm*, sulla quale ho dedicato ulteriore tempo per comprenderne logica applicativa e iterazione con il framework.

Tutto questo lavoro preliminare mi ha permesso di entrare in confidenza con Android e di avere una prima stima sulla mole di lavoro necessaria, quantomeno per reimplementare i moduli principali di *MbmPlam* su Andorid.

Come accennato le applicazioni Android sono scritte in Java, che è lo stesso linguaggio utilizzato dalla piattaforma Java ME. Effettuare il porting non implica quindi una riprogrammazione radicale ma piuttosto una sorta di adattamento del codice alla nuova piattaforma. Questo è uno dei principali motivi per cui MBM ha deciso di puntare Android. Molti dei moduli di *MbmPalm* sono quindi stati semplicemente riportati nel nuovo progetto, con l'apporto di qualche piccola modifica. Altri invece, come ad esempio tutto ciò che comprendeva user interface o data storage, sono stati completamente rivisti e riprogrammati.

Terminata questa prima fase ho iniziato ad adattare il framework alla nuova piattaforma e parallelamente ho iniziato lo sviluppo di *WCubeApp*. *WCubeApp* nella fase iniziale fungeva da applicazione di test per effettuare test reali che verificassero la fattibilità del porting e la bontà delle soluzioni adottate in fase di riprogettazione del framework.

Inizialmente, framework e applicazione, per questione di comodità erano racchiusi in un unico progetto; ma una volta implementate e testate tutte le funzionalità richieste si è proceduto a separare il progetto in due entità: il framework, rinominato *MbmDroid* nel nuovo contesto, e l'applicazione prototipo. Il progetto *MbmDroid* fungerà quindi da libreria per *WCubeApp*, esattamente com'è *MbmPalm* per tutte le applicazioni attualmente realizzate in MBM su Java ME.

Per verificare la compatibilità i test sono stati effettuati su tre dispositivi aventi versioni di Android differenti: un tablet, uno smartphone e il PDA sul quale con molta probabilità sarà distribuita l'applicazione.

Capitolo 2

Android

Come introdotto precedentemente il lavoro di tesi è stato incentrato completamente sulla ricerca e sviluppo in ambiente Google Android. Non avendo alcun tipo di background la prima parte del mio lavoro è stata acquisire una conoscenza preliminare di tale ambiente e degli strumenti che mi permettessero poi di iniziare lo sviluppo. In questo capitolo viene quindi introdotto il SO Android e la sua architettura, saranno poi presentati gli strumenti di sviluppo utilizzati nel corso della mia attività, ed infine verranno analizzati i componenti base su cui costruire ogni applicazione e l'iterazione tra essi. Prima però vediamo quali sono le possibilità per sviluppare un applicativo mobile.

2.1 Mobile Development

Un'applicazione mobile può essere sostanzialmente di due tipi: nativa oppure una web-app.

Le applicazione native sono specifiche per ogni piattaforma e lo sviluppo avviene utilizzando il relativo SDK, ciò implica che i linguaggi di sviluppo possono essere anche differenti, come nel caso di Android (Java) , iOS (Object-C) e Windows Mobile (ambiente .NET). Le applicazioni native sono installate direttamente nel dispositivo.

Le web-app sono invece applicazioni internet fruibili via un comune browser. Si differenziano dalle tradizionali web application per il fatto che la loro user interface è progettata per essere visibile su un display di piccole

dimensioni. Le web-app sono scritte utilizzando HTML, JavaScript e i CSS, più eventuali framework.

La prima importante differenza tra i due approcci riguarda la compatibilità. Adottare l'approccio nativo per realizzare un'applicazione su vasta scala, fruibile da dispositivi con diversi OS, implica lo sviluppo di un'applicazione per ogni piattaforma, mentre ciò non è vero nel caso si decida di realizzare una web-app, in quanto è sufficiente un comune browser internet. La semplicità nell'ottenere una applicazione cross-platform è quindi a pannaggio delle web-app. Uno svantaggio sta invece nel fatto che le web-app sono utilizzabili solo se è presente una connessione internet, non sono quindi in grado di lavorare "offline".[1]

L'approccio nativo riscontra grandi vantaggi dal punto di vista delle performance: accedendo direttamente all'application framework del sistema (utilizzando le API) le applicazioni native presentano una UI più fluida e reattiva rispetto alle web-app. Inoltre sono in grado di accedere direttamente a molte funzionalità del dispositivo, come sensori, fotocamera, ecc.

In futuro, soprattutto per quel che riguarda la UI, la differenza tra app native e web app dovrebbe comunque assottigliarsi, questo grazie alla recente introduzione di HTML5. Quest'ultima versione del linguaggio di markup introduce infatti nuove funzionalità create appositamente per venire incontro alle esigenze delle applicazioni mobile.

In MBM l'idea di effettuare il porting su web-app non è mai stata presa in considerazione, poiché ciò comporterebbe una totale riprogettazione dell'applicazione e dell'intero sistema.

2.2 Introduzione ad Android

Android è, ad oggi, di gran lunga il sistema operativo più utilizzato nei dispostivi mobile: secondo una ricerca eseguita dalla società di rating IDC (sett. 2012) la piattaforma nata dalla collaborazione tra Google e la Open Handset Alliance detiene infatti il 68% del mercato, un volume quattro volte superiore rispetto a quello del maggiore concorrente, iOS di Apple. Essendo la quota in progressivo aumento si può quindi affermare che Android è sulla buona strada per raggiungere la posizione di monopolio, un fenomeno che ricorda quello di Windows negli anni novanta con soluzioni desktop. Il para-

[1] A meno che non si adotti un particolare sistema di local caching.

gone però ha valenza solo in termini statistici, in quanto Windows e Android hanno una filosofia completamente differente: il primo è tutt'ora un sistema chiuso e proprietario, mentre Android è completamente opensource.

Android nasce sostanzialmente della necessità di una standardizzazione, fino ad allora assente, nello sviluppo di applicazioni per dispositivi mobile. Fino all'avvento ed alla consacrazione di Android ogni produttore realizzava per i propri dispositivi un suo SO ed un suo kit di sviluppo; sviluppare un'applicazione su larga scala e fruibile su dispositivi prodotti da vendor differenti richiedeva quindi la conoscenza di piattaforme, tecnologie e linguaggi diversi, nonché l'utilizzo di strumenti di sviluppo proprietari. Per eliminare queste barriere e far esplodere definitivamente il boom degli smartphone, Google e la Open Handset Alliance, nel 2007, si sono messe attorno ad un tavolo e dalla loro collaborazione è nato Android.

Android non è solamente un sistema operativo, ma va molto oltre, una possibile definizione potrebbe essere: uno stack di componenti software opensource che includono sistema operativo, middleware, e un set di applicazioni native, che, insieme ad un set di librerie e API, permettono ai costruttori di realizzare dispositivi compatibili, e agli sviluppatori di creare applicazioni che possono accedere alle funzionalità dei device.

Android, come detto, è un sistema open, dove il termine assume diverse sfumature:

- Android è open in quanto utilizza, come vedremo meglio successivamente, tecnologie open, prima fra tutte il kernel di Linux.
- Android è open in quanto le librerie e le API che sono state utilizzate per la realizzazione di applicazioni native sono esattamente le stesse che sono utilizzate per sviluppare le applicazioni realizzate da terzi.
- Android è open in quanto il suo codice è open source (Open Source Apache License 2.0). Questa licenza libera permette agli sviluppatori di visionare e migliorare il codice e ai diversi vendor di costruire su Android le proprie estensioni, anche proprietarie, senza legami che ne potrebbero limitare l'utilizzo.
- Android è open poiché fornisce un SDK (Software Development Kit) gratuito che contiene le librerie e i tool di sviluppo necessari per sviluppare le applicazioni per vari dispositivi.

Sempre per venire incontro alle esigenze degli sviluppatori Android utilizza un linguaggio già ampiamente conosciuto e collaudato come Java. Come vedremo in seguito, nel caso si vogliano sfruttare al massimo le potenzialità del dispositivo, oppure le librerie di Android non permettano di utilizzarne determinate periferiche, c'è la possibilità di utilizzare anche codice nativo (C/C++). Ciò è reso possibile grazie al tool di sviluppo denominato Android Native Development kit (NDK).

Poiché i dispositivi mobile hanno solitamente performance hardware (processore e memoria in primis) inferiori ai pc, le applicazioni non vengono eseguite su una Java Virtual Machine comune, come accade per Java ME, ma da una JVM ottimizzata che prende il nome di Dalvik Virtual Machine (DVM).

Le librerie standard di Java utilizzate da Android sono la quasi totalità, escluse, non a caso, le Abstract Window Toolkit (AWT) e le Swing. La definizione dell'interfaccia grafica è infatti un aspetto fondamentale nell'architettura di Android e può essere fatta in due modi: programmaticamente, oppure in modo dichiarativo utilizzando file XML. Un'altra libreria di una certa rilevanza non inclusa è quella relativa al paradigma RMI. Vediamo ora in maggiore dettaglio l'architettura del sistema Android.

2.3 Architettura

Come accennato nell'introduzione, Android può essere visto come uno stack di componenti software in cui ogni layer ha lo scopo di fornire un'astrazione sempre maggiore del sistema sottostante. Nella figura 2.1 è mostrata una rappresentazione di tale stack e dei componenti che lo compongono.

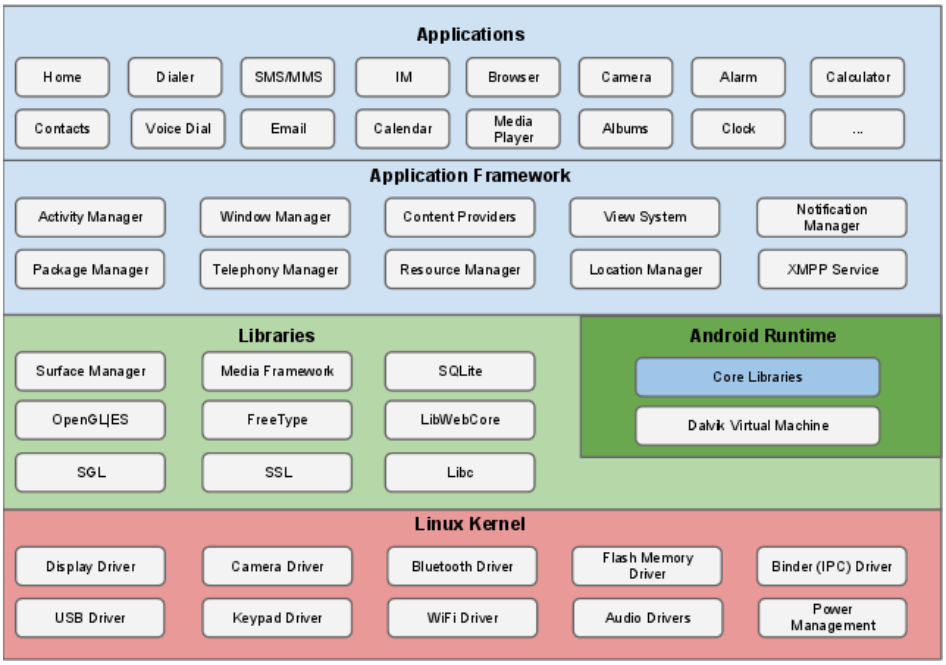

Figura 2.1: Android software stack

Analizziamo lo stack partendo dal layer di livello più basso:

- Linux kernel — I core services (hardware drivers, gestione dei processi e della memoria, sicurezza, rete, gestione dell'alimentazione, ecc.) sono gestiti da un kernel Linux 2.6. Il kernel fornisce quindi un'astrazione dell'hardware sottostante per gli elementi superiori dello stack.

- Librerie native — Una serie di librerie native (di solito scritte in C/C++) che rappresentano il cuore di Android. Queste librerie si occupano di implementare e gestire le varie funzionalità del dispositivo. È importante sottolineare come esse siano ottimizzate per lavorare in ambiente mobile. Tra esse possiamo citare:

  - Surface Manager — Libreria che dà accesso alle funzionalità del display e permette la visualizzazione contemporanea di grafica 2D e 3D dalle diverse applicazioni. Sostanzialmente si occupa di gestire tutto ciò che viene visualizzato dal display.
  - SGL e Open GL ES — Un insieme di API multipiattaforma ottimizzate per i dispositivi embedded che forniscono l'accesso, rispettivamente, a funzionalità 2D e 3D.

  - Media Framework — Librerie per la gestione di contenuti multimediali. Sostanzialmente stiamo parlando dei CODEC.
  - SQLite — Librerie per il supporto nativo a database.
  - WebKit — Si tratta di un framwork che funge da browser engine open source basato sulle tecnologie HTML, CSS, JavaScript e DOM.

- Android Runtime — Ciò che realmente differenzia un device Android da una comune distro Linux installata su un dispositivo mobile è il cosiddetto "Android Runtime". Includendo le core libraries e la Dalvik virtual machine (DVM), questo layer è il motore che permette alle applicazioni di essere eseguite. Le core libraries sono sostanzialmente una revisione delle librerie Java, in cui è stata tolta qualche libreria (es. AWT, Swing e RMI) per aggiungerne di più specifiche per l'ambiente Android. La DVM è stata progettata per fare in modo che ogni applicazione in esecuzione, solitamente rappresentata da un singolo processo, utilizzi una propria istanza della virtual machine. La DVM si pone quindi come middleware tra ogni applicazione e il SO Linux. L'Android run time, assieme alle librerie native, pone le basi per lo strato superiore: l'application framework.

- Application framework — Questo strato è costituito da un insieme di classi e API per l'esecuzione di funzionalità ben precise e di fondamentale importanza in ciascuna applicazione Android. Grazie a queste API è possibile avere accesso e gestire tutte le risorse e le funzionalità del dispositivo, come ad esempio la connessioni WiFi, oppure i dati contenuti in un database SQLite. Questo strato inoltre permette di gestire la user interface (touchscreen, microfono ecc.).

- Application layer — Tutte le applicazioni Android native, o prodotte dalla grande comunità di sviluppatori esterna, sono incluse in questo strato. Tutte le applicazioni sono eseguite all'interno del Android run time, usando le stesse classi e servizi resi disponibili dall'application framework.

2.4 Framework di Sviluppo

Uno dei vantaggi di sviluppare per Android è il fatto che il framework di sviluppo è immediato, potente e gratuito. Qualsiasi applicazione Android viene creata utilizzando questo framework composto da più entità. Oltre al Java Develoment Kit (JDK) sono necessari l'Android Software Develoment Kit (SDK) e, nel mio caso, anche il Native Development Kit (NDK) per la programmazione nativa.

2.4.1 Android SDK

Il Software Development Kit è lo strumento principale ed è composto a sua volta da varie entità:

- Android APIs — Sono il cuore del SDK, poiché forniscono il punto di accesso allo software stack descritto precedentemente.
- Development tools — Semplici tool che permettono di compilare, debuggare e testare in modo ottimale un'applicazione.
- The Android Virtual Device Manager and Emulator — L'SDK fornisce un emulatore che simula un dispositivo Android. poiché il device emulabile è altamente personalizzabile sia come caratteristiche hardware (memoria, display, ecc.), sia software (versione di Android) questo strumento è molto utile in fase di test. Sfortunatamente è assai lento e quindi nel corso della fase di sviluppo ho utilizzato device reali. Inoltre per testare determinate funzionalità in modo ottimale, come ad esempio il touchscreen, è scomodo e poco affidabile.
- Documentazione e codice di esempio.

2.4.1.1 Android APIs

Le API costituiscono il cuore del SDK, in quanto sono lo strumento da cui partire per programmare un'applicazione Android. Il rilascio di nuove API, identificate da un "API Level", coincideva solitamente con l'aggiornamento della versione del SO, ultimamente invece Google tende a rilasciare più API per la stessa versione, più atte a risolvere piccoli bug che introdurre nuove feature. Solitamente ogni nuova versione prevede una versione aggiornata

del kernel Linux. La tabella 2.1 riassume la breve storia delle versioni di Android e delle relative API.

Version	Nome	Api Level
1.0–1.1	Android 1.X	1,2
1.5	Cupcake	3
1.6	Donut	4
2.0, 2.0.1 ,2.1	Eclair	5,6,7
2.2–2.2.3	Froyo	8
2.3.3–2.3.7	GingerBread	9,10
3.0–3.2	HoneyComb	11,12,13
4.0–4.0.4	Ice Cream Sandwich	14,15
4.1–4.2	Jelly Bean	16,17

Tabella 2.1: Storia delle versioni di Android

Per quel che riguarda la distribuzione, come si evince anche dalla figura 2.2, la versione tutt'ora più diffusa rimane GingerBread. Questa versione è anche la più utilizzata dai PDA che montano SO Android attualmente in commercio. Questo è il motivo per cui il target del progetto è la versione GingerBread.

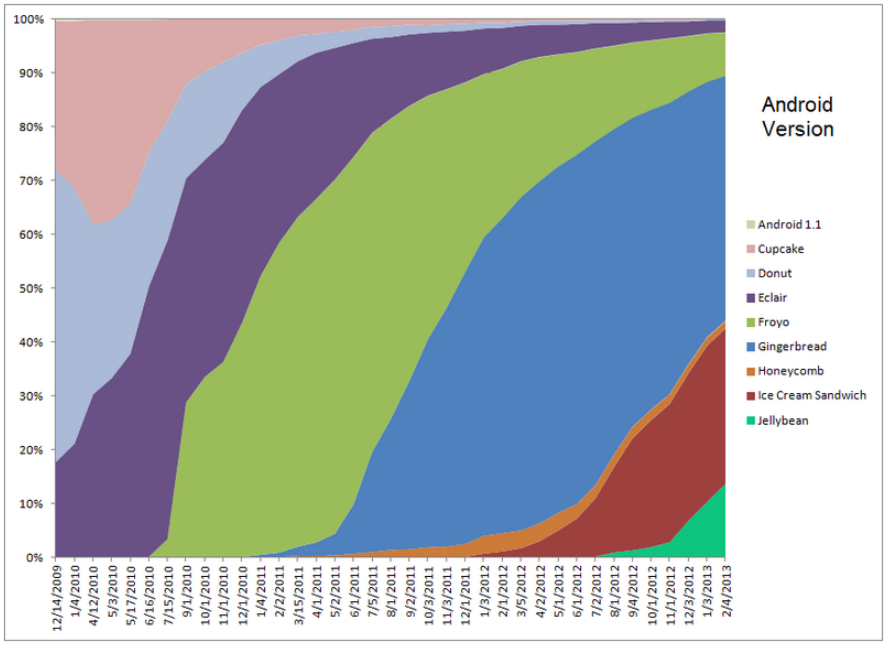

Figura 2.2: Diffusione delle versioni nel tempo

2.4.1.2 Development Tools

Presento brevemente i tool di sviluppo presenti nell'SDK che ho utilizzato nel corso della mia attività.

- Android Debug Bridge (`adb`): come il nome suggerisce questo strumento fa da ponte tra il device (virtuale o reale) e l'ambiente di sviluppo. È principalmente usato per installare l'applicazione sul device e caricare/scaricare file.

- `android`: utilizzato per creare ed eseguire gli Android Virtual Device (AVD), gestire i progetti e l'SDK.

- `logcat`: permette di visualizzare i log provenienti dal dispositivo. Utilissimo in fase di debug poiché permette di visualizzare l'Error Output Stream della DVM.

- `lint`: questo strumento analizza il codice ricercando eventuali bug e ottimizzazioni, al fine di migliorare correttezza, sicurezza, performance, usabilità ed accessibilità del software.

A convogliare e organizzare tutti questi utili strumenti in un unico IDE c'è un plug-in di Eclipse che prende il nome Android Developer Tools (ADT). L'utilizzo di Eclipse con l'ADT è considerato da Android lo strumento di sviluppo ufficiale. Questo plug-in aggiunge anche nuove funzionalità, come ad esempio un editor grafico per la creazione dei layout, che rende ancora più veloce e immediata la programmazione della GUI.

Uno degli strumenti più utili messi a disposizioni dall'ADT è il Dalvik Debug Monitor Server (DDMS): in fase di debug è infatti possibile visualizzare log, file system, processi e altre informazioni in un unica videata. Altri tool incorporati nell'ADT sono l'Android SDK Manager, il Virtual Device Manager e l'Android Asset Packaging Tool (AAPT). Quest'ultimo è in particolare utilizzato nella fase di building dell'applicazione.

Per quel che riguarda i dispositivi utilizzati nei test, essi sono un Tablet Toshiba AT300 (v. OS Ice Cream Sandwitch), uno smartphone Samsung Galaxy S3 Mini (v. OS JellyBean) e un PDA Geofanci MT35 (v. OS GingerBread).

2.4.2 Android NDK

Solitamente per sviluppare un'applicazione Android è sufficiente il SDK in quanto le sue API permettono di avere un controllo sulle principali funzionalità di un comune dispositivo mobile (fotocamera, sensori, GPS, WiFi, ecc). Nel caso però siano richieste altissime prestazioni o si voglia (per scelta o perché non esistono API nel SDK per tale scopo) utilizzare direttamente le librerie native (es. OPEN GL ES) è possibile programmare in modo nativo (C/C++). Utilizzando questa soluzione la DVM viene bypassata e quindi le prestazioni aumentano. Nel mio caso ho dovuto ricorrere alla programmazione nativa per gestire il lettore dei barcode, infatti il produttore non metteva a disposizione alcun SDK per Android, ma solamente API in C++. La programmazione nativa in Android è supportata dall'Native Development Kit (NDK).

2.5 Struttura di un progetto Android

affinché un progetto Android possa essere compilato correttamente è necessario che questo contenga directory e file dai nomi predeterminati. Tale strut-

tura viene creata in modo automatico dall'ADT al momento della creazione del progetto Android.

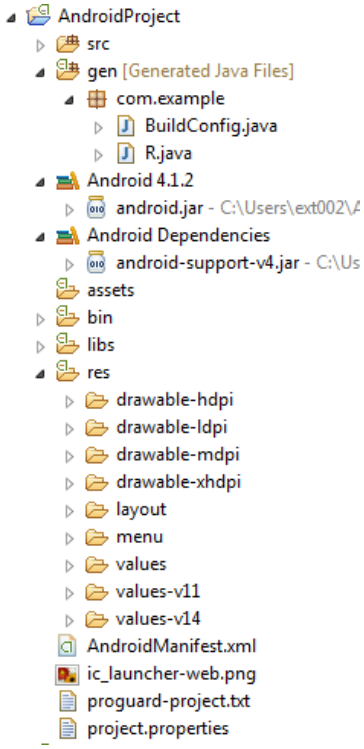

Figura 2.3: Struttura di un progetto Android

Analizziamo file e cartelle contenute nel progetto:

- `src`: directory contenete i sorgenti;
- `gen`: directory contenente i file generati dall'ADT. Il file `R.java`, in particolare, contiene un riferimento ad ogni risorsa (contenuta nella cartella `res`) del progetto;
- `android.jar` e `android-support-v4.jar`: librerie di Android che contengono le API;
- `assets`: directory usata per contenere *raw* file, come ad esempio file di configurazione. La cartella è navigabile dall'applicazione come fosse un file system;

- bin: directory contenente il risultato della compilazione;
- libs: directory in cui inserire librerie aggiuntive;
- res: directory contenente tutte le risorse dell'applicazione come, ad esempio, immagini, layout e stringhe di testo. Le risorse sono definite utilizzando il formato XML. Come organizzare le risorse dentro questa cartella è tema della sezione2.9;
- `AndroidManifest.xml`: il file che descrive l'applicazione e dichiara tutti i suoi componenti. Sarà preso in esame nella sezione 2.8.

L'eventuale codice nativo, più le classi di raccordo previste da JNI vanno inserite nella directory jni.

2.5.1 Building and Running

Nel processo di build il progetto viene compilato e impacchettato in un file, che funge da archivio, con estensione `.apk`. L'archivio contiene tutto ciò che è necessario per l'esecuzione dell'applicazione: file `.dex` (file `.class` convertiti in bytecode specifico per la DVM), una versione binaria del file `AndroidManifest.xml`, risorse compilate nel file `resources.arcs` (es. file XML dei layout), e non compilate (es. immagini). Come detto il processo di build è automatizzato dall'ADT; se invece si vogliono utilizzare altri tool, come ad esempio ANT, è necessario creare un file `build.xml` ad hoc.

Per fare si che l'applicazione possa essere eseguita in un device o nell'emulatore è necessario che questa sia firmata. Il procedimento può essere effettuato in due modalità: "*debug*" e "*release*". La prima modalità viene utilizzata nella fase di sviluppo, in cui c'è bisogno di fare frequenti test. In questa modalità l'AAPT firma l'applicazione sempre con la medesima chiave, fornita nell'SDK, favorendo la velocità dell'operazione. La modalità "*release*", invece, viene utilizzata solamente quando l'applicazione è pronta per essere rilasciata e distribuita. In questo caso la chiave deve essere privata e fornita dallo sviluppatore.

La figura 2.4 fornisce una rappresentazione schematica del processo di building appena illustrato.

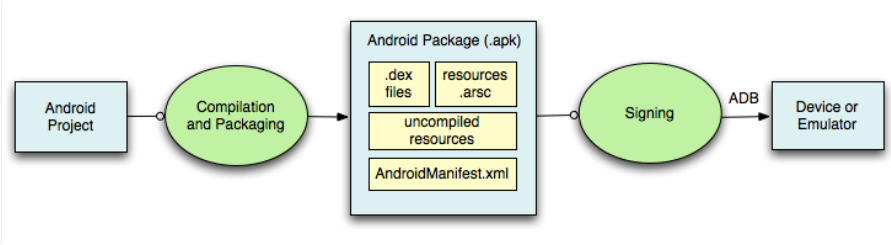

Figura 2.4: Processo di building

2.6 Componenti

I componenti sono i mattoni sui quali è costruita ogni applicazione Android. Comprenderne caratteristiche, funzionalità, ciclo di vita e le modalità per farli interagire tra loro è il punto di partenza per realizzare un'applicazione.

I componenti principali sono: Activities, Services, Content Providers e Broadcast Receivers, più un componente di collegamento chiamato Intent. Ogni componente è implementato da una classe presente nelle API. Per crearne una propria istanza è necessario estendere tali classi e i suoi metodi.

2.6.1 Activity

La Activity sono il componente principe poiché rappresentano la singola schermata di ogni applicazione Android. Ogni Activity deriva dalla classe `Activity` presente nelle API. La user interface è quindi costituita da un insieme di Activity gestite dall'entità dell'application framework Activity Manager. L'Activity che funge da punto di accesso per l'applicazione è denominata "Main Activity".

Quando un'Activity viene visualizzata quella precedente cambia stato e passa in background; quest'ultima però non viene eliminata, ma mantenuta in uno stack (detto "back stack"), per poi essere riesumata qualora venisse premuto il tasto "back". La navigazione tra le Activity risulta così immediata. Ogni qualvolta un'Activity cambia di stato il sistema chiama un metodo (callback) in cui è possibile gestire l'evento associato. L'insieme delle transizioni di stato di una Activity forma così un grafo chiamato ciclo di vita, la cui comprensione è fondamentale per assicurare un corretto funzionamento all'applicazione.

2.6.1.1 Activity Life Cycle

Una Activity, come detto, può trovarsi in più stati diversi; sostanzialmente quattro:

- Resumed (Running): l'Activity è in foreground e ha il focus dell'utente.
- Paused: l'Activity è visibile nonostante ci sia un'altra Activity in foreground e con il focus dell'utente. Questa situazione accade, ad esempio, quando compare una dialog. L'Activity è comunque viva, ovvero è ripristinabile in qualsiasi istante poiché è ancora presente in memoria e nel back stack.
- Stopped: l'Activity è in background. Anche in questo caso, nonostante non sia più visibile, l'Activity è viva.
- Destroyed: l'Activity è distrutta e non può essere ripristinata in alcun modo, se non ricreandola da zero.

Come già accennato, Android è nato per gestire un ambiente a risorse limitate e quindi si preoccupa lui stesso di terminare quei processi che fanno un uso eccessivo di risorse inutilmente. poiché l'Activity (e il relativo processo) in foreground ha priorità massima, Android potrebbe decidere di terminare le Activity poste in background per liberare risorse. Se si vuole mantenere lo stato dell'applicazione è quindi necessario implementare un sistema che salvi lo stato ogni qualvolta un'Activity viene messa in background e lo ripristini una volta che (l'Activity) viene riposta in foreground. Per fare in modo che un processo in background abbia poche possibilità di essere terminato è quindi necessario che esso non consumi risorse inutilmente. Nella sezione 2.7 verrà approfondita la politica di sistema che regola l'eliminazione dei processi.

In figura è schematizzato il ciclo di vità di un'Activity e i metodi di callback associati ad ogni cambiamento di stato.

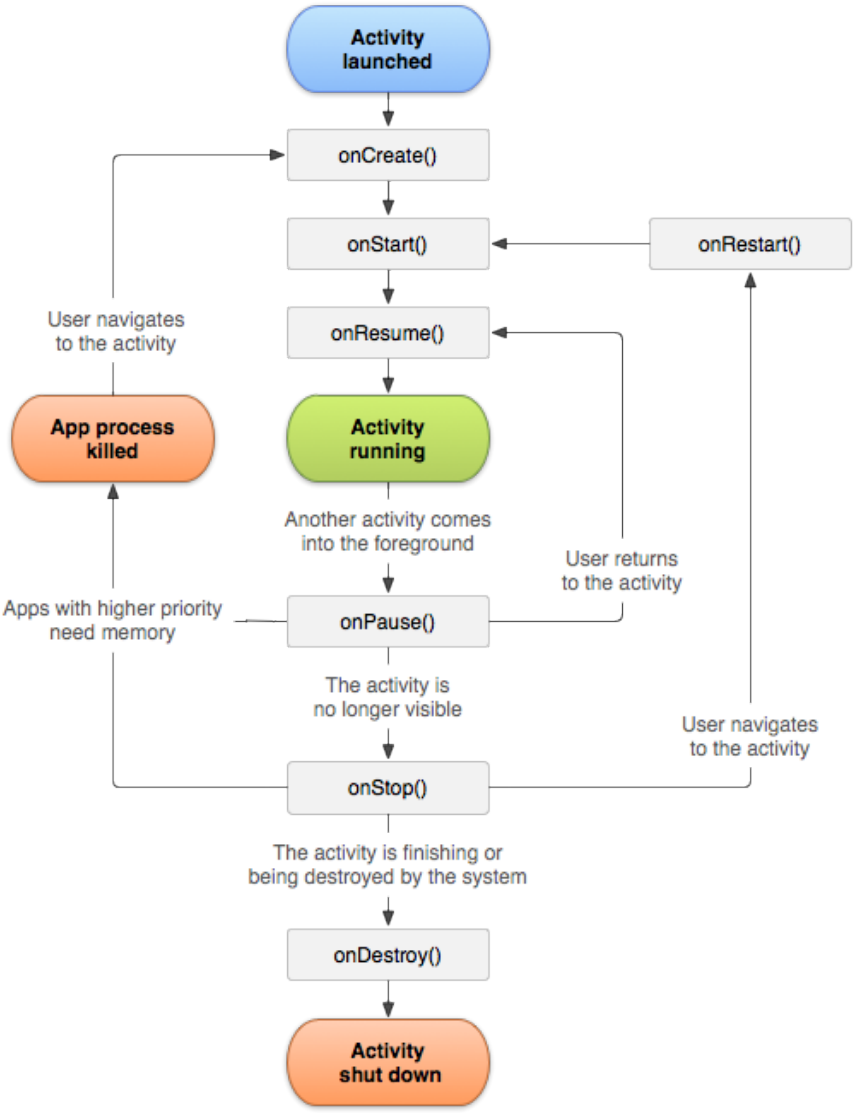

Figura 2.5: Activity Life Cicle

2.6.1.2 User interface

La user interface delle Activity è costituita da oggetti chiamati *view*. Ogni view è un oggetto che estende la classe `View`. La disposizione delle view all'interno della Activity è gerarchica e presenta una struttura ad albero, ricalcando l'idea del DOM di HTML.

Ogni view occupa uno spazio rettangolare che riconosce l'iterazione da parte dell'utente. Le view possono essere classificate in due grandi categorie:

layout e *widgets*. I layout rappresentano i nodi dell'albero e forniscono una disposizione grafica (layout model) per gli elementi figli, mentre le widget sono le foglie e costituiscono i mattoncini base. I layout, avendo una funzione che va oltre il stampare qualcosa a schermo, non estendono direttamente la classe `View` (come invece fanno le widget), ma la classe astratta `ViewGroup`, che a sua volta estende le classe `View`.

Le API di Android forniscono già un numero di view preconfezionate, sia layout (lineare, a griglia, relativo, ecc), che widget (caselle di testo, label, bottoni, ecc), sufficienti per creare un'interfaccia grafica non estremamente complessa. Per ottenere un aspetto e un comportamento completamente personalizzato è necessario estendere le classi di queste view predefinite.

Il modo più comune per definire un layout è utilizzare un file XML, salvato come risorsa dell'applicazione, che ne ricalchi l'albero associato. Un layout può essere creato a runtime anche in modo programmatico all'interno dell'Activity.

Per fare sì che un'Activity carichi un layout definito in un file XML è necessario fare l'override della callback `onCreate()` e utilizzare il metodo `setContentView()`. Come detto è possibile dare dinamicità alla user interface modificando il layout a piacimento nel corso dell'esecuzione.

Per fare in modo che un'applicazione abbia un'interfaccia grafica adeguata a display di dimensioni diverse, Android, dalla versione 3.0, ha messo a disposizione un componente supplementare che si va ad integrare con le Activity: i *Fragment*. Un Fragment è sostanzialmente una porzione di interfaccia grafica che può essere ancorata al layout (associandola ad un ViewGroup) di una Activity in modo dinamico. In questo modo è possibile costruire applicazioni con un'interfaccia grafica che si adatta nel modo migliore alla dimensione del display. Se il display è sufficientemente grande, ad esempio, è possibile adottare un layout di tipo multi-pane, cioè un layout costituito da più Fragment. I Fragment risultano inoltre comodi per cambiare il layout di una Activity a runtime, ad esempio quando avviene una rotazione del display. Usare i Fragments, infine, consente un maggiore riutilizzo del codice, in quanto questi possono essere integrati anche da più Activity.

È possibile utilizzare i Fragment anche se il target del progetto è inferiore alla versione 3.0 in quanto Android mette a disposizione una libreria atta a favorire la retrocompatibilità.

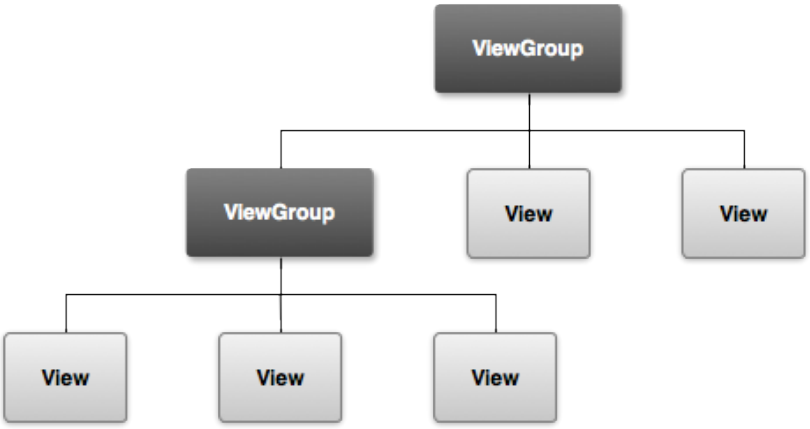

Figura 2.6: Esempio di Layout

2.6.2 Service

I Service sono componenti che consentono di performare operazioni a lungo termine in background. Un Service, non disponendo di una user interface, può continuare la propria esecuzione anche se l'utente passa ad un'altra applicazione. Ogni componente può interagire con un Service, perfino se questo è allocato in un altro processo. Un Service può essere sostanzialmente di due tipi:

- Started: una volta avviato (utilizzando il metodo `startService()`) rimane in esecuzione indefinitamente, anche se il componente che l'ha avviato viene distrutto dal sistema. Solitamente questo tipo di Service vengono utilizzati per performare una singola operazione e non ritornano nulla al componente chiamante.

- Bound: un servizio che permette a qualsiasi altro componente di interfacciarsi a lui utilizzando il metodo `bindService()`. Questo tipo di servizio è paragonabile ad un server dove i client (i componenti interfacciati) possono fare richieste e ottenere risultati. Un Bound Service viene distrutto solamente quando tutti i componenti si sono svincolati (unbind).

I Service, come le Activity, possono essere distrutti dal sistema in caso di carenza di risorse. La politica di eliminazione è, anche in questo caso, prioritaria. Un Bound Service a cui è interfacciata un'Activity in foreground,

ad esempio, ha priorità più alta di uno Started Service, e quindi ha meno probabilità di essere distrutto. Un Service, per avere la certezza di non essere eliminato dal sistema, deve essere dichiarato "run in foreground". Da queste considerazioni si intuisce come i Service abbiano un ciclo di vita legato a quello delle Activity interfacciate. In figura è possibile vedere tale ciclo di vita e i metodi di callback che rappresentano i cambi di stato.

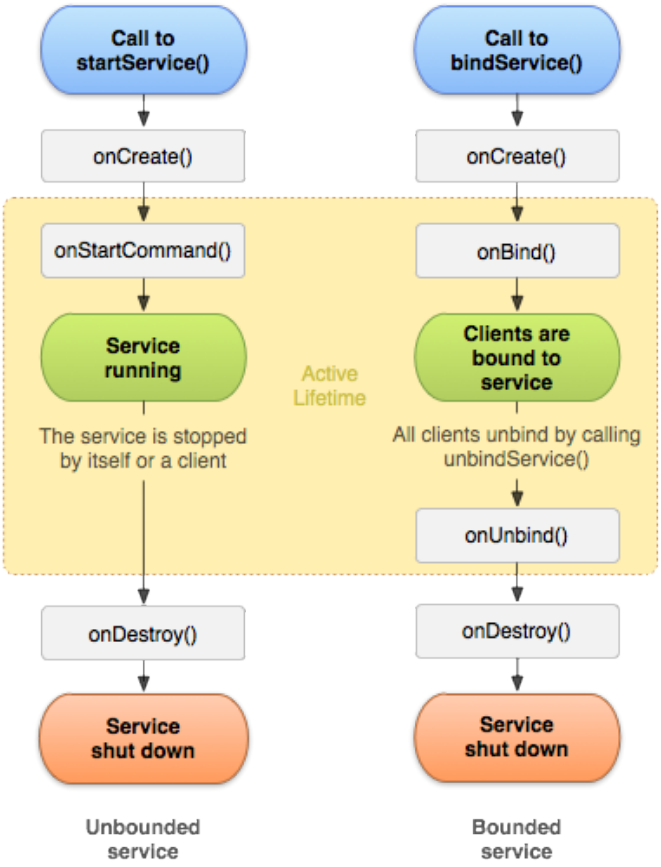

Figura 2.7: Ciclo di vita di un Service Unbounded (Starded) e di un Service Bounded

2.6.3 Content Provider

I Content Provider, come il nome suggerisce, gestiscono l'accesso a dati strutturati, in quanto, oltre a fornirne meccanismi di incapsulazione, provvedono a definire delle politiche di sicurezza per il loro uso. Per utilizzare un Content Provider è necessario utilizzare un oggetto di tipo ContentResolver che,

svolgendo il ruolo di client, si occupa di comunicare con il Provider stesso. Ad ogni modo utilizzare i Content Provider non è necessario se i dati interessati devono essere utilizzati solo localmente all'interno dell'applicazione. Implementare un Content Provider ha quindi senso solo in un'ottica di condivisione, cioè se si vuole fare in modo che i dati vengano utilizzati anche da altre applicazioni. Android stesso mette a disposizione Content Provider per gestire audio, video, immagini e le informazioni dei contatti. Non dovendo condividere i dati con altre applicazioni il componente in questione non è stato utilizzato nel mio lavoro e quindi non verrà approfondito ulteriormente.

2.6.4 Broadcast Receiver

I Broadcast Receiver sono componenti atti a monitorare e gestire e le comunicazioni provenienti dal sistema o dalla stessa applicazione. Un broadcast receiver, ad esempio, può essere usato per monitorare lo stato della batteria o lo spegnimento dello schermo. Questi componenti non dispongono di una interfaccia grafica, ma permettono di segnalare all'utente tale evento, utilizzando, ad esempio, una notifica nella status bar. I Broadcast Receiver, come detto possono ricevere messaggi provenienti anche da altri componenti della stessa applicazione. Per comprenderne il funzionamento è però necessario introdurre l'elemento che consente ai vari componenti di interagire tra loro, e che in questo caso, rappresenta il messaggi catturati: gli Intent.

2.6.5 Intent

Gli Intent sono oggetti che consentono ai componenti di avviarsi vicendevolmente: possono essere quindi intesi come messaggi asincroni di avvio. Un componente, utilizzando un Intent, può avviare anche un componente di un'altra applicazione, se quest'ultimo è predisposto per esaudire tale richiesta. Un Intent contiene sempre una descrizione astratta dell'operazione da eseguire, e in aggiunta, può incapsulare anche dati.

Gli Intent possono essere divisi in due gruppi principali:

- Explicit Intents: il target del Intent è definito e specificato nel campo "Component name". Tale campo contiene quindi il nome della classe che rappresenta il componente da avviare. poiché i nomi dei componenti non sono noti alle altre applicazioni questo tipo di Intent è usato esclusivamente all'interno dell'applicazione.

- Implicit Intents: il target non è esplicitamente definito ("Component name" nullo) ma è specificato il tipo di azione (action) richiesta. Sarà Android a determinare quali componenti dovranno esaudire tale richiesta. Il SO per fare tale scelta confronterà la action (più altre informazioni) dell'Intent con gli Intent Filter di ogni componente. Un Intent Filter è un oggetto associabile ad ogni componente che definisce il tipo di richiesta (più altre informazioni) che quel componente può esaudire. Un componente in cui non è definito alcun Intent Filter può quindi ricevere solamente Explicit Intent. Un Broadcast Receiver atto a monitorare lo stato della batteria dovrà quindi, ad esempio, definire un Intent Receiver che contenga la action definita da Android per segnalare lo stato della batteria.

2.7 Processi e Threads

Come già accennato ad ogni applicazione Android è associato un processo costituito da un unico thread (Main, o UI, thread). Questo comportamento di default è modificabile in modo che un componente possa essere ospitato da un determinato processo o thread. Nella grande maggioranza dei casi non ha senso assegnare più processi alla stessa applicazione, mentre, come vedremo, è necessario utilizzare il multithreading.

Android utilizza un sistema a priorità per determinare quei processi che vanno terminati causa scarsità di risorse. Nello specifico, vi sono cinque livelli di priorità, che elenco brevemente. I processi a priorità più bassa sono i primi ad essere eliminati. Partendo dalla priorità più alta abbiamo:

1. Foreground process. Processi necessari per l'utilizzo corrente del sistema da parte dell'utente. Sono sostanzialmente i processi che ospitano Activity in foreground (`onResume()` è l'ultima callback invocata), i Service interfacciati con tali Activity, e i Broadcast Receiver che stanno ricevendo un messaggio;

2. Visible process. Processi che ospitano componenti (Activity) non in foreground ma comunque visibili (`onPause()` è l'ultima callback invocata) e i Service interfacciati con tali Activity;

3. Service Process. Processi che ospitano Service non appartenenti alle due categorie appena citate;

4. Background process. Processi che ospitano Activity in background (`onStop()` è l'ultima callback invocata) e Service associati.

5. Empty process. Processi che non contengono alcun componente. Solitamente usati per funzioni di caching.

Essendo l'Android UI toolkit non thread-safe una regola fondamentale da tenere sempre in mente quando si usa il multithreading è non manipolare la user interface da thread diversi (detti worker thread) del "main thread". Nelle sezioni 2.12, 2.12.1 sarà spiegato perché è necessario utilizzare il multithreading e saranno illustrate le tecniche per manipolare la UI dai worker thread.

2.8 Android Manifest

Il file `AndroidManifest.xml` ha lo scopo di dare una rappresentazione dell'applicazione al sistema operativo. In modo particolare il Manifest provvede a:

- definire i Java package i cui nomi servono come identificatore unico per l'applicazione;
- descrivere i componenti definendo i nomi delle classi e pubblicando le loro capacità (es. Intent Receiver);
- determinare quali processi ospitano i componenti;
- dichiarare i permessi per accedere a parti protette delle API e per regolare l'iterazione con altre applicazioni;
- dichiarare il livello minimo di API che l'applicazione richiede e la versione target.
- elencare le librerie esterne necessarie all'esecuzione.

L'immagine 2.8sottostante mostra la struttura generale del file Manifest e gli elementi che può contenere.

```
<?XML version="1.0" encoding="utf-8"?>

<manifest>
   <uses-permission />
   <permission />
   <permission-tree />
   <permission-group />
   <instrumentation />
   <uses-sdk />
   <uses-configuration />
   <uses-feature />
   <supports-screens />
   <compatible-screens />
   <supports-gl-texture />

   <application>
      <Activity>
         <intent-filter>
            <action />
            <category />
            <data />
         </intent-filter>
         <meta-data />
      </Activity>
      <Activity-alias>
         <intent-filter> . . . </intent-filter>
         <meta-data />
      </Activity-alias>
      <service>
         <intent-filter> . . . </intent-filter>
         <meta-data/>
      </service>
      <receiver>
         <intent-filter> . . . </intent-filter>
         <meta-data />
      </receiver>
      <provider>
         <grant-uri-permission />
         <meta-data />
      </provider>

      <uses-library />
   </application>
</manifest>
```

Figura 2.8: Android Manifest

2.9 Gestione delle risorse

Le risorse di un'applicazione Android vanno esternalizzate e organizzate in modo opportuno in sottocartelle della cartella res del progetto. Esternalizzare le risorse ha due grandi vantaggi: il primo è che vengono rese indipendenti dal codice; il secondo, e più importante, è che in questo modo è possibile fornire compatibilità tra vari dispositivi. Su Android è infatti possibile specificare risorse alternative a seconda della configurazione (es. lingua) e delle proprietà (es. grandezza display) dei device sui quali l'applicazione verrà eseguita.

Per raggruppare le risorse Android utilizza un sistema di naming per le sottocartelle che fa uso di qualificatori atti a specificare ogni proprietà o configurazione possibile. La gestione è affidata completamente al SO, in quanto è lui stesso che si occupa in automatico di caricare la giusta risorsa a runtime; lo sviluppatore deve solo crearla, nel caso sia un file XML, e porla nella giusta sottocartella.

Android consente di definire 9 macro-tipi di risorse: *animator*, *anim*, *color*, *drawable*, *layout*, *menu*, *raw*, *values*, *XML*. Ad ogni tipo corrisponde una sottocartella della cartella res avente ugual nome. In modo particolare le risorse contenute nella cartella values sono dette semplici e, come vedremo, sono referenziate in modo diverso dalle altre. Le risorse contenute nelle sottodirectory appena citate sono risorse di default poiché i nomi delle cartelle non presentano alcun qualificatore[2]. È inoltre possibile specificare più qualificatori contemporaneamente, separandoli con un trattino, per ottenere una configurazione obiettivo più selettiva. Come detto esistono qualificatori che consentono di creare raggruppamenti che coprono ogni tipo di dispositivo in qualunque configurazione.

Per quel che riguarda la localizzazione, ad esempio, Android mette a disposizione qualificatori per ogni paese e regione, nonché per MCC (mobile country code) e MNC (mobile network code). Questi qualificatori saranno verosimilmente applicati alla sottocartella values poiché è in essa che solitamente si definiscono le risorse di tipo testo di un'applicazione. Per fare un esempio, se un'applicazione deve supportare sia inglese che italiano, come nel mio caso, è necessario creare la sottodirectory values-it (oppure values-en) oltre alla cartella values di default.

[2] I qualificatori sono posti come desinenza nel nome di una sottocartella

Per supportare device differenti esistono qualificatori per la grandezza e la risoluzione del display, per l'input (touchscreen o no), ecc. In questo caso le subdirectory interessate saranno drawable (contenente i bitmap file, più altre risorse grafiche) e layout (contenente i file XML per definire i layout).

Per accedere alle risorse dal codice si utilizza un identificatore univoco (ID) che viene generato in automatico dal tool AAPT di Eclipse. L'insieme degli identificatori è definito nel file `R.java`, file creato per l'appunto dall'AAPT e collocato nella cartella `gen`. Un ID ha la seguente struttura: `R.resource-type.resource-name`, dove il `resouce-type` rappresenta il tipo, ad esempio *`drawable`* o *`menu`*, mentre il `resource-name` può essere, o il valore del attributo `android:name` se la risorsa è di tipo semplice (macro-tipo values), oppure il nome del file se la risorsa appartiene agli altri macro-tipi.

2.10 Data Storage

Android fornisce diverse opzioni per salvare i dati di un'applicazione in modo persistente. Ogni opzione è finalizzata ad un determinato tipo di dati, alla sua mole e alle finalità che questi hanno. Le possibilità sono tre e sono elencate in seguito:

- Shared Preferences: contengono solo dati di tipo primitivo sotto forma di coppie chiave-valore. Sono solitamente usate per salvare le preferenze dell'applicazione.

- File Storage: i dati sono salvati in file che possono risiedere, o nella memoria interna del dispositivo, oppure in memoria esterna (ad esempio una micro SD). Nel primo caso i dati sono privati, in quanto solo l'applicazione ha accesso diretto ad essi; altre applicazioni possono accedervi solo via Content Provider. I file vengono eliminati al momento della disinstallazione dell'applicazione.

  Se i dati vengono salvati esternamente questi sono pubblici e totalmente accessibili. I dati vengono eliminati al momento della disinstallazione solo se sono stati salvati nella cartella in memoria esterna associata all'applicazione.

- Database: Android implementa nativamente un RDBMS già integrato nel sistema operativo: SQLite. Le API forniscono allo sviluppatore

tutti gli strumenti necessari per creare, mantenere ed eseguire le operazione CURD sui db di un'applicazione. Solo l'applicazione che ha creato il db può accedere ai dati, a meno che questa non predisponga dei Content Provider.

Nel proseguo della tesi verranno riprese queste tecniche ed in particolare sarà spiegato in che modo sono state utilizzate nel progetto.

2.11 Android vs. Java ME

Questa sezione ha lo scopo di introdurre le differenze principali tra la programmazione su Java ME e quella su Android. Come abbiamo accennato nei precedenti capitoli effettuare il porting non implica una riprogrammazione radicale ma piuttosto una sorta di adattamento del codice alla nuova piattaforma. L'adattamento è dovuto sostanzialmente al fatto che Android e Java ME non forniscono lo stesso development kit, e quindi le stesse API. In questo senso la parte di user interface e data storage deve essere completamente riscritta.

Per quel che riguarda la UI l'SDK di Android prevede l'utilizzo dei componenti delle API visti precedentemente (Activity, Fragment, Layout e Widget); mentre nei progetti MBM su Java ME si utilizzano le AWT. Inoltre su Android, come già accennato, per definire i layout è possibile utilizzare due approcci: procedurale e dichiarativo. La via programmatica prevede che i layout siano definiti nel codice Java, utilizzando le librerie fornite dalle API; l'approccio dichiarativo invece, pur utilizzando sempre gli stessi componenti, prevede che essi siano definiti in file XML. Questo secondo approccio è preferibile poiché consente una migliore separazione tra ciò che è interfaccia utente, quindi riferibile al presentation tier, e la logica business.

Anche la parte di data storage, fondamentale per garantire la criticità delle applicazioni, è stata completamente rivista, in quanto su Android la via più naturale per salvare dati in modo persistente è, come visto, utilizzare SQLite. SQLite, essendo nativamente integrato nel sistema, non prevede che l'accesso avvenga utilizzando un driver JDBC comune, ma le classi delle API.

Inoltre, su Android la programmazione è legata a dei componenti software forniti dalle API e pattern architetturali da cui non si può prescindere. Anche

gli aspetti riguardanti sicurezza, ciclo di vita dell'applicazione, gestione delle risorse e dell'internalizzazione vanno approcciati in modo totalmente diverso.

2.12 Design Pattern

Quando si programma un'applicazione per Android, ma in generale per un dispositivo mobile, bisogna tenere a mente alcune considerazioni che derivano dalla natura degli stessi dispositivi . Questi infatti hanno caratteristiche hardware inferiori ai pc e quindi l'attenzione deve essere rivolta in primo luogo alla gestione delle (limitate) risorse. Inoltre bisogna tenere conto di aspetti ininfluenti in altri ambienti, come, ad esempio, la grandezza variabile dei display sui quali le applicazioni verranno eseguite; oppure il fatto che la user input interface può variare a seconda che il dispositivo sia touchscreen o presenti una tastiera fisica.

Per far si che la presentazione dei contenuti sia chiara a prescindere dal display è necessario seguire delle linee guida comuni. Queste sono presenti nella sezione "Design" del sito sviluppatori di Android e riguardano un po' tutto quello che è user interface: disposizione dei contenuti in una schermata, navigazione all'interno dell'applicazione, gestione degli input (gesture), visualizzazione delle notifiche, ecc. Seguire questi pattern è inoltre necessario se si vuole raggiungere una certa omogeneità nel mondo sconfinato delle applicazioni Android. Seguire le linee guida è quindi la chiave per creare un'applicazione che vada incontro all'utente generico in cui quella applicazione è una delle tante presenti nel suo device, verosimilmente uno smartphone o un tablet. Se invece l'applicazione è destinata ad un gruppo ristretto di utenti ed è pressoché la sola che viene utilizzata, come nel mio caso, il concetto di omogeneità non ha più molto senso. Ad ogni modo io ho voluto rimanere il più possibile conforme a questi standard, in primo luogo perché, essendo stati accuratamente studiati, si ha la certezza che essi producano un buon risultato, poi perché le API e la documentazione incoraggiano lo sviluppatore ad adottarli.

Come detto su Android la gestione efficiente delle risorse è fondamentale. Anche qui Google suggerisce di adottare determinati pattern architetturali, ma se nella progettazione della user interface si può un po' uscire dagli schemi, in questo caso è bene non farlo causa il mal funzionamento dell'applicazione e dell'intero sistema.

Una cosa da tenere sempre ben presente è il concetto di responsiveness: il sistema operativo Android, infatti, impone ad ogni applicazione di essere reattiva, e se questa non lo è compare un messaggio di errore chiamato in gergo ANR (Android Not Responding). Chiaramente il presentarsi di questa situazione non è accettabile poiché l'applicazione viene sostanzialmente bloccata. Ma cos'è che pregiudica la reattività? La responsiveness viene a mancare quando operazioni onerose dal punto di vista del consumo delle risorse (es trasferimento dati) o bloccanti (es. connessioni alla rete) vengono eseguite nel thread principale, o Main Thread, dell'applicazione. Il Main Thread ospita l'applicazione al suo avvio e il suo scopo principale è quello di gestire tutto ciò che è user interface. La regola, al fine di non sovraccaricare il thread principale, è quindi quella di sfruttare il multithreading per eseguire le operazioni sopra citate. Utilizzare il multithreading ha però delle limitazioni, una di queste, che abbiamo già accennato, è che dai worker thread non è possibile modificare gli elementi della user interface, pena errore fatale dell'applicazione. Le API, come vedremo nella prossima sezione, mettono a disposizioni classi e metodi atti a rendere più semplice la gestione del multithreading all'interno dell'applicazione.

2.12.1 Multi Threading e UI

Abbiamo visto come sia fondamentale utilizzare il multithreading se si vuole realizzare un'applicazione fluida e senza errori ANR. Per creare un worker thread concorrente al main thread è sufficiente estendere la classe `Thread` ed implementare il metodo `run()`. Questa soluzione è la più semplice, ma all'interno del metodo `run()`non è possibile manipolare gli elementi della UI. Le API per eliminare questa limitazione mettono a disposizione i seguenti metodi e classi:

- `runOnUiThread(Runnable)` della classe `Activity`.
- `post(Runnable)` della classe `View`.
- `postDelayed(Runnable, long)` sempre della classe `View`; il secondo parametro indica il tempo di cui ritardare l'esecuzione.
- `Handler`. Questa classe merita un approfondimento perché è stata ampiamente usata nel progetto.

Prima di introdurre la classe `Handler` è però necessario spiegare come su Android in ogni thread sia possibile creare una coda di messaggi con logica FIFO[3]. A differenza del main thread, il sistema non associa di default ai worker thread questa coda, per cui se c'è questa necessità è compito dello sviluppatore crearla. Per fare ciò si utilizzano i metodi statici `prepare()` e `loop()` della classe `Looper`. Tale classe implementa sostanzialmente un loop per ricevere i messaggi dagli Handler e inviarli alla coda del relativo thread. Un oggetto della classe `Handler` è quindi lo strumento che consente di porre dei messaggi in coda al thread in cui viene istanziato. Un Handler può postare due tipi di messaggi: oggetti della classe `Message`, oppure della classe `Runnable`. Mentre i primi vengono gestiti dentro il metodo `handleMessage()`, i secondi, essendo istanze della classe `Runnable`, vengono eseguiti automaticamente dal sistema nel momento in cui vengono estratti dalla coda.

Il listato sottostante mostra l'implementazione standard di un LooperThread, ovvero un thread dotato di Message Queue.

Listato #2.1

```
class LooperThread extends Thread {
    public Handler mHandler;
        public void run() {
            Looper.prepare();
            mHandler = new Handler() {
                public void handleMessage(Message msg) {
                    // process incoming messages here
                }
        };
        Looper.loop();
    }
}
```

Le API mettono a disposizione una classe, `HandlerThread`, che ricalca il comportamento di Looper Thread.

Il listato sottostante (#2.2), invece, mostra come è possibile manipolare la UI da un worker thread utilizzando gli Handler. Non è necessario predisporre la Message Queue in quanto nel main thread è già implementata di default.

[3]La coda di messaggi è implementata dalla classe `MessgeQueue`

Listato #2.2

```
public class MyActivity extends Activity {
    [ . . . ]

    // Need handler for callbacks to the UI thread
    final Handler mHandler = new Handler();

     // Create runnable for posting
    final Runnable mUpdateResults = new Runnable() {
        public void run() {
            updateResultsInUi();
        }
    };

     @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
            [ . . . ]
    }

     protected void startLRO() {
         // Fire off a thread to do some work that we shouldn
             't do directly in the UI thread
        Thread t = new Thread() {
            public void run() {
                mResults = doLRO();
                mHandler.post(mUpdateResults);
            }
        };
        t.start();
    }
     private void updateResultsInUi() {
         // Back in the UI thread -- update our UI elements
             based on the data in mResults
        [ . . . ]
    }
}
```

L'esempio mostra come sia possibile utilizzare un worker thread per eseguire un'operazione onerosa e al termine aggiornare la UI. Questo ti-

po di operazione è abbastanza comune in un'applicazione Android; in modo particolare nel nostro progetto, come vedremo, è stata utilizzata molto spesso.

Fortunatamente le API mettono a disposizione uno strumento, AsyncTask, che permette di replicare il comportamento dell'operazione appena menzionata senza la necessità di creare appositamente thread e Handler, poiché questi sono stati mascherati nella sua implementazione. AsyncTask è implementato dall'omonima classe e il suo funzionamento è illustrato in figura 2.9. Come è possibile vedere utilizza delle callback (`onPreExecute()`, `doInBackground()`, `onProgressUpdate()`, e `onPostExecute()`) di sistema, dove la prima e le ultime due vengono eseguite nel Main Thread, mentre la seconda (`doInBackground()`) viene eseguita da un worker thread creato dal sistema al momento dell'avvio dell'AsyncTask[4]. AsyncTask è uno strumento molto potente perché garantisce la sincronizzazione tra i thread coinvolti, ovvero `doInBackground()` viene chiamato dal sistema solo al termine di `onPreExecute()`, mentre `onPostExecute()` è invocato solo dopo che il worker thread ha terminato l'esecuzione di `doInBackground()`. `onProgressUpdate()` invece viene chiamato solo se in `doInBackground()` viene invocato il metodo `publishProgress()`.

[4] Un AsyncTask viene avviato utilizzando il metodo `execute()`

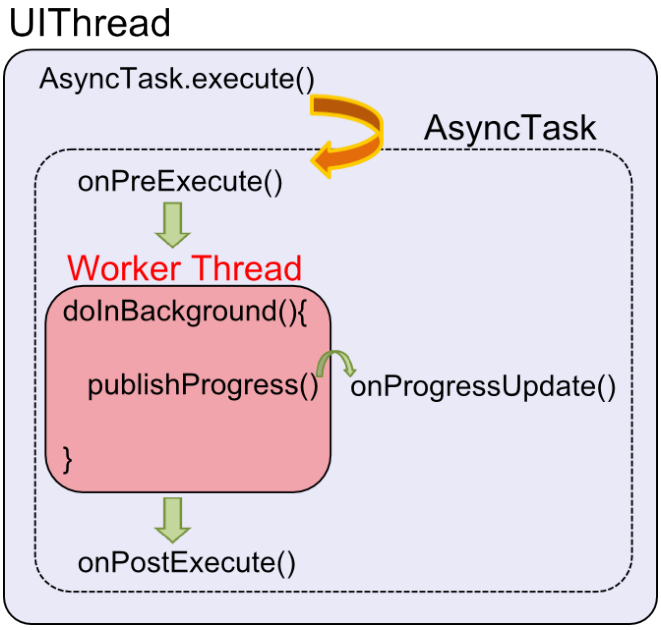

Figura 2.9: Sequenza di callback in un AsyncTask

Il listato sottostante, invece, mostra come è possibile implementare AsyncTask affinché replichi il comportamento del listato precedente.

Listato #2.3

```
public class MyActivity extends Activity {
    [ . . . ]

     @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
            [ . . . ]
    }

    // LRO= Long Running Operation
     protected void startLRO() {
         // Fire off a thread to do some work that we shouldn
             't do directly in the UI thread
        new LROTask().execute();
    }

    private class LROTask extends AsyncTask<Void, Void,
        Object> {
```

```
        @Override
        protected Object doInBackground(Void... params) {
            mResults = doLRO();
            return mResults;
        }

        @Override
        protected void onPostExecute(Object mResults) {
            updateResultsInUi();
        }
    }
}
```

Capitolo 3

Implementazione

3.1 Implementazioni richieste

Ora che è stato introdotto Android in modo più approfondito possiamo presentare le funzionalità applicative che sono state oggetto di ricerca nella fase iniziale al fine di implementarle in *WCubeApp*. Parte di queste funzionalità sono comuni a qualsiasi applicazione e quindi sono state inserite nel framework *MbmDroid*, altre invece sono specifiche dell'applicazione *WCube*.

I problemi che mi erano stati posti riguardano i seguenti temi:

- Data storage. Trattandosi di applicazioni "mission critical" è necessario salvare i dati in modo persistente localmente. Il modulo adibito alla gestione del db deve quindi provvedere a gestire la logica di accesso, le transazioni, e supportare le operazioni CURD sulle tabelle.
- User Interface. In particolare visualizzazione dati in forma di tabella/griglia e dialog. In entrambi i casi c'era l'esigenza di implementare componenti che fossero riutilizzabili in più schermate dell'applicazione.
- Gestione della configurazione e delle preferenze - L'applicazione deve essere configurabile con un file di properties o XML. Parte di queste configurazioni devono essere viste come preferenze modificabili dall'utente.
- Comunicazione con il server. Modulo che gestisce la comunicazione con il server ed in particolare con il Web Service.

- Gestione di thread secondari. L'applicazione all'avvio lancia un pool di thread per svolgere determinate funzionalità di supporto. Questi thread dialogano con il server in background e interferiscono con la UI in modo minimale e solo in particolari circostanze. I principali sono il LUW Thread e il PalmClient. Mentre il primo gestisce le LUW, il secondo si occupa di interagire con il PalmMonitor, uno strumento lato server che consente di monitorare il palmare (eseguire query sul db, visualizzare i log, ecc).

- Gestione dei log - Il logging deve essere eseguito in maniera accessibile e condizionato dal livello di debug richiesto.

- Gestione dello scanner per i barcode. Modulo che gestisce lo scanner di codici a barre se questo è presente nel dispositivo.

3.2 Data storage

In questa sezione vediamo come è stato implementato il data tier dell'applicazione e l'iterazione con la logica business.

Nel business tier i dati sono ovviamente oggetti. In MBM, in particolare, sono oggetti di una classe "Data", ovvero una classe avente questo tipo di desinenza. Nel data tier i dati sono contenuti nelle tabelle (relazioni) che costituiscono il db dell'applicazione. Nel progetto *WCube* ad ogni classe "Data" corrisponde una relazione nel db. Un oggetto di una classe "Data" rappresenta quindi nella logica business una tupla della relativa relazione.

L'architettura prevede inoltre che la logica business non si interfacci direttamente con il database per recuperare i dati, ma utilizzi dei DAO (Data Access Object), che a loro volta accedono al database. I DAO svolgono quindi il ruolo di middleware tra business tier e data tier. poiché ad ogni classe "Data" è associata una relazione, e nelle applicazioni non sono previste query che fanno uso di join, non è stato realizzato un unico DAO, ma uno per ogni coppia classe-relazione. L'impiego dei DAO rende l'architettura dell'applicazione più modulare e quindi più facilmente riutilizzabile e adattabile a diverse implementazioni.

Le applicazioni MBM realizzate nella piattaforma Java ME prevedono che i siano salvati in modo persistente in un db locale a cui si accede via JDBC. L'applicazione *WCubePalm*, ad esempio, utilizza un db Oracle. poi-

ché Android deve necessariamente utilizzare SQLite il data tier è stato completamente riadattato al nuovo DBMS, mentre la modularità garantita dal pattern appena citato ha fatto si che le modifiche fossero apportate solo ai DAO, lasciando intaccata la logica business. Come si può intuire nella logica di eseguire il porting dell'applicazione questo si presenta come un grandissimo vantaggio.

Per non stravolgere completamente la logica delle applicazioni questo pattern architetturale è stato riportato tale e quale su Android. Vediamo come.

3.2.1 SQLite e Android

Come detto SQLite è integrato nel SO Android in modo nativo. Le API, in questo senso, mettono a disposizione tutto ciò che serve per creare, gestire ed utilizzare il database.

Uno degli aspetti che differenzia SQLite dalla maggior parte dei database relazionali è la tipizzazione dei dati.

Data types I tradizionali RDBMS usano una tipizzazione statica e rigida dove il tipo è determinato dalla colonna della tabella in cui il dato è stato inserito. SQLite in questo senso affronta il problema della tipizzazione in modo più dinamico e flessibile poiché ad un dato viene attribuito il tipo in base al valore del dato stesso.

Ogni dato immagazzinato nel database appartiene ad una delle seguenti classi:

- NULL: il valore è nullo;
- INTEGER: il valore è un intero con segno, immagazzinato in 1, 2, 3, 4, 6, o 8 byte a seconda della sua grandezza;
- REAL: il valore è numerico con decimali, immagazzinato come un 8-byte IEEE floating point number;
- TEXT: il valore è una stringa di testo, immagazzinato secondo la codifica del database (UTF-8, UTF-16BE or UTF-16LE);
- BLOB: il valore è un "blob", immagazzinato esattamente come è stato inserito.

Si può notare come il concetto di classe classi sia più generale di quello di un datatype tradizionale. La classe INTEGER, ad esempio, racchiude 6 tipi di dato diversi.

Per massimizzare la compatibilità tra SQLite e gli altri DBMS, SQLite supporta il concetto di affinità sulle colonne. L'affinità su una colonna raccomanda, non impone, quale tipo/i di dato (classe) dovrebbe essere inserito in quella colonna. Ogni colonna di un database SQLite potrebbe quindi contenere valori appartenenti a qualsiasi classe. Ad ogni colonna è assegnata una delle seguenti affinità: TEXT, NUMERIC, INTEGER, REAL, NONE. La tabella3.1 riassume la relazione tra tipo di dato di SQL, affinità e classe di dato SQLite.

Typenames From The CREATE TABLE Statement or CAST Expression	Resulting Affinity	Preferred Class
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER	all
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT	NULL TEXT BLOB
BLOB no datatype specified	NONE	all
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL	all
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC	all

Tabella 3.1: Affinità dei tipi di dato

È importante notare come nessun tipo di dato predisponga controlli sulla lunghezza del valore inserito[1]. Ciò implica che questo tipo di controllo sul dato deve essere fatto dall'applicativo prima del suo inserimento nel db.

Android API Introduciamo ora in che modo le API si integrano con SQLite. Le classi che consentono ad un'applicazione Android di utilizzare SQLite sono sostanzialmente tre: `SQLiteOpenHelper`, `SQLiteDatabase`, `Cursor`.

[1] a meno del limite globale SQLITE_MAX_LENGTH

- `SQLiteOpenHelper`: classe astratta che funge da helper alla creazione dei database e all'eventuale aggiornamento del db. La classe implementa le callback del SO designate alle operazioni appena citate e i metodi per ottenere una connessione. Per ogni database deve essere creata un'implementazione di `SQLiteOpenHelper`.

- `SQLiteDatabase`: classe che rappresenta una connessione al generico database. Questa classe fornisce i metodi per eseguire le operazioni CURD. Facendo il confronto con le API `java.sql` può essere paragonabile alla `Connection`.

- `Cursor`: classe che rappresenta il result set di una query. Anche qui è possibile un paragone con una classe del package `java.sql.ResultSet`.

Essendo la nostra applicazione multithreading bisogna porre molta attenzione a come vengono ottenute le connessioni al db, SQLite infatti non garantisce che le connessioni siano thread-safe di default. Su Android, come vedremo, per garantire al db questa fondamentale caratteristica è necessario implementare la classe `SQLiteOpenHelper` secondo un pattern ben preciso che vedremo nelle prossime righe.

3.2.2 Implementazione

Vediamo ora come gli strumenti appena presenti ci hanno permesso di realizzare il modulo che racchiude il data tier dell'applicazione e i DAO. Il modulo è composto dalle seguenti classi:

MyDBOpenHelper poiché le applicazioni fanno uso di un solo database è stato sufficiente creare un'unica implementazione di `SQLiteOpenHelper`, chiamata `MyDBOpenHelper`. affinché il db sia thread safe è necessario che all'interno dell'applicazione si utilizzi sempre la stessa istanza di `MyDBOpenHelper`. Il motivo risulterà chiaro tra poche righe. La classe `MyDBOpenHelper` è stata quindi implementata come un singleton in cui l'unica istanza viene restituita da un metodo statico. La classe `SQLiteOpenHelper` implementa una callback (`onCreate(SQLiteDatabase db)`) adibita a creare lo schema del database. Tale metodo viene invocato dal SO quando l'applicazione richiede una connessione al db ma questo non è ancora stato creato.

Una connessione, come già anticipato, non è altro che un'istanza della classe `SQLiteDatabase`. Per ottenere una una connessione è sufficiente invocare il metodo `getReadableDatabase()` (o `getWritableDatabase()`) dell'oggetto `MyDBOpenHelper`. Tali metodi hanno la caratteristica di restituire sempre una referenza allo stesso oggetto. Come accennato un oggetto della classe `SQLiteDatabase` fornisce tutti i metodi per eseguire le operazioni CURD sul database. Ma c'è di più: esso garantisce anche che tali operazioni siano thread safe, in poche parole serializzate. Da qui si comprende il motivo per cui è fondamentale creare una sola istanza di `SQLiteOpenHelper`: per far si che venga restituita sempre la stessa connessione, ovvero lo stesso oggetto `SQLiteOpenHelper`.

Poiché le applicazioni costruite sul framework avranno db con schemi diversi risulta chiaro che ognuna dovrà implementare il proprio `MyDBOpenHelper`. Per questo motivo la classe non fa parte del framework ma sarà bensì contenuta all'interno del progetto di ogni singola applicazione. Nel mio caso specifico è stata riportata nel progetto *WCubeApp*.

DBManager Classe statica adibita a fornire metodi di helper per ottenere/rilasciare connessioni ed eseguire transazioni sul db. Questa classe, se non fosse per il fatto che fa riferimento alla classe `MyDBOpenHelper` avrebbe anche valenza generale e quindi potrebbe fare parte del framework. Per ovviare al problema e mantenere tale classe nel progetto di framework si è deciso di utilizzare una Factory (`SQLiteDatabaseFactory`) di supporto che restituisce l'istanza del `MyDBOpenHelper` utilizzando la reflection.

I DAO Come accennato ad ogni tabella/classe è associato un DAO, cioè una classe statica i cui metodi vengono invocati dalla logica business per eseguire operazioni su quella particolare tabella del db. Ai metodi del DAO vengono passati come parametri la connessione e le informazioni da inserire nella query. Chiaramente i DAO sono stati riscritti per fare si che si adattassero ai nuovi componenti. Il nomi delle classi DAO terminano con "Home".

SQLMapper Questa classe di supporto ha lo scopo di generalizzare e automatizzare il mapping tra il result set (Cursor) di una query e una ArrayList i cui oggetti di classe "Data" sono le tuple della tabella interrogata. Più con-

cretamente `SQLMapper` è una sottoclasse di `ArrayList` parametrizzata secondo il tipo generico `RowShadow`, che non è altro che la superclasse (astratta) di tutte le classi "Data" presenti nell'applicazione. `SQLMapper` è stata ampiamente utilizzata nei DAO per restituire alla logica business il risultato di una query sottoforma di una `ArrayList` di oggetti "Data". Sia `SQLMapper` che `RowShadow` per la loro generalità sono state inserite nel framework.

Il listato sottostante mostra il metodo del DAO (classe `MovementHome`) che restituisce una `ArrayList` contente tutte le movimentazioni (oggetti di classe `MovementData`) presenti nella relativa tabella (MovementTable) del db.

Listato #3.1

```
public static ArrayList<MovementData> selectAllMovement(
    SQLiteDatabase db) throws SQLException {
    ArrayList<MovementData> vector = null;
    Cursor cursor = null;
    try{
        cursor = db.rawQuery("select * from MovementTable",
            null);
        vector = new SqlMapper<MovementData>(cursor,
            MovementData.class);
    }finally{
        if(cursor != null)
            cursor.close();
    }
    return vector;
}
```

3.3 User interface

Come accennato nell'introduzione, la user interface è stata completamente riscritta in quanto Android non supporta le AWT, ma prevede che vengano utilizzati i componenti forniti dalle API.

L'approccio standard per eseguire il porting della UI prevederebbe che ogni schermata dell'applicazione (cioè un oggetto `java.awt.Panel`) venga convertito in una Activity. La UI delle applicazioni mobile MBM era progettata per dispositivi (PDA) con display di dimensioni con caratteristiche simili. Con la scusa del porting l'azienda però voleva rivisitare questo ap-

proccio così limitativo e quindi l'obiettivo era quello di predisporre l'applicazione all'utilizzo su vari dispositivi: ovvero la UI doveva essere facilmente adattabile a display di dimensioni diverse. Nella progettazione questo si traduce nell'utilizzare i Fragment. Questi componenti consentono di rendere più modulare l'interfaccia grafica lasciando alle Activity il compito di gestire i Fragment all'interno della schermata. I Panel sono stati quindi riprodotti nei Fragment, non nelle Activity.

Capire come implementare la UI è stata forse la parte più onerosa del progetto. Nel proseguo della sezione verrano presi in considerazione gli aspetti che hanno richiesto maggiore lavoro di ricerca.

3.3.1 Visualizzazione dati in forma tabellare

La maggior parte delle schermate di *WCube* mostra dei dati in forma tabellare. Da qui è nata l'esigenza di creare un modello di tabella (tabel model) che fosse comune a tutte le tabelle mostrate dall'applicazione. Ogni tabella poi implementerà tale modello al fine di personalizzarlo a seconda dei dati che deve mostrare.

Il table model su Android è stato implementato da un Fragment, chiamato `TableFragemnt`, che implementa le interfacce `TableBuilder` e `TableModel`. Mentre la prima espone i metodi per costruire la struttura della tabella, la seconda espone metodi astratti per personalizzare il contenuto.

In figura 3.1è possibile vedere l'albero delle view (widget e layout) che formano il layout di una tabella. Il layout della tabella è stato definito in modo dichiarativo su file XML; mentre il layout delle righe (numero di colonne) e dell'header (nomi delle colonne), essendo differente da tabella a tabella, è costruito in modo programmatico nelle implementazioni dei metodi dell'interfaccia `TableBuilder`.

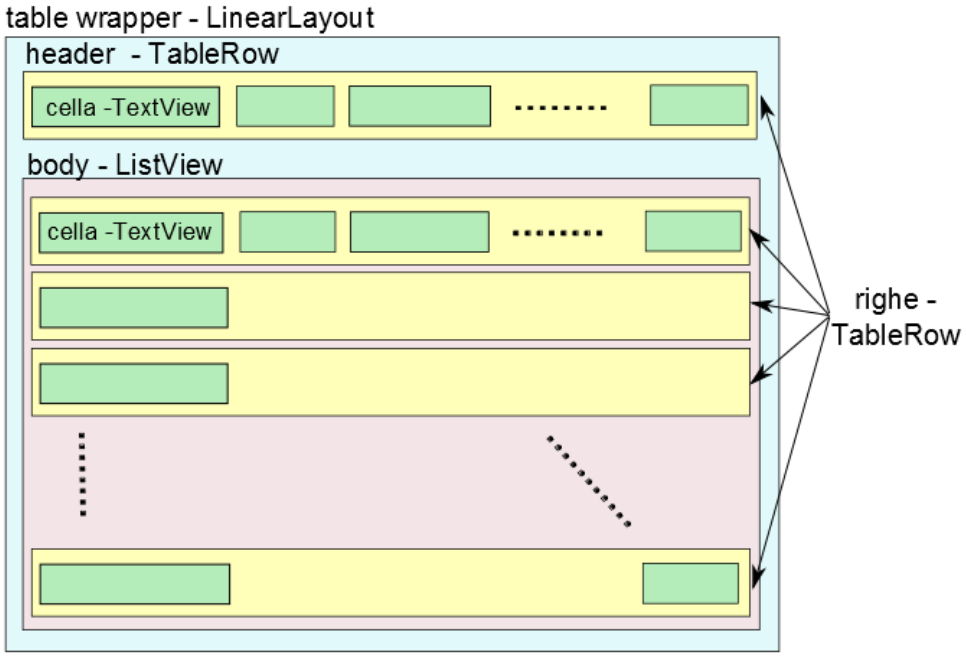

Figura 3.1: Layout e Widget che compongono una tabella

E' importante evidenziare come il body della tabella sia una ListView, i cui item sono le righe. Ho deciso di utilizzare una ListView essenzialmente per due motivi: il primo è che in questo modo la tabella è scrollabile verticalmente; la seconda ragione, più importante, è che la ListView consente di utilizzare un ListAdapter.

ListAdapter è uno strumento delle API atto a facilitare il binding tra la collezione di dati da visualizzare e la ListView. ListAdapter è una classe astratta che presenta numerose implementazioni a seconda della provenienza dei dati. Le API , in questo senso f implementazioni di ListAdapter per mappare dati contenuti su Array, ArrayList e Cursor. Nel contesto MBM la collezione dati da visualizzare è sempre un'ArrayList di oggetti "Data" visti precedentemente e quindi è stato utilizzato un ArrayAdapter generico parametrizzato dalla classe `RowShadow`.

L'implementazione di ArrayAdaper fornita dalle API prevede che il layout degli item della ListView sia una semplice TextView in cui viene stampato un oggetto dell'ArrayList. poiché tale comportamento non è quello da noi voluto è stato necessario estendere `ArrayAdaper` e fare l'override del metodo `getView()`, ovvero il metodo che implementa la funzione di mapping. La

classe risultante è stata chiamata `RowShadowAdpater`. Il metodo `getView()` è stato implementato in modo che associ gli attributi di un oggetto "Data" alle celle (TextView) di un riga (TableRow) della tabella. Per fare ciò `getView()` fa ricorso al metodo `getValue(int row_index, int column_index)` esposto dall'interfaccia `TableModel`. Uno dei vantaggi di utilizzare i ListAdapter è che essi provvedono anche al riutilizzo delle view utilizzate dagli item, con conseguente aumento delle prestazioni.

Ogni Fragment che contiene una tabella estende quindi la classe `TableFragment` e implementa quei metodi (come `getValue()`) che consentono di personalizzarne il contenuto. La classe `TableFragment` vista la sua generalità è stata inserita in *MbmDroid*.

In figura 3.2 viene schematizzato il comportamento di `RowShadowAdaper`, mentre la figura 3.3 rappresenta uno screenshot di un'Acitivity dell'applicazione *WCube* che utilizza un TableFragment.

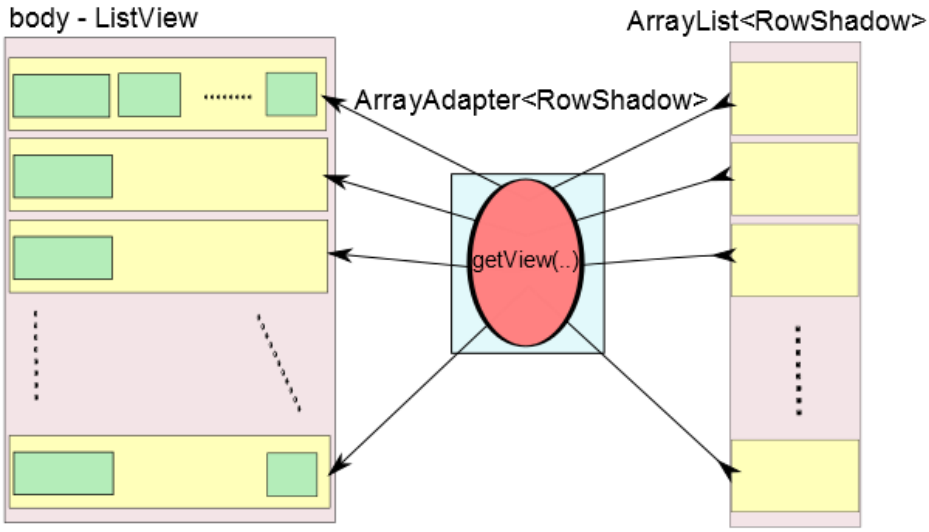

Figura 3.2: Funzionamento di un ArrayAdapter

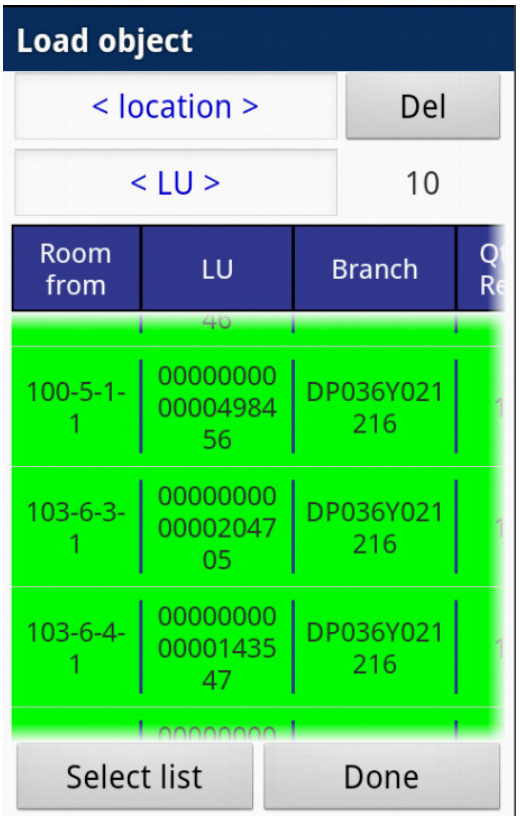

Figura 3.3: Una Activity di WCubeApp con un TableFragment.

3.4 Implementazione degli AsyncTask

Le applicazioni MBM, essendo di tipo "mission critical" prevedono che ogni modifica ai dati da parte dell'applicazione venga riprodotta sul db. Nonostante SQLite sia molto veloce il fatto di avere più thread concorrenti che accedono al db (vedremo poi quali) fa si che ogni operazione possa venire rallentata. Ogni funzione che aggiorna i dati è quindi considerata onerosa per il sistema e per questa ragione andrebbe eseguita fuori dal main thread. Se poi si considera che in certi casi è previsto che i dati vengano scaricati dal server questo non fa che aumentare la voluminosità dell'intera operazione.

Per questo motivo ogni operazione di questo tipo è stata implementata da un AsyncTask in cui nel metodo `doInbackground()` vengono eseguite le operazioni più onerose, mentre in `onPostExecute()` si aggiorna la UI. AsyncTask, come detto, garantisce che i due metodi, nonostante siano eseguiti su thread diversi, siano sincronizzati, ovvero avvengano in sequenza.

3.4.0.1 Porting delle dialog

Un altro componente della UI che si voleva rendere standard sono le dialog, in particolare le dialog modali. Nelle applicazioni MBM le dialog sono nella maggior parte dei casi modali, cioè vengono interposte nel flusso procedurale di una funzione, associata ad una determinata azione, solitamente per richiedere all'utente se procedere o meno con il completamento dell'azione[2]. Le dialog modali sono quindi bloccanti, cioè bloccano l'esecuzione del codice finché l'utente non conferma, o annulla, premendo rispettivamente il bottone "ok" e "cancel" della dialog.

In Android le dialog vengono implementate da dei Fragment (classe `DialogFragment`) e sono quindi parte della UI. Ciò fa si che non possano replicare il comportamento delle dialog modali di Java ME perché ciò comporterebbe un blocco del main thred. Per questo motivo l'evento "utente ha interagito con la dialog" viene gestito con dei listener, oppure in delle callback.

Su Android nasce quindi il problema di replicare le funzioni che prevedono delle dialog, poiché queste, dovrebbero essere smantellate e il codice riportato su più funzioni e callback. poiché ogni schermata presenta almeno una funzione di questo tipo si capisce come il porting risulterebbe molto più complesso.

Facciamo un esempio per chiarire il problema. Il listato #3.2 mostra come viene implementata una funzione onerosa che prevede una dialog, e al termine, al solito, aggiorna al UI; mentre il listato #3.3 mostra il modo più immediato per replicare il comportamento di tale funzione su Android.

Listato #3.2

```
public class APanel{
   [...]

   public void dialogLRO(){

      preDialogLRO();
      int result = new Dialog("a dialog..");
      if(result is not "OK")
         return;
      postDialogLRO();
```

[2] In alcuni casi prevedono anche di inserire un dato in input.

```
        UpdateUI();
    }
    [...]
}
```

Listato #3.3

```
public class Fragment{
    [...]

    public void dialogLRO(){
        new PreDialogAsyncTask().execute();
    }

    private class PreDialogAsyncTask extends AsyncTask<Void,
        Void, Void> {

        @Override
        protected Void doInBackground(Void... params) {
            preDialogLRO();
            return null;
        }

        @Override
        protected void onPostExecute(Void params) {
            new DialogFrgment("a dialog..").show();
        }
    }

    //callback invocata quando l'utente ha interagito con al
        dialog
    public void onDialogResult(int result){
        if (result is "OK")
            new PostDialogAsyncTask().execute(key);
    }

    private class PostDialogAsyncTask extends AsyncTask<Void,
         Void, Void> {

        @Override
        protected Void doInBackground(Void... params) {
            postDialogLRO();
```

```
            return null;
        }

        @Override
        protected void onPostExecute(Void params) {
            updateUI();
        }
    }
}
```

Come è possibile notare il codice risulta molto più arzigogolato e sono neccessari due AsyncTask, più la callback che gestisce l'iterazione dell'utente con la dialog. L'obiettivo era utilizzare un solo AsyncTask ed evitare di implementare ogni volta la callback.

Per utilizzare un unico AsyncTask è necessario lanciare la dialog nel metodo `doInBackground()`. Nella sezione 2.12.1 abbiamo visto come ci siano vari modi per manipolare la UI fuori dal main thread. In questo caso è stato utilizzato un Handler. Per ottenere la sincronizzazione, ovvero bloccare l'esecuzione di `doInBackground()` per riprenderla una volta che l'utente ha interagito con la dialog (`onDialogResult()` è stato invocato), è stata utilizzata la concorrenza, in modo particolare i metodi `wait()` e `notify()`. Il meccanismo di sincronizzazione è stato integrato in due nuovi componenti, `DialogAsyncTask` e `ConfirmDialogFragment`, che estendono rispettivamente le classi `AsyncTask` e `DialogFragment`. Nel listato #3.4 vediamo come utilizzare questi nuovi componenti per replicare il comportamento dei listati #3.2 e #3.3.

Listato #3.4

```
public class MyActivity extends Activity {
    [ . . . ]

    protected Handler UIHandler
     @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        UIhandler = new handler();
            [ . . . ]
    }
```

```
 protected void dialogLRO() {
     // lancia un AsyncTask che prevede una dialog.
   new ADialogAsyncTask().execute();
}

private class ADialogAsyncTask extends DialogAsyncTask<
    Void, Void, Void> {

   private Object result;

   @Override
   protected Void doInBackground(Void... params) {

      preDialogLRO();

      // mostra la dialog che permette all'utente di
          confermare l'operazione
      UIhandler.post(new Runnable(){
         public void run(){
            //AConfirmDialogFragment estende
                ConfirmDialogFragment
            AConfirmDialogFragment.newInstance(this, "a
                dialog..").showDialog();
         }
      }
      wait();

      // controlla se l'utente ha confermato o meno.
      if (result is not "OK")
         return;

      //ha confermato
      postDialogLRO();

   @Override
   protected void onPostExecute(Void params) {
      updateUI();
   }

   //questo è un metodo di DialogAsyncTask
   // è stato riportato qui per comodità, ma non c'è l'
```

```
        @override
        public void setDialogResult(Object result){
            this.result = result;
        }
    }
}

//riporto parte della classe ConfirmDialogFragment
public class ConfirmDialogFragment extends DialogFragment{
    AsyncTask task;
    public Dialog newInstance(AsyncTask task, String title){
        this.task = task;
        return new DialogFragment(title);
    }

    [...]

    // callback invocata quando la dialog è dismessa(l'utente
       ha premuto ok o cancel)
    public void onDismiss(Object result){
        task.setDialogResult(result)
        task.notify();
    }
}
```

3.5 Configurazioni e Preferenze

Le applicazioni MBM prevedono di utilizzare un file di configurazione di default per configurare l'applicazione al primo avvio. Il file di configurazione è più precisamente un file di properties, ovvero un file contenente una lista di attributi chiave-valore dove ogni coppia rappresenta una singola proprietà di configurazione. poiché queste proprietà devono essere accessibili dall'applicazione in qualsiasi momento in modo rapido è necessario mappare queste proprietà in oggetti globali. Su Android la via migliore per fare ciò è utilizzare le SharedPreference, presentate nel capitolo 2. I vantaggi che si ottengono nel utilizzare questo approccio sono molti, in particolare le SharedPreference:

- ricalcano la struttura chiave-valore delle properties;
- sono salvate in modo persistente dal SO;

- l'accesso in modalità lettura/scrittura è semplice e veloce;
- si integrano in modo automatico con le Preference e con le PreferenceActivity.

Il primo punto si traduce nel fatto che la mappatura con le properties risulta immediata. La persistenza fa invece si che tale mappatura sia sufficiente eseguirla una sola volta, al primo avvio dell'applicazione.

Ma è l'ultimo punto a giustificare l'implementazione delle proprietà in questo modo. Per capirne il motivo è necessario dire che le proprietà di configurazione sono di due tipi: definitive e configurabili. Nel primo caso si tratta di proprietà (SharedPreference) modificabili solo dal codice, mentre le proprietà configurabili sono quelle modificabili da parte dell'utente via UI. Le proprietà configurabili sono quindi in definitiva le preferenze dell'applicazione.

Le API di Android in questo senso mettono a disposizione una Activity, denominata PreferenceActivity, che facilita le realizzazione di un menù per la gestione delle preferenze. Questa è Per implementare una PreferenceActivty è sufficiente estendere tale classe e fare l'override del metodo `onCreate()` in cui viene associato il menu, definito, al solito, o in modo programmatico, o in un file XML.

Gli item del menu corrispondono ad implementazioni della classe astratta `Preference`. Esempi di implementazioni presenti nelle API sono `CheckBoxPreference`, `ListPreference`, `EditTextPreference`. Come si può apprezzare dall'immagine sottostante queste implementazioni forniscono la UI agli item a cui vengono associate. La preferenza "Debug", ad esempio, è implementata da una `CheckBoxPreference`, "Livello di Debug" da una `ListPreference`, mentre "Server IP" da una `EditTextPreference`. Nel caso di `ListPreference` e `EditTextPreference` una volta cliccato l'item viene aperta automaticamente una dialog per cambiare la preferenza.

La prima immagine (figura 3.4) si riferisce alla parziale realizzazione della schermata "Opzioni" (`SettingsActivity`) dell'applicazione *WCubeApp*. Il listato #3.5 invece mostra il contenuto del file XML che definisce il menu per `SettingsActivity`.

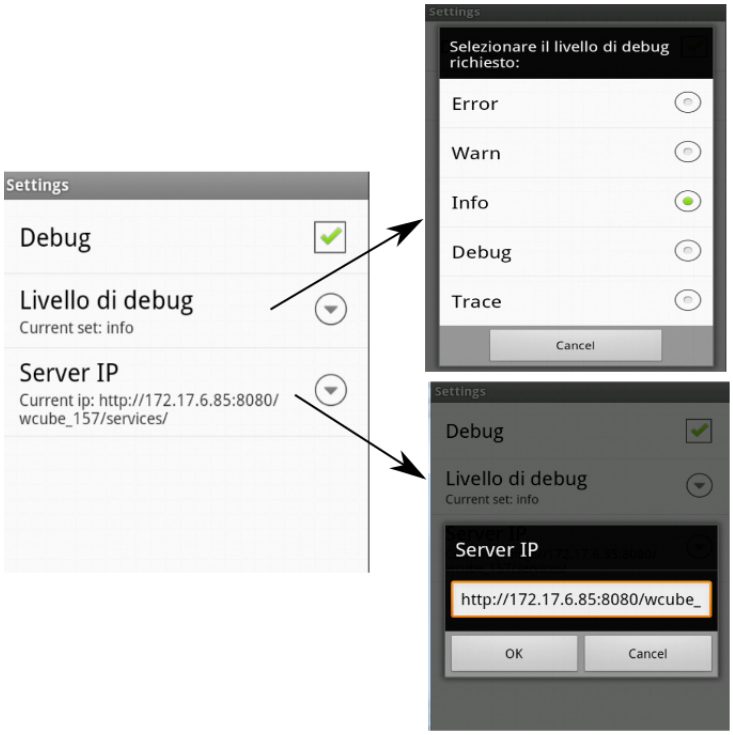

Figura 3.4: Schermate relative alle opzioni di *WCubeApp*

Listato #3.5

```
<?XML version="1.0" encoding="utf-8"?>
<PreferenceScreen XMLns:android="http://schemas.android.com/
    apk/res/android">
    <CheckBoxPreference
        android:key="DEBUG"
        android:title="@string/pref_title_debug"/>
    <ListPreference
        android:dependency="DEBUG"
        android:key="DEBUG_LEVEL"
        android:title="@string/pref_title_debug_level"
        android:dialogTitle="@string/
            pref_dialog_title_debug_level"
        android:entries="@array/debug_entries"
        android:entryValues="@array/debug_values"/>
    <EditTextPreference
        android:key="BASE_WEB_URL"
        android:title="@string/pref_title_server_ip"
```

```
        android:inputType="textUri" />
</PreferenceScreen>
```

Il binding tra SharedPreference e gli oggetti Preference è garantito dalla chiave e la relazione è ovviamente bidirezionale, ciò implica che la modifica ad una Preference apportata dall'utente in una PreferenceActivity è immediatamente riscontrabile nel valore della SharedPreference associata, e viceversa.

affinché l'utente possa ripristinare le impostazioni di default è stata prevista una opportuna opzione che non fa altro che rimappare il file di properties nelle SharedPreference.

3.6 Comunicazione con il server

Per quel che riguarda il modulo di comunicazione il lavoro da fare è stato minimo poiché la quasi totalità delle classi utilizzate da tale modulo sono presenti anche nelle API di Android. In particolare stiamo parlando delle classi appartenenti ai package `java.net.*` (gestione delle connessioni) e `javax.xml.*` (creazione e il parsing degli EcoMessage). A parte piccole modifiche il modulo è stato riportato tale e quale sul framework.

Nel futuro questo modulo potrebbe essere riprogettato al fine di utilizzare JSON come formato per i messaggi. JSON infatti risulta essere più semplice, leggero ed immediato rispetto all'XML.

3.7 Gestione dei background thread

Come accennato nella presentazione di *WCube* le applicazioni MBM utilizzano delle routine di servizio eseguite in background per comunicare determinate informazioni al server.

Uno dei servizi di cui era richiesta l'implementazione era quello che si occupava di comunicare lo stato delle LUW al server. L'iterazione di questo servizio con la user interface è minimo e avviene solo se entrambe le seguenti condizioni sono verificate: il server ha processato correttamente le LUW e l'applicazione sta visualizzando in foreground determinate Activity. Il LUW service, nel framework *MbmPalm*, è stato implementato da un thread che cicla all'infinito ripetendo le seguente operazioni:

1. verifica lo stato delle LUW nel db;
2. se trova LUW da spedire le invia al server in un ECO Message e attende una risposta;
3. processa la risposta ed eventualmente aggiorna la UI.

Per fare si che il thread non cicli inutilmente è previsto che questo si sospenda se non ci sono LUW da inviare, oppure se determinate condizioni sul loro stato non sono soddisfatte; e si risvegli, o al momento della creazione di una nuova LUW, oppure dopo un determinato intervallo di tempo.

Su Android il comportamento del Luw Thread è stato realizzato con un IntentService: ovvero un Service che gestisce richieste, sotto forma di Intent, in modo asincrono. Un IntentService ha la caratteristica di processare le richieste nell'ordine in cui arrivano nella callback `onHandleIntent(Intent)`. E' stato scelto di utilizzare un IntentService per il semplice motivo che le richieste vengono esaudite in un worker thread.

Per fare si che il Service venga avviato a intervalli regolari è stato utilizzato AlarmManager, ovvero un servizio di Android che consente di accedere ai servizi di allarme di sistema rendendo possibile la schedulazione di attività nel futuro, nel nostro caso l'avvio del Luw Service.

Per fare si che il Luw Service aggiorni la UI di determinate Activity solo se queste sono in frontend è stato utilizzato un BroadcastReceiver con un IntentFilter predefinito. Tale componente verrà registrato a runtime dalle Activity in `onResume()` e deregistrato in `onPause()`: in questo modo si garantisce che egli riceva Intent solo se l'Activity è in frontend. Affinche il Luw Service aggiorni la UI è quindi sufficiente che invii un Intent in braodcast avente action uguale a quella specificata nell'IntentFilter, sarà poi il Broadcast Receiver di ogni Activity ad occuparsi di aggiornare la UI nel modo appropriato.

Il Luw Service è un servizio presente in ogni applicazione MBM ed è quindi stato inserito nel framework *MbmDroid*.

3.8 Gestione dei LOG

Un'applicazione enterprise necessita di un sistema di logging che sia automatizzato e abbia le seguenti funzionalità:

- supporti diversi livelli di debug;
- condizioni l'output del debug a seconda del livello specificato dai componenti che concorrono a produrre messaggi;
- formatti il log secondo un determinato pattern;
- consenta di reindirizzare il log su diversi output, in modo particolare su file. In questo caso i file devono essere organizzati secondo una logica configurabile.

Fortunatamente esistono già framework che svolgono le funzioni sopracitate; quindi non c'è la necessità di sviluppare un proprio sistema di logging. Dopo una lavoro di ricerca per individuare il framework che meglio venisse incontro alle nostre esigenze, e che fosse compatibile con Android, la scelta è ricaduta su Logback - versione Android.

3.8.1 Logback

Logback è uno dei tanti framework per il logging in ambiente Java. Logback può essere considerato come l'evoluzione di log4j, probabilmente il più popolare di questi framework. Come vedremo però rispetto al predecessore risulta essere più efficiente.

L'architettura di Logback è sufficientemente generica per essere applicata in differenti contesti, come dimostra il fatto di essere compatibile per Android. Logback è organizzato in 3 moduli separati: logback-core, logback-classic and logback-access. logback-core in particolare svolge il ruolo di piattaforma per gli altri due moduli. logback-classic invece può essere visto come un significativo miglioramento di log4j. Inoltre logback-classic implementa le SLF4J API, cioè una libreria che funge da interfaccia comune a molti framework di logging per Java, come lo stesso log4j, oppure la libreria `java.util.logging`. A differenza di questi ultimi Logback ha però il vantaggio di implementare nativamente le API SLF4J; non c'è quindi bisogno di inserire un bridge per il binding tra le due entità. Questo è anche il motivo per cui Logback ha prestazioni migliori rispetto agli altri framework. Nella figura 3.5si può vedere il raffronto tra l'architettura log4j-SLF4J e quella Logback - SLF4J utilizzata nel mio progetto. Il terzo modulo, logback-access, serve infine per integrare Logback con i Servlet container per fornire funzionalità di log remoto via HTTP.

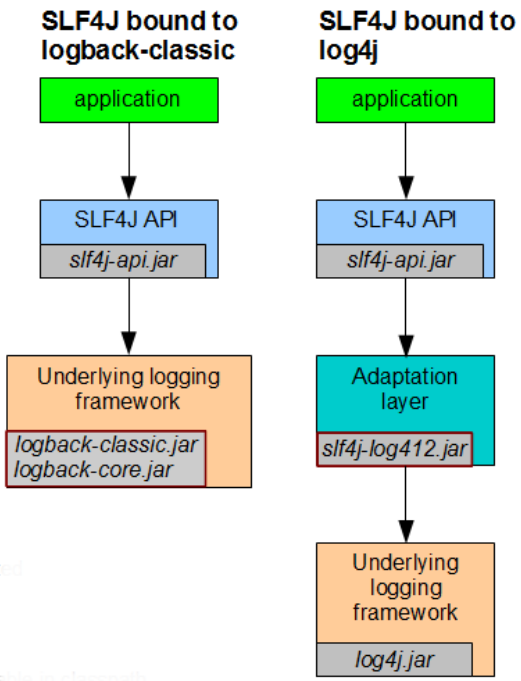

Figura 3.5: Confronto tra l'integrazione con SLF4J tra il framework Logback e log4j

3.8.1.1 Architettura

Il framework è costruito su tre elementi principali (Logger, Appender e Layout) e supporta cinque livelli di debug: ERROR, WARN, INFO, DEBUG e TRACE. I livelli è necessario siano ordinati tra di loro. In SLF4J abbiamo che TRACE < DEBUG < INFO < WARN < ERROR.

Un Logger è uno strumento che permette di inviare messaggi, cioè eseguire una request di log, verso una determinata destinazione (Appender) secondo un determinato formato (Layout). I logger sono strutturati gerarchicamente in modo analogo ai package in Java, dai quali prendono spunto per la organizzazione dei nomi. Ad esempio un Logger di nome[3] `it.tesi` assume il ruolo di padre del Logger con nome `it.tesi.unipd`. L'organizzazione gerarchica dei Logger è la base su cui è stato progettato il framework permettendone flessibilità e maneggevolezza. Al vertice della gerarchia dei Logger si trova il nodo radice che prende il nome di RootLogger. Un logger può eseguire una request per ognuno dei cinque livelli di debug possibili, ma

[3] generalmente ci si riferisce al nome di un logger con il termine TAG

solamente le request di livello maggiore o uguale al livello impostato per quel Logger (effective level) saranno prese in considerazione. Se la request viene presa in considerazione dal sistema si dice che essa è abilitata. La regola può essere riassunta dalla tabellina sottostante. Se ad un logger non è associato alcun livello di debug viene ereditato di default quello (non nullo) dell'antenato più prossimo.

level of request p	effective level q					
	TRACE	**DEBUG**	**INFO**	**WARN**	**ERROR**	**OFF**
TRACE	YES	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO
ERROR	YES	YES	YES	YES	YES	NO

Figura 3.6:

Ora che è stato visto come condizionare il risultato dell'output a seconda del livello di debug occupiamoci della sua destinazione. Ad un Logger può essere registrato più di un Appender, cioè più di una destinazione. Ciò implica che ogni request abilitata eseguita su un particolare Logger, verrà inviata a tutti gli Appender che si sono registrati presso quel Logger. Ma c'è di più: l'organizzazione gerarchica dei Logger fa si che la request venga propagata anche verso tutti gli Appender registrati sui Logger antenati[4]

Il framework mette a disposizione alcuni Appender predefiniti come la console, i file di testo, mail, ecc. Gli Appender possono essere ulteriormente configurabili, ad esempio specificando il Layout, ovvero il format dell'output.

Introdotti i concetti base di Logback occupiamoci della sua configurazione.

3.8.1.2 Configurazione

Logback può essere configurato in due modi: programmaticamente o via file XML. La seconda opzione risulta ovviamente la migliore poiché non c'è bisogno di mettere mano al codice, ed è quella che è stata utilizzata nel progetto.

[4]Questo comportamento, detto Appender Additivity, può essere evitato settando a false l'additivity flag.

Su Android la configurazione via XML può essere definita nei seguenti modi:

- tra i tag <logback></logback> nel Manifest file;
- nel file `logback.xml` collocato o nella cartella `/asset` del progetto, oppure nella cartella `/sdcard/logback` del dispositivo.

La versione di Logback per Android (detta logback-android) ricercherà automaticamente se c'è una configurazione valida; in caso contrario sarà caricata una configurazione base che consente di utilizzare il framework solo nella modalità di default.

La struttura del file XML deve avere lo schema sottostante:

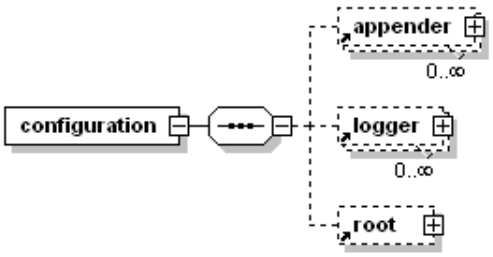

Figura 3.7: UML schema del file di configurazione di Logback

Non approfondiamo ulteriormente la trattazione del file di configurazione in questa sezione poiché nella prossima vedremo come è stato implementato nel progetto.

3.8.2 Implementazione

Nel mio progetto il sistema di logging doveva prevedere che l'output di destinazione fossero file, in particolare si richiedeva che fosse creato un report giornaliero. Per fare in modo che Logback cambiasse il file di destinazione al cambio di data è stato utilizzato un appender speciale chiamato RollingFileAppender. Questo Appender si comporta in base a delle policy atte a definire le condizioni di rollover. Logback fornisce policy già preconfezionate che lo sviluppatore deve quindi solo configurare (sempre da XML) e associare al RollingFileAppender. Una di queste policy, TimeBasedRollingPolicy, fa

esattemente al caso nostro in quanto definisce una policy di rollover bastata sul tempo.

Per quel che riguarda l'albero dei logger è stato deciso che ogni classe del progetto implementasse la sua istanza di Logger, il TAG corrisponde quindi al nome esteso della classe. Questo è un pattern abbastanza comune e quindi è stato adottato tale e quale. In questo modo si uniforma anche l'albero dei logger, poiché questo corrisponde a quello dei package del progetto.

Il logger di ogni classe è quindi definito come variabile statica e istanziato dalla factory fornita dalle librerie di Logback.

```
private static Logger logger = LoggerFactory.getLogger(
    MyClass.class);
```

Ai logger non è stato assegnato alcun Appender ed effective level poiché si è deciso di sfruttare la struttura gerarchia del framework e far ereditare queste configurazioni, che sarebbero state comuni, dal nodo root, configurato nel file XML.

Infine era richiesto che fosse possibile impostare il livello di debug a runtime dalle preferenze dell'applicazione. Ciò è stato possibile grazie alla possibilità di configurare Logback anche programmaticamente. Per come è stato progettato l'albero dei logger è infatti sufficiente cambiare il livello di debug associato al nodo root.

3.9 Barcode Scanner

Le applicazioni *WCube* prevedono l'utilizzo di uno scanner per i barcode come periferica di input aggiuntiva. Era quindi richiesto di realizzare un modulo che fornisse gli strumenti per gestire lo scanner ed eseguire le operazione di lettura. Tale modulo doveva essere progettato affinché fosse più indipendente possibile dal driver utilizzato per lo scanner.

poiché le API di Android non hanno il controllo sullo scanner dei dispositivi bisogna necessariamente utilizzare librerie native fornite dal costruttore. Il binding tra codice Java e le librerie native è garantito da JNI (Java Native Interface). JNI è un framework del linguaggio Java che consente al codice Java di richiamare (o essere richiamato da) codice cosiddetto "nativo", ovvero specifico di un determinato sistema operativo o, più in generale, scritto

in altri linguaggi di programmazione, in particolare C e C++. L'interfacciamento è basato sulla definizione di un insieme di classi di raccordo fra i due contesti, che presentano un'interfaccia Java, ma che delegano al codice nativo l'implementazione dei loro metodi.

Come accennato nella presentazione di Android Google mette a disposizione un SDK specifico, chiamato NDK, per facilitare lo sviluppo (building, debug, ecc) di applicazioni che prevedono di utilizzare il framework JNI. La figura 3.8 sottostante schematizza quanto detto.

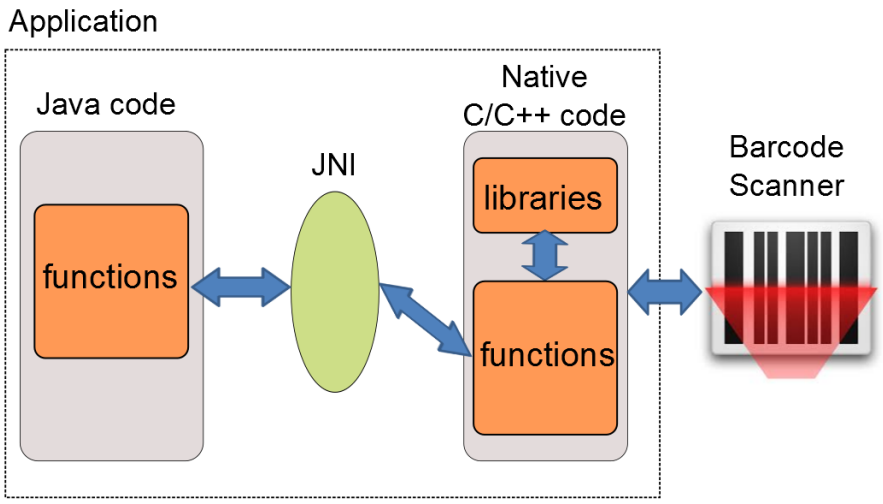

Figura 3.8: Architettura di un'applicazione che utilizza codice nativo

Fortunatamente il costruttore del dispositivo sul quale verrà installato *WCube* ci ha fornito un'applicazione di esempio con il relativo codice sorgente. Nel progetto sorgente, oltre al codice nativo, sono contenute delle classi Java che fungono da API per lo scanner e che al loro interno inglobano le funzioni di raccordo previste da JNI.

Una volta importate le API e compilato il codice nativo utilizzando i tool di NDK non è rimasto che implementare il modulo che utilizzasse le suddette API al fine di fornire all'applicazione uno strumento per utilizzare lo scanner nel modo corretto.

`WCube` prevede che possano essere letti barcode con due simbologie diverse[5], i quali poi si differenziano in ulteriori sottotipi a cui vengono asso-

[5] Code39 e Code128

ciati diverse entità nella logica dell'applicazione. Ciò che viene letto dallo scanner deve quindi subire un processo di raffinamento affinché possa essere identificata l'associazione corretta.

Le operazioni di lettura da parte dello scanner e la successiva elaborazione del barcode letto, per il solito motivo, devono essere necessariamente eseguite al di fuori del main thread. Tenendo conto che il processing del barcode non implica che la stringa in input provenga dallo scanner (l'utente può inserire il codice utilizzando la tastiera), è stato deciso di implementare lettura e processing su thread differenti. I thread sono stati realizzati estendendo la classe `HandlerThread`, che come detto implementa un Looper e relativa MessageQueue di default.

Non entro nel dettaglio di come sono stati implementati i thread poiché la logica è molto complessa e a parole impossibile da spiegare. Accenno solo che non in tutte le Activity è prevista la lettura dei barcode e per questo motivo solamente in quelle che lo prevedono sono stati istanziati i thread per la lettura e processamento dei barcode. Il ciclo di vita degli HandlerThread ricalca quindi quello delle Activity che li hanno istanziati. In questo modo si rimane fedeli alla filosofia di Android di limitare lo spreco di risorse. Per gestire eventi rilevanti (es. "barcode letto") provenienti dal thread che gestisce lo scanner i Fragment utilizzano un Listener (Observer Pattern).

Il modulo contente le classi che implementano i thread di cui abbiamo parlato, le API Java per lo scanner, e tutte le classi di supporto per processare i barcode sono state inserite in *MbmDroid*.

3.10 La classe Application

Nel corso dello sviluppo era nata l'esigenza di avere un componente che mantenesse lo stato globale dell'applicazione. Il modo più semplice per realizzare ciò è estendere la classe `Application` e specificarne il nome completo nell'elemento `<application>` del file `AndroidManifest.xml`. Così facendo il sistema operativo creerà un'istanza di tale classe all'avvio dell'applicazione. La classe `Application` implementa inoltre una callback (`onCreate()`) che viene invocata dal sistema operativo prima che ogni altro componente venga istanziato. In tale metodo è quindi possibile eseguire tutte le operazioni di

inizializzazione. Una di queste operazioni, ad esempio, eseguita solo al primo avvio, è il caricamento del file di configurazione nelle SharedPreference.

Nella nostra implementazione di `Application`, rinominata `WcubeApp`, verranno salvate inoltre variabili e oggetti globali. Per non creare un attributo per ogni oggetto è stato deciso di creare una `HashMap<String, Object>` in cui l'acceso in lettura/scrittura sugli oggetti avviene tramite chiave di tipo Stringa.

Capitolo 4

Conclusioni

L'esperienza di stage in azienda è stata dal mio punto di vista molto positiva poiché mi ha permesso di fare un po' di esperienza nel mondo IT. Aver affrontato una tesi su un argomento come Android credo mi abbia permesso di acquisire un bagaglio di conoscenze molto richieste al giorno d'oggi. Nel 2011[1] infatti, per la prima volta nella storia, nel mondo sono stati venduti più smartphone che pc, e la tendenza è in costante aumento. Lo sviluppo di applicativi mobile è quindi un tema molto attuale e con grandi prospettive per il futuro.

In questo contesto Android si pone come piattaforma di riferimento, in quanto è il sistema operativo mobile più diffuso al mondo, detenendo circa due terzi del mercato. I motivi del successo di Android sono molteplici: uno di questi è sicuramente il fatto di essere *open*, garantendo quindi strumenti di sviluppo non a pagamento e la possibilità di pubblicare applicazioni liberamente nel Android Market al fine di incentivarne lo sviluppo. L'utilizzo di un linguaggio già ampiamente conosciuto e collaudato come Java è senza dubbio un'altra chiave del suo successo.

Rispetto a Java ME Android introduce un notevole salto di qualità, soprattutto per quel che riguarda la user experience delle applicazioni e la facilità di accesso alle funzionalità hardware dei device. .Per quel che riguarda la programmazione il fatto di poter usare un approccio dichiarativo che prevede file XML fa si che ci sia una migliore separazione tra presentazione e logica, il che è indubbiamente un vantaggio.

[1] http://www.pcmag.com/article2/0,2817,2399846,00.asp

Per quel che riguarda gli obiettivi dello stage essi sono stati raggiunti. All'azienda ritengo di avere fornito tutte le indicazioni e gli strumenti per poter iniziare ad eseguire il porting completo delle loro applicazioni mobile su Android. Le funzionalità principali del framework *MbmPalm*, al quale fanno riferimento tutte le applicazioni mobile MBM, sono state reimplementate in *MbmDroid*. Per fornire una applicazione prototipo che fungesse da linea guida ho poi sviluppato *WCubeApp*, il cui scopo, soprattutto nella fase iniziale, era quello testare le soluzioni adottate e verificare la fattibilità del porting sia in termini di tempo che di prestazioni. *WCubeApp* rappresenta quindi il porting di una parte di *WCubePalm*, il cui completamento può essere realizzato a partire da ciò che è già stato fatto.

Durante la realizzazione del progetto sono state incontrate difficoltà di diverso tipo. In primo luogo non è stato facile capire la logica che sta dietro ai progetti a cui dovevo fare riferimento: *MbmPalm* e *WCubePalm*. Fortunatamente in azienda c'è sempre stata la disponibilità da parte di tutti di aiutarmi nei momenti in cui incontravo difficoltà in questo senso.

Altre difficoltà sono nate nella fase di ricerca in cui bisognava scegliere come realizzare le funzionalità richieste. Essendoci la possibilità di fare le cose in vari modi si è sempre dovuto provare più soluzioni, per poi scegliere quella che meglio interpretasse quel tipo di funzionalità. Per migliorare le performance è stato poi fatto un ampio uso del multithreading, che, come si può intuire, non facilita di certo il porting, però è assolutamente necessario se si vuole realizzare un'applicazione reattiva, prerogativa per un integrazione ottimale nel sistema Android.

Bibliografia e sitografia generale

[1] Reto Meier, Professional Androdi 4 Application Development, Wrox

[2] Massimo Carli, Guida per lo Sviluppatore, Apogeo

[3] James Steele, Jenson To, The Android Developer's Cookbook: Building Applications with the Android SDK

Documentazione online:

[1] Android Developer Official Site,
http://developer.android.com/index.html

[2] Android Developer Blog
http://android-developers.blogspot.it/

Ringraziamenti

Desidero ringraziare tutti coloro che mi hanno supportato in questi anni e reso possibile il raggiungimento di questo importante traguardo.

Voglio quindi innanzitutto ringraziare i miei genitori per avermi sempre sostenuto in questo percorso di studi, sia materialmente, ma soprattutto moralmente. Li ringrazio inoltre per non aver mai interferito nelle mie scelte e avermi lasciato totale libertà in ogni momento; vi ringrazio quindi della fiducia che avete riposto in me, spero di averla ripagata.

Voglio poi ringraziare il mio relatore, Prof. Sergio Congiu, e l'azienda MBM Italia S.r.l. per avermi dato l'opportunità di svolgere la tesi in azienda, in quella che considero un'esperienza davvero molto positiva. Vorrei quindi ringraziare in modo particolare chi mi ha aiutato a renderla tale, ovvero chi mi ha seguito in azienda in questi 6 mesi : Ing. Alessandro Costacurta e Ing. Davide Baracco.

Infine vorrei ringraziare tutti gli amici per aver reso questi anni di vita universitaria assolutamente da ricordare.