ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

SVILUPPO DI APPLICAZIONI MOBILE SU PIATTAFORMA ANDROID: ANALISI DEL MODELLO DI PROGRAMMAZIONE E SPERIMENTAZIONI

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
MATTIA BALDUCCI

Relatore:
Prof. ALESSANDRO RICCI

Co-relatori:
Ing. ANDREA SANTI

ANNO ACCADEMICO 2011–2012
SESSIONE II

PAROLE CHIAVE

Android

Applicazione

Eventi

SMS Backup+

Alla mia famiglia...

A mio nonno Luciano...

Indice

Capitolo 1

Introduzione

Dal 3 aprile 1973, data in cui Martin Cooper (direttore della sezione "ricerca e sviluppo"di Motorola) fece la prima storica telefonata da un telefono cellulare, sono passati quasi 40 anni. Fortunatamente, dal quel primo modello sperimentale, più simile ad una piccola valigia che all'attuale concetto di telefonino, sono stati fatti notevoli progressi, sia in termini di funzionalità che di dimensioni. Questa rivoluzione può esser considerata come la naturale risposta del mondo tecnologico ad uno dei bisogni principali della società moderna: la possibilità di comunicare. Come ogni grande cambiamento, anche questa svolta nel mondo della *comunicazione* è avvenuta perché vi era alla base un intenso desiderio da soddisfare, ovvero portare ad un livello superiore le capacità comunicative di ogni individuo, consentendogli di mettersi in contatto con altre persone in modo più dinamico e di scambiare informazioni con metodologie più efficaci della semplice comunicazione verbale.

La richiesta di una comunicazione più agile, quindi non più legata alla presenza di un telefono fisso, trovò risposta grazie all'incessante progresso tecnologico nel campo dell'elettronica. La disponibilità di nuovi circuiti integrati, sempre più piccoli ed economici, permise alle aziende del settore di lanciare sul mercato modelli di cellulari aventi un costo alla portata di molti, e con delle dimensioni tali da consentirne un trasporto ed un utilizzo agevole. Queste caretteristiche furono la chiave del successo della telefonia mobile, che vide un'esplosione di vendite tra la metà degli anni novanta e i primi anni del nuovo millennio (si stima che nel 2007, in paesi come Italia e Gran Bretagna, vi fossero già più telefonini che abitanti, e che il 50% della

popolazione mondiale avesse un cellulare [3].

Il problema dell'efficacia comunicativa, che richiedeva maggiori risorse dal punto di vista computazionale ma anche di una maggiore complessità della rete di comunicazione, venne affrontato adottando il computer come strumento principale di elaborazione. La ricerca di una soluzione portò alla creazione di Internet, che permise, alla comunità scientifica internazionale prima e al resto delle persone poi, lo scambio di informazioni in forme e modalità completamente nuove.

Ad oggi si può affermare che la rivoluzione sino ad ora descritta sia tutt'altro che conclusa, e che al contrario stia attraversando un importante periodo di transizione, nel quale il mondo della comunicazione mobile e quello di Internet convergono verso una nuova realtà tecnologica, in cui gli *smartphone* rappresentano la concretizzazione dei concetti che ne stanno alla base.

1.1 Una nuova generazione di devices

Da anni ormai si tende a non riferirsi più al proprio dispositivo attraverso le parole "cellulare"o "telefonino", sostituite dalla più moderna "smartphone". Questo cambiamento di terminologia non è solo il risultato di una brillante strategia di *marketing*, ma nasconde in sé un vero e proprio cambio generazionale, rispetto ai dispositivi precedenti.

La prima e fondamentale differenza riguarda le sorprendenti caratteristiche hardware degli smartphone di ultima generazione, peraltro in costante sviluppo, come processori *multi-core* con elevate capacità computazionali, memoria *RAM* con capacità superiori ad 1GB, e memorie per lo *storage* talmente ampie da essere capaci di contenere anche un centinaio di film. Alla crescenti capacità hardware si affianca l'evoluzione delle reti di telecomunicazione, capaci di fornire all'utente una connessione internet sufficientemente rapida non solo da garantire un'agile navigazione sul web, ma anche in grado di sostenere lo *streaming* di video e comunicazioni *VoIP (Voice over IP)*.

Altri strumenti, precedentemente considerati accessori, sono diventati indispensabili, e la loro presenza nei dispositivi viene data ormai per scontata; rientrano in questa categoria le fotocamere integrate ed il localizzatore GPS: le prime hanno raggiunto una qualità tale da sostituire le cosiddette

fotocamere *compact*, utilizzate in ambiti non professionali, l'altro ha invece permesso di dotare ogni persona di un piccolo navigatore satellitare sempre a portata di mano. L'avvento della tecnologia *touchscreen*, ed in seguito di quella *multi-touch*, hanno rivoluzionato il modo di interfacciarsi con l'utente, garantendo un'interazione più naturale ed espressiva, aprendo la strada ad infinite nuove possibilità.

Quelle elencate fino ad ora sono tutte tecnologie che negli anni hanno subito notevoli miglioramenti, ma che sostanzialmente eran già presenti anche su dispositivi non considerati *smart*. La vera novità è stata portata dall'introduzione di una moltitudine di sensori all'interno di questi dispositivi, come ad esempio accelerometri a bassa potenza, giroscopi, magnetometri ed altri ancora come sensori di luce, di temperatura, di pressione ed addirittura di umidità.

Riassumendo, lo sviluppo degli smartphone è stato sospinto fortemente dall'integrazione di tutte le tecnologie indicate all'interno di un unico dispositivo, pensato per esser portato ovunque andiamo [5], e che vanta tra le sue caratteristiche principali quella di essere un oggetto strettamente legato al proprio possessore, e di rappresentare una sorta di estensione delle capacità, per ognuno di noi, di rapportarsi con il mondo. Tuttavia, è necessario chiarire che i prodotti derivanti dai progressi della microelettronica hanno offerto solo la possibilità di creare qualcosa di innovativo: ciò che rende questi dispositivi veramente interessanti ed utili, sono l'infinità di applicazioni sviluppate per essi e rese disponibili agli utenti.

1.2 Nuove sfide applicative

Le applicazioni, o più comunemente *apps*, costituiscono il cuore pulsante di ogni smartphone; senza di esse, questi dispositivi, che noi tutti conosciamo e apprezziamo per le loro innumerevoli funzionalità, sarebbero solamente dei costosi cellulari, rivisitati nello stile e con un enorme potenziale non sfruttato.

Il concetto di applicazione si è per anni limitato a quello di un programma sviluppato sulla falsariga di quelli creati per soluzioni desktop, in grado cioè di sfruttare solamente le capacità computazionali del dispositivo, introducendo come unico elemento di novità una differente modalità di interazione, dovuta all'utilizzo della tecnologia touchscreen in sostituzione dei

classici mouse e tastiera. Di fatto questa tipologia di applicazioni rendeva gli smartphone dei piccoli computer tascabili, ma nulla più; le vere risorse da sfruttare sono invece i sensori che, combinati alla estrema portabilità di uno smartphone, aprono la strada ad una nuova concezione di applicazione, capace di interagire con l'utente non solo tramite i metodi classici ma introducendone di nuovi, come ad esempio inclinando o scuotendo il dispositivo.

Ancora più importante delle nuove forme di interazione con l'utente è la possibilità, per una applicazione, di ottenere informazioni riguardanti l'ambiente reale in cui sta operando. Questa capacità è alla base di una nuova ed interessante sfida per il mondo dell'informatica, ovvero la creazione di programmi *context-aware*. Nell'articolo pubblicato nel 2009 "Get Smart"[2] vengono identificati quattro livelli di *system awareness*:

- *Basic context awareness*: questo livello indica una consapevolezza basilare derivante da informazioni quali la posizione fisica della persona, l'ora e altri dettagli dell'ambiente circostante, ottenibili tramite i sensori precedentemente elencati;

- *Behaviour awareness*: applicazioni di questo genere sono in grado di comprendendere e reagire a seconda che l'utente stia camminando, scrivendo o stia effettuando altre tipologie di azioni;

- *Activity awareness*: rappresenta lo stadio in cui si ha consapevolezza dei fini delle azioni svolte, ad esempio sapere che si sta camminando perché si sta andando a pranzo;

- *Intent awareness*: viene definito il *Sacro Graal* del *context awareness* e rappresenta la capacità di anticipare le intenzioni dell'utente, basandosi su di una vasta gamma di informazioni, sia riguardanti l'ambiente sia riguardanti le azioni passate dell'utilizzatore.

Questi tipi di applicazioni, in particolare quelle appartenenti al quarto livello, rappresentano per gli sviluppatori un nuovo traguardo, il cui raggiungimento richiede inevitabilmente il supporto di *middleware* che forniscano loro un facile accesso alle risorse dei dispositivi, e li sollevi da problemi derivanti dalla eterogeneità hardware di quest'ultimi. Questi middleware sono rappresentati dalle varie piattaforme per dispositivi mobile, che hanno fatto la loro comparsa nel corso degli anni sul mercato internazionale.

1.3 Piattaforme mobile

Le piattaforme mobile costituiscono ad oggi uno degli elementi più significativi al momento della scelta di uno smartphone. Negli anni passati la sfida nel campo della telefonia riguardava principalmente i diversi costruttori di cellulari: ogni dispositivo era dotato di una semplice piattaforma operativa, sviluppata dall'azienda stessa, che doveva fornire un supporto ad operazioni basilari, come la gestione delle chiamate o l'invio e la ricezione di sms. Insieme ai dispositivi hanno dovuto evolversi anche queste piattaforme, che di fatto rappresentano il mezzo con cui è possibile offrire un'astrazione ad alto livello delle potenzialità derivanti dall'hardware; pertanto, esse si adattarono alla maggior complessità delle nuove funzionalità che, di volta in volta, venivano implementate.

Man mano che queste piattaforme divenivano più complicate e costose da sviluppare, alcune case produttrici decisero di abbandonare i loro software proprietari ed adottare soluzioni comuni sviluppate da terzi; esempi di questo genere sono *Windows Mobile*, largamente utilizzato da molti costruttori sino al 2009, o anche *Symbian*, che per anni ha contraddistinto gli smartphone di *Sony Ericsson*, *Samsung* ed in particolare *Nokia*. Ad oggi il mercato degli smartphone è conteso da quattro sistemi operativi, due dei quali sviluppati internamente dalle case produttrici, e sono:

- *BlackBerry OS*: sviluppato da *RIM (Research In Motion)* per i suoi smartphone. Molto più diffuso in USA e Canada che in Europa, era stato originalmente destinato ad un utilizzo di tipo *business*, offrendo quindi ottime funzionalità per l'organizzazione di impegni e gestione delle *email*. Vedendo la sua quota di mercato scomparire velocemente sotto il peso della concorrenza, anche questo sistema ha dovuto riadattarsi alle esigenze del mercato; infatti, offre oggi pieno supporto a funzionalità multimediali, e dispone di decine di migliaia di applicazioni scaricabili dal proprio *store*.

- *Windonws Phone*: sistema sviluppato da *Microsoft* e lanciato nella sua versione 7.0 verso la fine del 2010, è l'evoluzione del sistema *Windows Mobile*, da cui eredita il conteggio della versione, ma allo stesso tempo segna un netto taglio con esso, evidenziato dal cambiamento del nome. Recentemente, il 29 ottobre 2012, a pochi giorni di distanza dalla versione desktop, è stata rilasciata la versione 8.0, che introduce

il supporto ad architetture *multi-core*. Questa versione per smartphone condivide molto con quella per pc, sia ad alto livello, offrendo un'interfaccia grafica molto similare, sia a basso livello, poiché utilizza lo stesso *kernel* e lo stesso *file system*. Attualmente, questo sistema, grazie anche ad una alleanza stretta con *Nokia*, sta riguadagnando importanza nel contesto internazionale, ma detiene ancora soltanto una piccola fetta delle quote di mercato.

- *iOS*: è il sistema proprietario sviluppato da *Apple*, e utilizzato per una larga gamma di prodotti quali *iPhone*, *iPod*, *iPad* e anche il meno conosciuto *Apple TV*. Lanciato nel 2007, rappresenta una versione adattata per dispositivi mobile di *OS X*, utilizzato sui computers Apple. Questo sistema non prevede l'installazione su dispositivi non Apple, e non offre la possibilità di installare applicazioni non derivanti dall'*App Store* ufficiale, che comunque offre quasi un milione di applicazioni disponibili per il download. Nonostante queste limitazione imposte dalla casa, questo sistema operativo rappresenta una delle piattaforme mobile più diffuse al mondo, e solo in quest'ultimo anno ha mostrato lievi cenni di cedimento per quanto riguarda il settore vendite.

- *Android*: unico dei sistemi operativi fin qui presentati ad essere *open source*, è autore di una vera e propria scalata del mercato, che lo ha portato in meno di quattro anni ad essere in assoluto la piattaforma più diffusa al mondo. Sviluppato da *Google*, è utilizzato anch'esso sia su smartphone che su tablet di svariate marche, e inoltre offre un fornitissimo *market* per le applicazioni.

A fare la differenza tra una piattaforma e l'altra, agli occhi dell'utente finale, sono sicuramente le applicazioni che queste mettono a disposizione. Risulta quindi chiaro che, per aver tante applicazioni da offrire al pubblico, ogni casa deve riuscire ad attrarre più sviluppatori possibile dalla propria parte; tale processo avviene in due passi fondamentali:

- Offrire agli sviluppatori possibilità di guadagno tramite la pubblicazione delle loro applicazioni sui relativi store, garantendogli quindi un'ampia visibilità e facilitando gli incassi derivanti da vendita o pubblicità;

- Fornire un valido supporto allo sviluppo tramite *SDK* dedicati, comprensivi di apposite *API*, strumenti per il *testing* ed il *debugging*, e generalmente di un *IDE* per facilitare lo sviluppo.

1.4 Obiettivo ed organizzazione della tesi

A fronte delle premesse effettuate sul mondo degli smartphone, risulta interessante indagare sui mezzi a disposizione degli sviluppatori, per affrontare la sfida rappresentata da applicazioni che devono essere in grado di reagire ai diversi stimoli derivanti dall'ambiente.

L'obiettivo di questo lavoro è effettuare un'analisi del modello di programmazione proposto da Android. L'attenzione verrà posta, in particolare, su quali meccanismi vengano forniti per la gestione di eventi asincroni generati dal sistema, allo scopo di notificare cambiamenti del contesto in cui si sta operando: dal modo in cui vengono intercettati, a come risulta possibile modificare il comportamento dell'applicazione, in reazione alle nuove informazioni acquisite. Si valuteranno gli elementi di novità introdotti nelle API di Android, in relazione ai classici mezzi disponibili nella programmazione standard in *Java*, atti a risolvere una nuova categoria di problematiche dovute alla natura *context-aware* delle applicazioni.

Sarà effettuata anche un'analisi più generale della qualità del modello proposto, in termini di estensibilità e modularità del codice; per fare ciò, si prenderà in esame un caso di studio e si proporranno delle possibili estensioni per verificarne la fattibilità. Ciò premesso, il lavoro che viene proposto sarà suddiviso come segue:

- Nel secondo capitolo verrà presentato il sistema operativo Android, fornendone inizialmente una descrizione sia della struttura sia del modello di esecuzione che implementa, e proseguendo poi con una descrizione accurata delle componenti basilari per lo sviluppo di un'applicazione, supportata dall'utilizzo di alcuni esempi. Alla fine del capitolo verrà affrontato l'argomento del *threading*, mostrando gli strumenti presenti nelle API per la gestione di differenti flussi d'esecuzione;

- Nel terzo capitolo si prenderà in esame come caso di studio l'applicazione *SMS Backup+*; attraverso l'analisi di quest'ultima si testerà il modello di programmazione proposto da Android, portando alla

luce eventuali problemi pratici, che rimarrebbero altrimenti nascosti ad uno studio limitato agli esempi riportati dalla manualistica ufficiale. Alla fine del capitolo si effettuerà, come anticipato, un'analisi sull'estensibilità del codice promosso dal modello Android, grazie all'implementazione di alcune estensioni dell'applicazione;

- Il capitolo conclusivo è dedicato alle osservazioni finali elaborate durante il percorso di analisi, esposte separando gli aspetti specifici del modello di programmazione, da quelli di esclusiva pertinenza dell'applicazione presa in esame nel capitolo precedente.

Capitolo 2

Piattaforma Android

La piattaforma Android nasce dall'idea di fornire un sistema *general purpose* per i moderni dispositivi portatili come *smartphone* e *tablet*. Il progetto Android nasce del 2005, con l'acquisizione da parte di Google della startup *Android Inc*. Da quel momento furono necessari due anni di lavoro per giungere alla sua presentazione, avvenuta nel novembre del 2007 da parte della *Open Handset Alliance*, consorzio di 84 compagnie dedito a promuovere lo sviluppo di standard aperti per dispositivi mobile, accelerandone l'innovazione e offrendo ai consumatori un'esperienza migliore, più completa e meno costosa [1]. Il rilascio ufficiale della versione 1.0, denominata *Apple Pie*, avvenne il 23 settembre 2008 in concomitanza con la presentazione del primo smartphone, sviluppato dalla compagnia taiwanese *HTC (High Tech Computer)*. Tale dispositivo, che prendeva il nome di *T-Mobile G1* per il mercato statunitense e di *HTC Dream* per quello europeo, ebbe un discreto successo, affermando conseguentemente l'entrata di Android nel mondo dei sistemi operativi per smartphone. Ad oggi il sistema ha subito numerose evoluzioni, fino a raggiungere la versione 4.2 denominata Jelly Bean; secondo i dati riportati da IDC [6], Android è presente sul 75% dei dispositivi venduti nel terzo quadrimestre del 2012, ed è il sistema operativo *mobile* più diffuso al mondo.

2.1 Introduzione all'ecosistema Android

Da un punto di vista architetturale Android segue un modello a pila, come illustrato in figura 2.1; in seguito ne verranno descritti i *layer* principali.

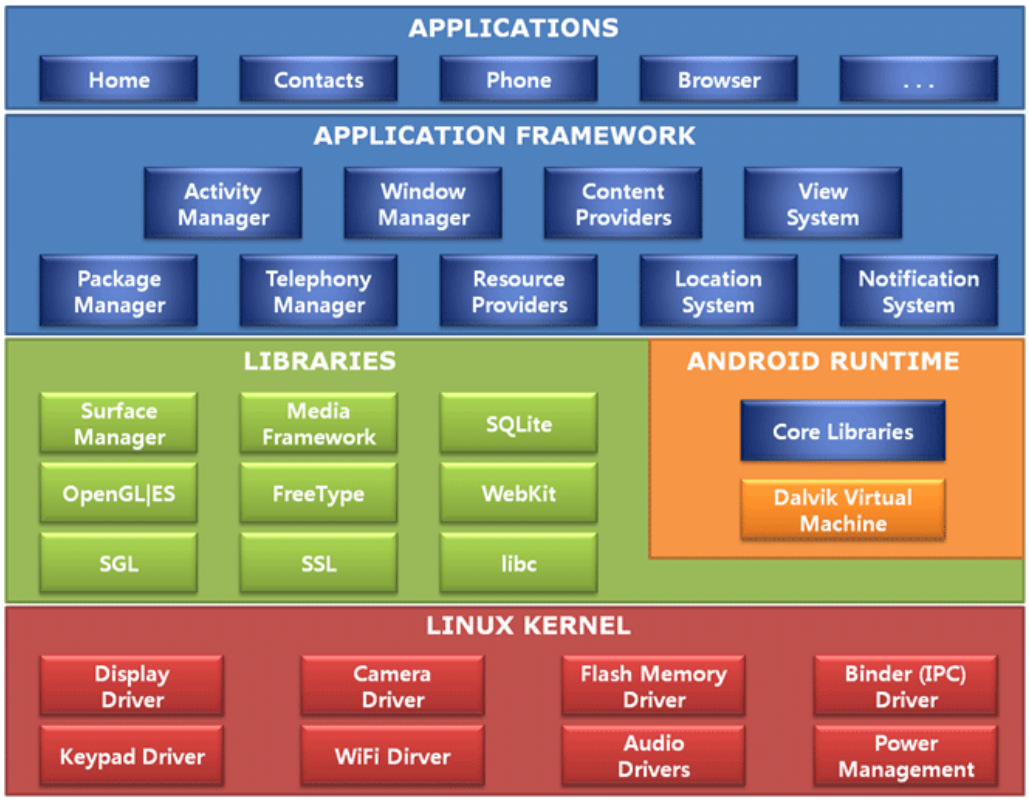

Figura 2.1: Rappresentazione grafica del modello a pila di Android

2.1.1 Linux Kernel Layer

Alla base dello stack software della piattaforma Android si ha un kernel Linux, che funge da livello di astrazione tra le componenti hardware del dispositivo e il resto del sistema, fornendo una serie di servizi fondamentali. Nella versione 1.0 Android era basato su Linux kernel 2.6, mentre l'attuale versione si basa su Linux kernel 3.4 . Il kernel si occupa principalmente di gestire i driver per l'hardware del dispositivo, di controllare la memoria e i processi, e di garantire la sicurezza del sistema ed il corretto utilizzo della rete e dell'energia.

2.1.2 Libraries Layer

Immediatamente sopra lo strato del kernel, Android include una serie di librerie scritte in *C/C++*, che implementano alcune delle funzionalità messe a disposizione dallo strato superiore; queste librerie possono esser sostituite da altre con il progredire delle versioni di Android, senza tuttavia compromettere le API che mette a disposizione. Al momento alcune delle librerie più importanti presenti in questo livello sono:

- **Libc**: implementazione della libreria standard C (libc), personalizzata per i sistemi Linux di tipo *embedded*; la dimensione di questa particolare versione è stata ridotta sino a circa la metà di quella originale;
- **PacketVideo's OpenCORE**: libreria per la registrazione e riproduzione di file multimediali;
- **Surface Manager**: libreria che gestisce le *View*, ovvero le componenti fondamentali dell'interfaccia grafica (come ad esempio bottoni, label o aree di testo), gestendone i problemi di renderizzazione più comuni;
- **OpenGL e SGL**: per la gestione della grafica 2D e 3D, come di consueto opportunamente ottimizzate per terminali mobile;
- **WebKit**: browser engine open source, utilizzato anche in Google Chrome e Safari. Non si tratta di un vero e proprio browser, tuttavia fornisce il supporto necessario per l'implementazione di applicazioni di questo genere;

- **SQLite**: *DBMS* relazionale, messo a disposizione da Android per permettere agli sviluppatori di salvare in modo persistente le informazioni delle proprie applicazioni sul dispositivo;

- **FreeType**: libreria per il supporto alla gestione dei font;

- **SSL**: libreria per la gestione del protocollo di crittografia *Secure Sockets Layer*, il quale permette una comunicazione sicura dal mittente al destinatario (*end-to-end*) su reti *TCP/IP*, mettendo a disposizione degli sviluppatori un solido sistema di autenticazione, integrità dei dati e cifratura.

2.1.3 Dalvik Virtual Machine

In ambiente Android, sia le applicazioni che le API messe a disposizione degli sviluppatori sono scritte in linguaggio JAVA, e necessitano quindi di una Java Virtual Machine per funzionare. Sapendo di avere a che fare con dispositivi con potenzialità hardware fortemente inferiori a quelle di un comune *desktop computer*, Google ha speso molte risorse nell'ottimizzare la JVM sviluppata da *Sun Microsystems*, cercando di ridurne lo spazio utilizzato in memoria, di migliorarne le performance e di allungare la vita della batteria del dispositivo.
La figura chiave di questo progetto è il dipendente di Google Dan Bornstein, il quale ha progettato la *Dalvik VM* (il nome *Dalvik* deriva dal suo paese di origine).

La Dalvik VM prende il bytecode contenuto nei files *.class* e lo trasforma in uno o più file Dalvik Excecutable (*.dex*), ottenendo files eseguibili di dimensioni quasi dimezzate rispetto all'origine. Il fatto che Android, ed in particolare la Dalvik VM, lavori con file .dex significa che non è possibile eseguire direttamente del Java bytecode; occorre invece partire dalle classi sorgenti (.java) e ricompilarle. Analogamente alla JVM, anche la Dalvik VM implementa una sua versione del *Garbage Collector*, sollevando quindi il programmatore dal compito di una gestione scrupolosa della memoria. Inoltre, come verrà spiegato in dettaglio successivamente, ogni singolo processo viene eseguito, per motivi di sicurezza, su una propria istanza riservata della VM.

2.1.4 Application Framework Layer

Questo strato fornisce agli sviluppatori una serie di API, grazie alle quali possono interfacciarsi con le librerie c/c++ dello strato sottostante. Queste API sono le medesime sia per gli sviluppatori ufficiali Google sia per sviluppatori indipendenti, che siano aziende o soggetti privati. Un concetto sul quale si è puntato molto è quello del riutilizzo del codice; l'application framework permette infatti di creare applicazioni in grado di fornire servizi ad altre applicazioni. Con questa strategia si cerca di incoraggiare gli sviluppatori ad utilizzare il lavoro già svolto da altri, in modo da non dover affrontare nuovamente problematiche già note in fase di realizzazione dei progetti.
Gli elementi principali presenti in questo framework sono:

- **Activity Manager**: componente che gestisce il *lifecycle* delle Activities. Queste ultime, che saranno descritte più dettagliatamente in seguito, sono le componenti delle applicazioni che si occupano di fornire all'utente una schermata con cui interagire. La activity manager si occupa quindi di gestirle correttamente, in risposta alle azioni dell'utente o ad eventi generici come, ad esempio, la ricezione di una chiamata;

- **Content Provider**:componente che gestisce la condivisione di dati tra più applicazioni. Costituisce uno dei componenti fondamentali delle applicazioni Android, e verrà descritto dettagliatamente in seguito;

- **Resource Manager**: questa componente si occupa della gestione di tutte le risorse presenti nelle applicazioni che non sono classificabili come codice sorgente. Si occupa ad esempio delle immagini incluse nelle view, dei file di configurazione dei layout o delle stringhe definite nell'apposito file xml;

- **Window Manager**: questa componente fornisce un'astrazione del Surface Manager, descritto precedentemente; mette quindi a disposizione un supporto per la gestione delle finestre nelle applicazioni;

- **View System**: gestisce tutte le componenti messe a disposizione per formare le view delle applicazioni, quali pulsanti o aree di testo;

- **Package Manager**: modulo che gestisce i processi di installazione, aggiornamento e rimozione delle applicazioni. Su Android, prima di esser installata, ogni applicazione è rappresentata da un file di estensione *.apk (Android Package)* contenente tutte le informazioni necessarie per una corretta installazione ed esecuzione;

- **Telephony Manager**: le tipiche funzioni di un cellulare vengono gestite attraverso questa componente che permette, ad esempio, l'invio di SMS o la lettura di informazioni sullo stato delle chiamate;

- **Location Manager**: è il modulo contenente le API utilizzate per ottenere informazioni sulla geolocalizzazione del dispositivo. Si possono ottenere informazioni sulla localizzazione attraverso l'uso del *GPS*, sfruttando quindi il sistema dei satelliti geostazionari, oppure sfruttando le informazioni delle reti alle quali si è connessi, come ad esempio le celle *GSM* o connessioni *Wi-Fi*;

- **Notification Manager**: permette alle applicazioni di notificare al sistema una serie di eventi e di scegliere come queste notifiche debbano essere visualizzate dall'utente. I metodi più comuni prevedono l'utilizzo della barra delle notifiche, della vibrazione, delle suonerie o dell'accensione di *LED* (il risultato di quest'ultima operazione può variare a seconda del modello del dispositivo).

2.1.5 Applications layer

In cima allo stack software si trovano le applicazioni destinate all'utente finale. Ogni versione di Android fornisce una suite di applicazioni di base quali browser, rubrica e gestore degli SMS, e ogni casa produttrice integra nei propri dispositivi altre applicazioni generalmente sviluppate internamente (ad esempio applicazioni per il backup dei dati, o la connettività con altri dispositivi della stessa marca). Si vuole sottolineare come non vi sia alcuna differenza tra le applicazioni preinstallate e quelle sviluppabili da qualsiasi altro soggetto: tutte le applicazioni in Android sfruttano le stesse API e godono degli stessi privilegi, in accordo con la filosofia OHA: "All applications are created equal"[1].

2.1.6 Ambiente di esecuzione

L'esecuzione di una qualsiasi applicazione richiede l'utilizzo della DVM, in quanto scritta utilizzando il linguaggio Java. Per poter garantire un elevato grado di sicurezza viene utilizzata una tecnica chiamata *security sandbox*, che prevede l'esecuzione di ogni applicazione su una istanza dedicata della macchina virtuale.

Android si basa su un sistema Linux multiutente, dove ogni applicazione viene gestita come se rappresentasse un diverso utente. Questo metodo prevede l'assegnazione di uno *user ID* univoco ad ogni processo (questo ID è conosciuto solo dal sistema e nascosto alle applicazioni). Attraverso l'utilizzo dell'ID il sistema è in grado di gestire i permessi per l'accesso ai files dell'applicazione, in modo che vi possa accedere solo ed esclusivamente quel processo. In alcuni casi particolari, in cui è necessario avere una forte condivisione dei dati, è possibile che due o più applicazioni condividano lo stesso user ID, ottenendo in questo modo il diritto di accesso concorrente agli stessi files. Coerentemente, le applicazioni con lo stesso ID condivideranno anche la stessa istanza della DVM e lo stesso processo (è bene chiarire che per ogni processo possono esistere uno o più *thread* distinti).

Il processo, relativo ad una specifica applicazione, viene avviato dal sistema ogni volta che uno qualsiasi dei componenti dell'applicazione richiede di esser eseguito, e viene terminato quando non è più necessario il suo utilizzo. Android si riserva anche la possibilità di terminare processi senza preavviso nel caso in cui debba recuperare memoria; la scelta del processo da terminare avviene tramite un algoritmo di *ranking*, che assegna ad ogni processo un punteggio a seconda dell'importanza. A titolo di esempio, è più probabile che si scelga di terminare un processo in background piuttosto che eliminare il processo responsabile dell'interfaccia grafica con cui sta attualmente interagendo l'utente. In questo modo Android implementa il *principio del privilegio minimo*, secondo il quale ogni applicazione può accedere solo ed esclusivamente alle componenti di cui ha realmente bisogno istante per istante per svolgere il suo lavoro, contribuendo così a creare un ambiente sicuro in cui non è possibile accedere a parti del sistema per le quali non si hanno i permessi. Questi ultimi, necessari per accedere ai dati del dispositivo (quali rubrica, scheda di memoria SD, fotocamera, SMS, contenuti multimediali ed altri), devono esser richiesti all'utente al momento dell'installazione tramite un apposito avviso, e devono essere specificati

nel file *manifest.xml*.

2.2 Modello di programmazione

Dal punto di vista dello sviluppatore, le applicazioni Android risultano formate da un insieme di componenti che, interagendo tra loro, contribuiscono a delinearne e garantirne il funzionamento.
Le principali componenti utilizzate per creare una applicazione, che verranno in seguito illustrate dettagliatamente, sono:

- *Activity*
- *Service*
- *Broadcast Receiver*
- *Content Provider*

Ognuna delle componenti è gestita come entità indipendente; essa possiede infatti un proprio ciclo di vita ed ha un ruolo ben definito all'interno dell'applicazione. Questa separazione dei compiti deriva dalla politica di riusabilità promossa da Android, secondo la quale un'applicazione può richiedere l'utilizzo di componenti appartenenti ad altre applicazioni installate sul dispositivo. Per far sì che il sistema conosca quali componenti sono implementate dall'applicazione, e quali tipi azioni queste possono soddisfare, viene utilizzato il file *AndroidManifest.xml*, nel quale devono essere registrate le suddette componenti (se una di queste non compare in tale file, è come se non esistesse agli occhi del sistema). Il file AndroidManifest.xml svolge quindi un ruolo di interfaccia dell'applicazione verso il sistema.

Si può a questo punto notare una prima fondamentale differenza dalle normali applicazioni Java, dove la classe *Main* rappresenta l'unico *entry point*. Al contrario, in Android è previsto che più di una delle proprie componenti possa esser utilizzata per lanciare l'applicazione.

2.2.1 Intent e Intent Filters

Prima di cominciare la trattazione delle componenti precedentemente elencate è bene introdurre il concetto di *Intent* e di *Intent Filter*, poiché risulteranno fondamentali per comprendere appieno le nozioni che verranno

esposte. Queste astrazioni vengono utilizzate dal sistema per implementare una politica di comunicazione tra componenti di tipo *message passing*, in cui l'intent ricopre il ruolo del messaggio. Come si vedrà questo tipo di comunicazione è fortemente utilizzata nel modello di programmazione Android. La vera particolarità di questo metodo di comunicazione risiede nel concetto di intent filter, che permette una semplice implementazione di un meccanismo di *late binding* tra le componenti della stessa applicazione o di applicazioni differenti.
Le informazioni che vi possono essere racchiuse sono:

- *Component name*: il nome della componente che sarà chiamata a gestire l'intent. Questo campo è composto dal nome esteso della classe target, comprensivo del percorso relativo al package in cui si trova, e dal nome del *package* presente nel file manifest dell'applicazione a cui appartiene. Questo è un campo opzionale, se presente si parlerà di *Explicit Intent* che verrà esaminato in seguito;

- *Action*: una stringa contenente il nome dell'azione che si desidera venga compiuta. All'interno della classe *Intent* e di altre classi Android sono definite alcune azioni standard, ma è comunque possibile crearne di personalizzate;

- *Data*: questo campo contiene un riferimento ai dati rappresentato mediante un *URI (Uniform Resource Identifier)*, al quale viene associato il corrispondente tipo MIME che può essere indicato esplicitamente o dedotto dal *Content Provider*. Un URI è costruito come segue: *scheme://authority/path*, dove l'authority rappresenta la combinazione *host:port*. Questo campo è strettamente legato all'azione da eseguire; ad esempio, un intent contenente l'azione *ACTION_CALL* avrà come campo *data* il numero di telefono da chiamare;

- *Category*: una stringa che fornisce informazioni aggiuntive sul tipo di componente che gestirà l'intent. Ad esempio, all'avvio di un'applicazione verrà cercata una activity marcata con *CATEGORY_LAUNCHER*;

- *Extra*: ogni tipo di informazione che non è compresa in quelle fino ad ora descritte rientra in questa tipologia. Android permette di aggiun-

gere all'intent alcune coppie di dati chiave-valore mediante il metodo *putExtras(Bundle extras)*, e di recuperarle con *getExtras()*.

La gestione delle intent inviate viene effettuata dal sistema tramite un meccanismo denominato *Intent Resolution*; tale processo può intraprendere due possibili strade, a seconda del tipo di intent da gestire.

Intent Esplicito

In un intent di questo tipo viene specificato esplicitamente il nome della classe target attraverso il campo *component name*. Il processo di intent resolution non è quindi necessario, in quanto le informazioni così fornite sono sufficienti alla determinazione della componente da invocare. La creazione di un intent esplicito avviene tramite l'uso del seguente costruttore:

Intent(Context packageContext, Class<?>cls)

oppure:

Intent(String action, Uri uri, Context packageContext, Class<?>cls)

L'oggetto di tipo *Context* rappresenta il mittente del messaggio, mentre *Class* indica la classe dell'oggetto che dovrà gestirlo.

Intent Implicito

Questa tipologia di intent sfrutta il concetto di intent filter ,e necessita del processo di intent resolution. Un intent implicito non specifica il nome della classe chiamata a gestirlo; tale scelta verrà effettuata dal sistema, il quale cercherà la corretta corrispondenza alle informazioni da lui contenute. Il processo di intent resolution terrà conto di tre informazioni:

- quale action deve essere soddisfatta;
- la categoria; se ne vengono specificate due o più, queste devono esser tutte presenti;
- il tipo di dato fornito; se non esplicitamente dichiarato si utilizza lo *scheme* dell'URI.

Nella fase di *matching* di queste informazioni giocano un ruolo fondamentale gli intent filter dichiarati nel file *AndroidManifest.xml*. Di seguito se ne propone un esempio:

```
<activity:name=".MyBrowser"
   android:label="@string/browser_name">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="http" />
    <data android:scheme="https" />
  </intent-filter>
</activity>
<activity android:name=".MyActivity"
   android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name=
            "android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Nell'esempio sono registrate due activities; la prima chiamata *MyBrowser* potrebbe essere scelta dal sistema per rispondere ad intent aventi come campo azione *VIEW*; essendo tuttavia un'azione generica, si controlla il tipo di dato presente nello *scheme* dell'URI, che conterrà 'http' nel caso in cui l'ambito di esecuzione corrisponda ad un sito web. La seconda activity contiene la definizione di intent filter, utilizzata per indicare al sistema che tale activity è la prima da esser lanciata all'avvio dell'applicazione. Questo sistema è stato ideato per permettere l'utilizzo di componenti di altre applicazioni senza dover preoccuparsi di quali siano effettivamente installate sul dispositivo.

2.2.2 Activity

Le *Activities* sono componenti in grado di fornire all'utente una schermata con la quale interagire, per poter eseguire un qualche tipo di azione o attività (effettuare una chiamata, inviare un sms, visualizzare una mappa). Un activity può quindi contenere qualsiasi elemento di visualizzazione, come ad esempio immagini o testi, o interattivo, come i bottoni. Un'applicazione,

che non sia un *toy example*, è solitamente formata da più activities, ad ognuna delle quali è affidata una diversa schermata.

Life Cycle

L'aspetto probabilmente più importante per chiunque voglia sviluppare applicazioni Android è il *Life Cycle* di una activity, la cui padronanza risulta fondamentale ai fini della comprensione del modello di programmazione utilizzato. Le tre principali modalità nelle quali una activity può trovarsi durante il suo periodo di vita sono:

- *Running*:la activity è visualizzata sullo schermo e pronta ad interagire con l'utente;
- *Paused*: l'utente non può più interagire con la activity ma il suo layout è ancora visibile (ad esempio, sopra di esso appare una finestra di dialogo). Le activity in questa modalità mantengono tutte le informazioni riguardanti il loro stato e possono essere terminate dalla DVM nel caso di scarsità di risorse disponibili;
- *Stopped*: si entra in questa modalità quando l'utente passa ad un'altra activity. Le attività in stop sono le prime ad essere terminate in caso di necessità di ulteriori risorse.

Callback

La gestione di queste componenti avviene mediante un insieme di metodi di *callback*, che il sistema richiama per notificare un cambiamento di stato. Il progettista può sfruttare questi metodi per conferire alla activity il comportamento desiderato, a seconda dello stato in cui essa si trova. Come ribadito più volte nel manuale, se nella propria activity si ridefinisce un metodo di callback è indispensabile richiamare il corrispettivo metodo della classe padre, solitamente rappresentata dalla classe *android.app.Activity*. Nella figura 2.2 viene riportato un modello rappresentante i possibili stati assumibili da una activity e i metodi di callback associati ad ogni transizione.

Ipotizzando di partire dalla prima chiamata ad una activity, i metodi di callback che di volta in volta saranno coinvolti sono:

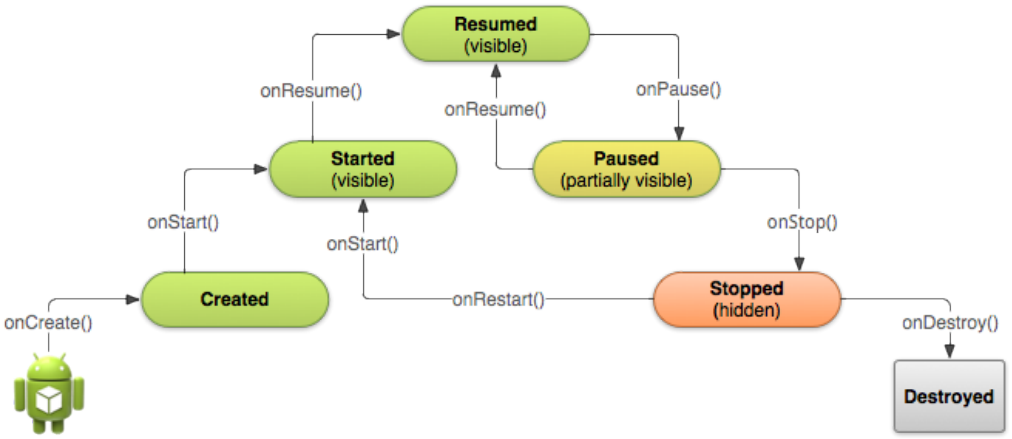

Figura 2.2: Modello degli stati di una activity

- *protected void onCreate(Bundle savedInstanceState)*: metodo richiamato alla creazione della activity. All'interno di questo metodo è opportuno effettuare l'inizializzazione degli elementi essenziali; inoltre è la parte di codice nella quale si chiama il metodo *setContentView()*, per esplicitare quale tipo di layout esporre. Terminata l'esecuzione di questo metodo è corretto presupporre che la activity esista all'interno del sistema, senza tuttavia essere ancora visibile (*stopped*);

- *protected void onStart()*: metodo che prepara la activity per la visualizzazione; in questa fase sarebbe opportuno effettuare operazioni come la registrazione di *Broadcast Receiever* (nel caso in cui gli eventi sui quali è in ascolto comportino modifiche della UI). Terminato questo metodo la activity si troverà in uno stato visibile ma che non permette ancora l'interazione con l'utente (*paused*). Se la activity si trovava precedentemente in uno stato di stop, verrà prima richiamato il metodo *onRestart()*; questo avviene perché può essere utile differenziare una situazione di creazione da una di ripristino, al fine di poter gestire il recupero dello stato precedente;

- *protected void onResume()*: eseguito immediatamente dopo il metodo *onStart()*, notifica che la activity è ora in primo piano e sensibile ai comandi dell'utente (*running*). Questa fase rappresenta il mo-

mento adatto per avviare animazioni, utilizzare dispositivi ad accesso esclusivo (come ad esempio la fotocamera) o, più in generale, qualsiasi attività che ha senso di esistere solo nel momento in cui gode dell'attenzione diretta da parte dell'utilizzatore;

- *protected void onPause()*: metodo duale a *onResume()*, viene chiamato quando la activity, pur rimanendo parzialmente visibile, passa in secondo piano a causa della perdita del focus, e diventa insensibile alle interazioni dell'utente. Pur non essendo l'ultimo metodo in un normale processo di arresto di una schermata, *onPause()* è ultimo di cui viene garantita l'esecuzione. In casi di estrema necessità di risorse Android potrebbe distruggere la activity senza preavvisi, e senza eseguire i metodi *onStop()* e *onDestroy()* (descritti più avanti); ciò vincola il progettista ad inserire in questa fase le istruzioni per il salvataggio consistente dei dati. E' buona norma, inoltre, disattivare le animazioni precedentemente attive, e più in generale tutto ciò che richiede l'utilizzo della CPU. Questo metodo solitamente notifica uno stato in cui la activity permane per un tempo breve; infatti, indipendentemente dalla necessità dell'utente di far tornare in primo piano la schermata o di terminarla, le operazioni eseguite devono essere in entrambi i casi molto rapide (evitando ad esempio chiamate a servizi remoti), assicurando in questo modo un buon livello di reattività nel passaggio tra schermate che eran state mantenute visibili e garantendo all'utente un'esperienza d'utilizzo positiva;

- *protected void onStop()*: metodo duale di *onStart()*, viene richiamato quando si ha la necessità di segnalare che la activity è completamente invisibile, ma è ancora esistente. Al termine di questa fase la activity si trova nello stato di stop, e potrà esser riportata in uno stato visibile oppure essere eliminata; in quest'ultimo caso, ne verrà notificata l'intenzione con il metodo *onDestroy()*;

- *protected void onDestroy()*: metodo duale di *onCreate()*, è l'ultima callback prima dell'eliminazione della activity, e deve esser usata per rilasciare tutte le risorse ancora legate ad essa.

Back Stack

Esaminato come il sistema notifica il cambiamento di stato di un activity, saranno ora analizzate più in dettaglio le singole chiamate ai metodi descritti finora; l'obiettivo è comprendere come avviene la gestione degli stati dell'applicazione, sia in situazioni lineari sia nel caso in cui siano presenti più schermate.

Si immagini di avere un'applicazione composta da quattro activities, ognuna delle quali contiene un bottone utilizzabile per passare alla activity successiva, si veda figura 2.3. All'avvio dell'applicazione, selezionando l'icona corrispondente nel menu, verrà creata e visualizzata la prima activity, eseguendo in sequenza i metodi *onCreate()*, *onStart()* e *onResume()* di quest'ultima; ogni volta che verrà premuto il bottone 'Start Activity x' apparirà sullo schermo la nuova activity, eseguendone la medesima sequenza di metodi. Sulla precedente activity verranno chiamati i metodi *onPause()* e *onStop()*, al termine dei quali risulterà non più visibile ma ancora esistente. Un caso del tutto analogo si verificherebbe alla pressione del tasto 'Home', il quale riporta alla schermata principale di Android, rendendo non visibile l'ultima activity senza tuttavia distruggerla. La pressione del tasto 'Indietro' sul dispositivo provocherà l'invocazione dei metodi *onPause()*, *onStop()* e *onDestroy()* sulla activity correntemente visualizzata, che verrà quindi distrutta, e *onRestart()*, *onStart()*, *onResume()* sulla precedente, che verrà di conseguenza riportata in primo piano.

Per gestire situazioni come quella descritta, Android si avvale di una struttura a pila denominata *Back Stack*, nella quale viene effettuato il *push* di ogni nuova activity creata ed il *pop* quando quella correntemente visualizzata viene terminata. La activity che di volta in volta si ritroverà in cima alla pila sarà quella visualizzata in primo piano. Viene associato un Back Stack ad ogni *task*, parola utilizzata in Android per esprimere il concetto di un insieme di activity con cui l'utente può interagire per svolgere un certo lavoro. In un task possono essere lanciate activities di differenti applicazioni; si pensi per esempio ad un programma per l'invio di sms che offra la possibilità di scattare e inviare foto: la activity che permette l'uso della fotocamera apparterrà ad un'altra applicazione ma verrà gestita dal Back Stack dello stesso task. Quando il Back Stack, a causa di ripetute pressioni del tasto 'Indietro', non conterrà più activities, cesserà di esistere insieme al task a cui era associato.

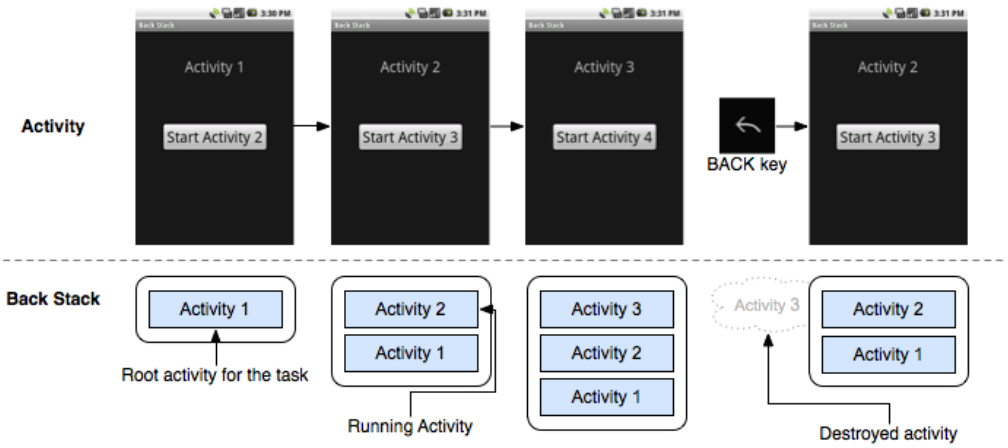

Figura 2.3: Schema di corrispondenza fra Activity e Back Stack

Salvataggio dello stato di una activity

Come precedentemente illustrato, il salvataggio di dati su supporti persistenti andrebbe effettuato nel metodo di callback *onPause()*, poiché quest'ultimo è l'unico metodo la cui esecuzione è garantita, nel caso in cui Android decidesse di distruggere una activity per recuperare memoria.

Oltre alle informazioni fondamentali per l'applicazione risulta utile salvare anche quelle riguardanti lo stato della activity, in modo tale che, quando il sistema andrà a ricrearla, questa si ritroverà nella stessa situazione in cui si trovava in precedenza. Si pensi per esempio ad un utente che ricerca un file da allegare ad una email: senza un adeguato salvataggio dello stato della activity che si occupa della gestione dell'email, il sistema potrebbe decidere di distruggere quest'ultima mentre è in uso la activity che gestisce la ricerca del file, causando un forte problema se è già stato scritto un lungo testo nel corpo della email da parte dell'utente.

Per far fronte a questa evenienza, Android mette a disposizione altri due metodi di callback: *onSaveInstanceState(Bundle outState)* e *onRestoreInstanceState(Bundle savedInstanceState)*. Il primo viene chiamato prima che la activity diventi vulnerabile alla distruzione, ovvero prima dell'esecuzione del metodo *onPause()*, e viene utilizzato per le operazioni di salvataggio. Il secondo viene richiamato dopo *onStart()* e serve per il recupero dello stato precedente. Si evince dalla dichiarazione dei due metodi che tale meccani-

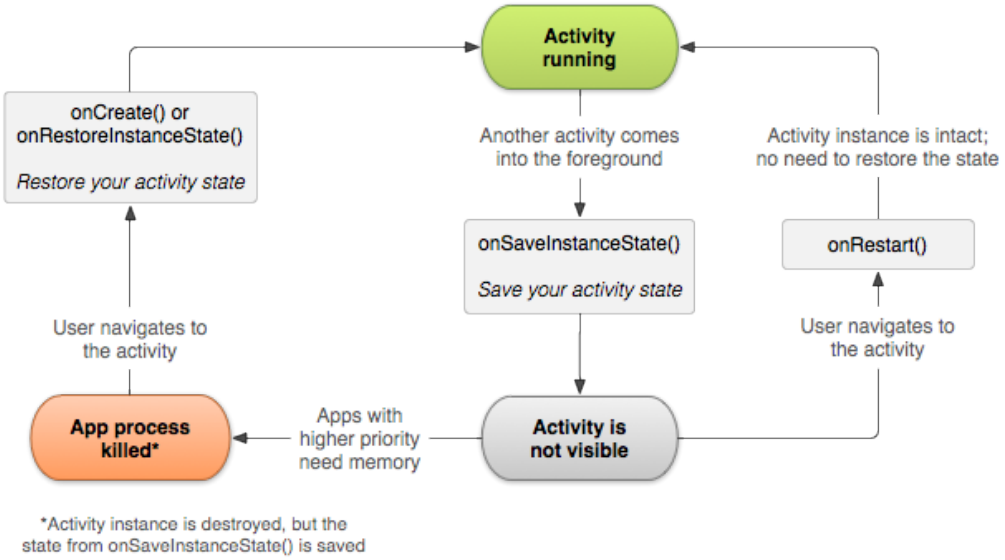

Figura 2.4: Flusso di esecuzione dei metodi per la gestione degli stati

smo sfrutta come tipo di dato il *Bundle*, ovvero un contenitore nel quale è possibile inserire le informazioni necessarie in forma di coppie di tipo chiave-valore. Da notare come anche il metodo *onCreate()* riceva in ingresso un Bundle contenente le informazioni riguardanti il precedente stato della activity; nel caso in cui l'informazione non serva è possibile passare un parametro *null*.

Per sollevare il programmatore dal noioso compito di implementare tali funzioni per ogni activity, Android effettua automaticamente le operazioni di salvataggio e ripristino degli stati di ogni oggetto di tipo *View* presente nel layout a cui è associato un ID univoco, lasciando allo sviluppatore il compito di salvare solo le informazioni addizionali (quali, ad esempio, il punteggio nel caso di un videogioco).

Utilizzare una Activity

Per comprendere meglio quanto detto fino ad ora, viene illustrata qui di seguito l'implementazione di un esempio di applicazione composta da una singola activity. Quest'ultima ha l'unica funzione di contare e mostrare

a schermo il numero di chiamate ai metodi di callback precedentemente illustrati.

```
public class CounterActivity extends Activity {

TextView tv;
int contStart,contResume,contPause,contStop;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_counter);

    tv= (TextView) findViewById(R.id.textViewCounter);
    contStart=0; contResume=0;
    contPause=0; contStop=0;
  }
```

Per prima cosa si estende la classe *Activity* e si ridefinisce il metodo *onCreate()*, inizializzando gli attributi ed impostando il layout della activity tramite il metodo *setContentView()*. Successivamente si ridefiniscono tutti gli altri metodi di callback, richiamando il metodo della classe padre ed incrementando un semplice contatore; nel metodo *onResume()* viene aggiornato l'unico elemento visivo della activity, costituito da un campo di testo compilato con i valori attuali dei contatori:

```
@Override
  public void onStart(){
    super.onStart();
    contStart++;
  }

  @Override
  public void onResume(){
    super.onResume();
    contResume++;
    tv.setText("Start: "+contStart+"; Resume: "+contResume
        +"; Pause: "+contPause+"; Stop: "+contStop);
  }

  @Override
```

```
public void onPause(){
  super.onPause();
  contPause++;
}

@Override
public void onStop(){
  super.onPause();
  contStop++;
}
```

Lanciando l'applicazione sull'emulatore, e premendo alcuni tasti per forzare la chiamata dei metodi reimplementati, ci si aspetta che il valore dei contatori cambi secondo il life cycle descritto. In particolare, all'apertura dell'applicazione dovranno essere chiamati i metodi *onStart()* e *onResume()*, premendo *Home* e riaprendo l'applicazione ci si aspetta che vengano chiamati tutti e quattro i metodi, mentre con il tasto *Block screen*, che manda la activity in uno stato *paused*, ci si aspetta solo la chiamata ai metodi *onPause()* ed *onResume()*. Alla pressione del tasto *Back*, non essendo presenti altre activities nel back stack, ci si aspetta la chiusura dell'applicazione e quindi l'azzeramento dei contatori.

Il risultato ottenuto è riportato in figura 2.5.

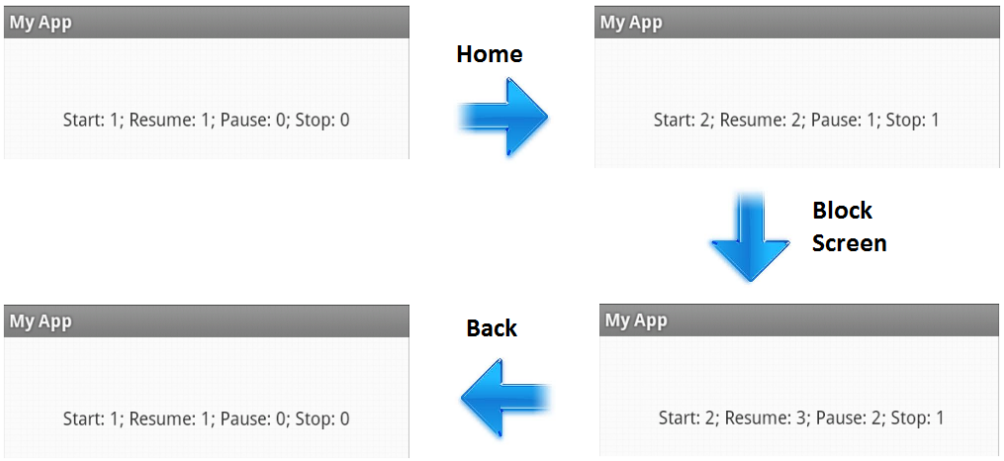

Figura 2.5: Chiamate ai metodi di una Activity durante l'esecuzione dell'applicazione

2.2.3 Service

Un *Service* viene descritto come la componente messa a disposizione da Android per effettuare lunghe operazioni computazionali; come vedremo in seguito, questa affermazione è vera solo in parte a causa della particolare gestione dei flussi di controllo.

Una delle caratteristiche principali di un service è quella di esser privo di interfaccia grafica; esso può essere infatti paragonato ad un servizio Windows o ad un demone Unix, e analogamente a questi è sempre disponibile anche se non attivo. Nonostante non abbia una sua interfaccia grafica, un service può comunque comunicare con l'utente tramite *Toast*, piccoli messaggi che appaiono per un tempo limitato sulla superficie dello schermo, o tramite notifiche nella barra di stato. Date le sue caratteristiche il service appare ideale per realizzare operazioni che non richiedano un feedback visivo da parte dell'utente (ad esempio accesso a servizi remoti, riproduzione musicale e operazioni di I/O).

Come le altre componenti anche il service gode di un proprio *life cycle* che varia a seconda del tipo di service che si è deciso di utilizzare; le due modalità di esecuzione, descritte nel paragrafo successivo, sono *local* e *remote*. In ogni caso, per utilizzare un service è necessario estendere la

classe *android.app.Service* o una delle sue sottoclassi, e ridefinire alcuni suoi metodi di callback per ridefinirne il comportamento.

Local service

Questo tipo di service viene lanciato quando una componente dell'applicazione chiama il metodo *startService(Intent service)*; a questo punto, il service sarà attivo fino a quando non verrà chiamato il metodo *stopService(Intent service)* o il service stesso non utilizzerà *stopSelf()*, anche se la componente che lo ha invocato dovesse cessare di esistere. L'oggetto *Intent* passato ai due metodi serve ad indicare ad Android quale servizio si vuole eseguire, ed eventualmente può essere utilizzato per fornirgli dei dati necessari al lavoro che deve svolgere. Viene definito 'locale' perchè è in grado di interagire solo con le componenti che appartengono al medesimo processo della componente che lo ha lanciato. A seguito della chiamata *startService()* vengono invocati in sequenza 2 metodi di callback:

- *onCreate()*, contenente le operazioni di inizializzazione; questo metodo viene eseguito solo per la prima invocazione di *startService()*;
- *onStartCommand(Intent intent, int flags,int startId)*, invocato ad ogni chiamata di *startService()*; all'interno di questo metodo deve essere specificata la *business logic* del servizio. Oltre all'intent gli altri due parametri forniscono ulteriori dati riguardo alla richiesta; il metodo deve fornire inoltre un valore di ritorno rappresentato da costanti definite nella classe *Service*: questo valore serve a comunicare al sistema come comportarsi nel caso sia terminato il processo a cui appartiene.

Una volta che il service è attivo, questo può esser fermato o invocando gli appositi metodi o direttamente dal sistema, in qualunque momento; ciò comporta, com'è prevedibile, la chiamata della callback *onDestroy()*.

Esaminando il flusso di esecuzione di un local service, emerge evidente un problema di notevole importanza: esso viene infatti eseguito sul medesimo *thread* del componente che lo richiama (il quale solitamente è una activity), e ciò può comportare rallentamenti o, nel caso in cui l'operazione duri più di 5 secondi, la segnalazione di un errore *ANR (Activity Not Responding)* e la conseguente chiusura dell'applicazione da parte di Android.
Le possibili soluzioni a questo problema sono:

- Definire un nuovo thread all'interno del service che esegua le operazioni computazionali (preferibile se si ha la necessità di servire richieste multiple contemporaneamente);

- Estendere la classe *IntentService* (gestendo, in questo modo, solo una richiesta alla volta).

Senza entrare nel dettaglio della seconda soluzione, è sufficiente sapere che la classe IntentService fornisce:

- un *worker* thread per l'esecuzione delle richieste;

- una coda per la gestione delle richieste;

- la gestione automatica del metodo *stopSelf()*, chiamato non appena tutte le richieste sono state soddisfatte;

- un metodo *onHandleIntent(Intent intent)* all'interno del quale è possibile specificare le azioni da compiere.

Remote service

Un remote service, allo stesso modo di un local service, si implementa estendendo la classe *Service*; l'unica differenza è data dalla ridefinizione del metodo di callback *onBind()*. Questa operazione cambia drasticamente il comportamento ottenuto: infatti, a differenza dei local services, un remote service è sempre eseguito dentro un nuovo processo, evitando quindi problemi di rallentamento dell'interfaccia grafica.

Il cambiamento più radicale risiede però nella possibilità di esser utilizzato anche da componenti di altre applicazioni tramite *remote procedure call*. Come in altre soluzioni *RPC*, anche per i servizi remoti si utilizza un linguaggio per la definizione di interfacce che assume in Android la sigla *AIDL (Android Interface Definition Language)*. Una descrizione dettagliata dei file *.aidl* esula dagli scopi di questo trattato, e perciò si rimanda alla guida online [4].

La creazione di un remote service avviene attraverso una prima registrazione, effettuata da una qualsiasi componente presso quel servizio tramite il metodo *bindService(Intent service, ServiceConnection conn, int flags)*; il

parametro *conn* rappresenta l'oggetto che si occuperà di notificare al chiamante gli eventi di avvenuta connessione e disconnessione dal servizio. I metodi di callback predisposti alla gestione del life cycle di un servizio remoto sono:

- *onCreate()*, eseguito solo durante la prima registrazione al servizio;
- *onBind(Intent intent)*, la cui esecuzione comporta la restituzione di un oggetto *IBinder* rappresentante l'implementazione dell'interfaccia AIDL; questo metodo viene eseguito ad ogni nuova registrazione presso il servizio;
- *onUnbind(Intent intent)*, che viene chiamato ogni qualvolta un client si disconnette dal servizio; la deregistrazione avviene tramite il metodo *unbindService(ServiceConnection conn)*. Questo metodo provoca l'esecuzione del metodo *onRebind(Intent intent)* per la gestione delle richieste di connessione da parte di nuovi client;
- *onDestroy()*, chiamato quando tutti i client si sono disconnessi e nessuno sta più utilizzando tale servizio; il sistema distrugge quest'ultimo per liberare risorse e lo notifica della decisione tramite questo metodo.

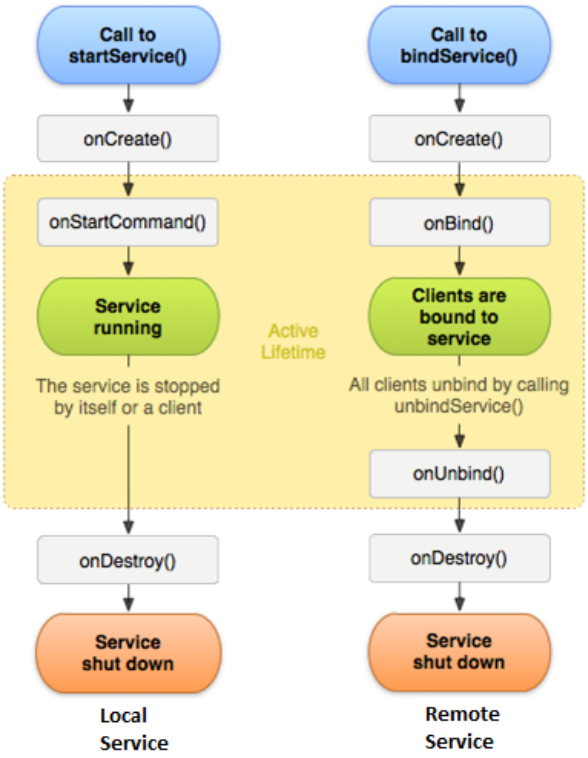

Figura 2.6: Life cycle di un local service (sinistra) e di un remote service (destra)

Utilizzare un service

Per fornire un esempio di utilizzo di un service, è possibile ampliare l'applicazione precedentemente creata, aggiungendo la definizione del service nel file *AndroidManifest.xml* nel modo seguente:

```
<service
   android:name=".MyCounterService" >
</service>
```

Si implementa la classe *MyCounterService*, la quale rappresenterà un local service, estendendo dalla classe *Service* e ridefinendo di quest'ultima i metodi di callback, come illustrato nel codice seguente:

```
public class MyCounterService extends Service {
```

```
private final static String TAG = "ServiceExample";

@Override
public void onCreate() {
  super.onCreate();
  Log.i(TAG, "[MyService] Created ");
}

//Si utilizza onStartCommand() per i local service
@Override
public int onStartCommand(Intent intent, int flags,
    int startId){
  Log.i(TAG, "[MyService] Started ");
  count();
  return START_NOT_STICKY;
}

private void count(){
  Log.i(TAG, "[MyService] Start counting to 3 ");
  for(int i=0; i<3; i++){
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  Log.i(TAG, "[MyService] Stop counting ");

  /*Si gestisce la chiusura del service
  * una volta terminato il lavoro */
  Log.i(TAG, "[MyService] Killing myself ");
  stopSelf();
}


@Override
  public IBinder onBind(Intent intent) {
    //Null in quanto è un local service
    return null;
```

```
    }

  @Override
  public void onDestroy() {
    Log.i(TAG, "[MyService] Destroyed");
    super.onDestroy();
  }
}
```

Il service così definito si occuperà di contare fino a tre attendendo un secondo ad ogni passaggio, e di scrivere su log l'operazione che è chiamato ad eseguire, allo scopo di evidenziarne il flusso di controllo. Per poter lanciare il service si è aggiunto un pulsante alla activity *CounterActivity*, alla quale è stato delegato il compito di gestire l'evento *onClick()* nel seguente modo:

```
public void onCreate(Bundle savedInstanceState) {
  ...
  //Recupero il riferimento al pulsante e registro
  // la activity come listener
  this.button = (Button) findViewById(R.id.buttonService);
  this.button.setOnClickListener(this);
  ...
}

public void onClick(View v) {
  /*Creo un intent esplicito per MyCounterService
  * e ne richiedo l'esecuzione */
  Intent intent = new Intent(this, MyCounterService.class);
  startService(intent);
}
```

Il risultato dell'esecuzione del service è dato dal file di log generato:

```
11-09 14:48:19.423:  [MyService] Created
11-09 14:48:19.423:  [MyService] Started
11-09 14:48:19.423:  [MyService] Start counting to 3
11-09 14:48:22.483:  [MyService] Stop counting
11-09 14:48:22.483:  [MyService] Killing myself
11-09 14:48:22.483:  [MyService] Destroyed
```

Nell'esempio proposto si è utilizzato un local service per svolgere del lavoro computazionalmente oneroso. Pur avendo la certenzza di non incorrere in

un ANR, come accennato precedentemente, questa pratica non è corretta poichè durante tutto il periodo d'esecuzione l'interfaccia grafica non sarà reattiva. Le metodologie per la separazione dei flussi di controllo verranno illustrate più avanti in questo capitolo.

2.2.4 Broadcast Receiver

Un *Broadcast Receiver* è una componente che permette di ricevere avvisi propagati a tutto il sistema; il principio su cui si basa è quello dell'*event listener*, secondo il quale gli avvisi sono notificati tramite l'utilizzo di intent. Si intuisce facilmente l'importanza di questa tipologia di oggetti, in quanto rappresentano un'importante via di comunicazione tra applicazioni; in particolar modo, l'utilizzo di tali componenti apre la strada per l'implementazione di una prima forma di *context awareness*. Con il loro uso è possibile, ad esempio, sapere quando l'utente spegne lo schermo, avvisare quando la batteria scende sotto un livello critico, o notificare quando un collegamento alla rete diventa disponibile. L'implementazione di questa componente avviene estendendo la classe *android.content.BroadcastReceiver* e registrandola nell'applicazione attraverso due possibili modalità:

- staticamente, tramite il file *AndroidManifest.xml*; un esempio di definizione è il seguente:

```
<receiver android:name="MyBatteryReceiver">
  <intent-filter>
    <action android:name=
        "android.intent.action.BATTERY_LOW"/>
  </intent-filter>
</receiver>
```

- dinamicamente, tramite il metodo *registerReceiver(BroadcastReceiver receiver, IntentFilter filter)*, al quale occorre fornire, come parametri, il receiver da registrare e un *intent filter* contenente il tipo di evento che si vuole intercettare.

Quando il sistema notificherà un avviso per il quale si è registrati, verrà chiamato il metodo *onReceive(Context context, Intent intent)*, all'interno del quale sarà possibile specificare le operazioni che si desidera effettuare. Bisogna tuttavia prestare attenzione al tipo di operazioni effettuate in

questo contesto, poiché questo metodo *onReceive()* viene eseguito sul *main thread* e, nel caso in cui necessiti di molto tempo per terminare, l'utente potrebbe riscontrare un blocco dell'interfaccia grafica (se visualizzata), o eventualmente una chiusura forzata dell'applicazione da parte di Android in seguito ad un *ANR (Activity Not Responding)*.

Creare un nuovo thread all'interno di *onReceive()* e delegare a lui il lavoro liberando così il main thread non è una soluzione valida: con la terminazione del metodo terminerà anche il ciclo di vita del receiver, conseguentemente Android considererà il lavoro compiuto e lo eliminerà, distruggendo anche l'eventuale thread ancora in esecuzione. L'approccio da seguire è quello di creare un local service al quale affidare il compito di creare un thread; in questo modo, il thread sarà legato ad una componente con un proprio ciclo di vita indipendente dal receiver e non verrà distrutto insieme a quest'ultimo. Infine, per impedire che il dispositivo entri nella modalità *sleep* bloccando le operazione del worker thread è necessario utilizzare un *wake lock* e rilasciarlo solo ad esecuzione terminata.
Per quanto concerne l'invio di una richiesta broadcast, vi sono due possibili modalità:

- Broadcast normale: si invia tramite il metodo *Context.sendBroadcast(Intent intent)*; l'intent è spedito in modo asincrono a tutti i receiver, i quali verranno eseguiti in ordine casuale, e spesso in contemporanea;

- Broadcast ordinato: si invia tramite il metodo *Context.sendOrderedBroadcast(Intent intent)*; l'intent è spedito secondo un ordine di priorità definito dall'attributo *android:priority*. L'intent inviato in questo modo possiede dei dati aggiuntivi che possono essere modificati man mano che vengono gestiti dai receiver e inoltrati ai successivi; inoltre, un receiver può decidere di interrompere la propagazione dell'intent nel caso lo ritenga opportuno.

Per fornire un esempio pratico di utilizzo si aggiungerà all'applicazione considerata sino ad ora un'implementazione di un Broadcast Receiver, capace di monitorare il livello della batteria e reagire quando questo scende sotto il livello critico definito da *Intent.ACTION_BATTERY_LOW*. Tale broadcast viene lanciato dal sistema in automatico ogni volta in cui il livello di carica della batteria scende sotto il 15%.
Per la registrazione si utilizza il metodo statico riportato in precedenza,

mentre l'implementazione della classe *MyBatteryReceiver* viene riportata di seguito:

```
public class MyBatteryReceiver extends BroadcastReceiver {
  private final static String TAG = "BatteryMonitor";

  @Override
  public void onReceive(Context context, Intent intent) {
    Log.i(TAG, "[MyBatteryReceiver] Low battery level!");
    /*Qui è possibile inserire il codice per la gestione
    * dell'applicazione in reazione all'evento */
  }
}
```

Avendo utilizzato la modalità statica per la registrazione del receiver non è necessario aggiungere istruzioni in nessun altra classe dell'applicazione; inoltre, l'oggetto che implementa la classe *MyBatteryReceiver* sarà reattivo anche quando l'applicazione non è in esecuzione.

2.2.5 Content Provider

Il *Content Provider* è il componente Android che si occupa di rendere disponibili i dati alle altre applicazioni installate nel sistema. Ogni applicazione, pertanto, può definire una o più tipologie di dati e rendere poi disponibili tali informazioni esponendo uno o più Content Provider. Prima di poter accedere ad un insieme di dati è necessario conoscere l'URI corrispondente; a questo scopo, Android mette a disposizione diversi tipi di dati pubblici e conseguentemente anche gli URI per accedervi.
Un esempio è costituito dai contatti in rubrica mappati dal seguente URI:

```
content://com.android.contacts/contacts
```

Data la lunghezza di questi indirizzi, è consuetudine salvarli in maniera statica; è possibile trovare l'URI mostrato qui sopra nella classe *android.provider.ContactsContract.Contacts*, accedendo all'attributo *CONTENT_URI*.

Una volta che si conosce l'URI di una tipologia di contenuto, interagire con il provider che la eroga consiste in un'operazione molto simile a quella con la quale si effettuano le interrogazioni di un database. Per prima cosa si deve recuperare un'istanza dell'oggetto *android.content.ContentResolver*; tale oggetto è ottenibile mediante il metodo

android.content.Context.getContentResolver(). Ottenuto tale oggetto è possibile effettuare operazioni di tipo *CRUD (create, retrieve, update, delete)*. I metodi messi a disposizione dalle *API* sono:

- *public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)*
  I parametri da fornire a questo metodo sono i seguenti:
  - *uri* è l'indirizzo che identifica il tipo di contenuto ricercato;
  - *projection* è la lista con i nomi delle colonne i cui valori devono essere inclusi nella risposta. Nel caso in cui *projection* sia *null* vengono restituite tutte le colonne disponibili;
  - *selection* corrisponde in *SQL* alla clausola *WHERE*. Se questo parametro è *null* vengono restituite tutte le righe disponibili;
  - *selectionArgs* è la lista degli argomenti della clausola *WHERE* al punto precedente;
  - *sortOrder* è la clausola SQL di ordinamento. Se questo parametro è *null* non si applica un ordinamento specifico alla lista dei risultati restituiti.

- *public Uri insert(Uri uri, ContentValues values)*
  Questo metodo permette di inserire un nuovo record del tipo specificato; i valori sono mappati nel parametro di tipo *android.content.ContentValues*;

- *public int update(Uri uri, ContentValues values, String where, String[] selectionArgs)*
  Questo metodo permette di aggiornare uno o più record, identificabili tramite i parametri *where* e *selectionArgs*;

- *public int delete(Uri uri, String where, String[] selectionArgs)*
  Questo metodo permette di cancellare uno o più record, identificabili tramite i parametri *where* e *selectionArgs*, analogamente al metodo update.

Il codice seguente rappresenta un esempio di implementazione della ricerca nel sistema di tutti i contatti in rubrica:

```
ContentResolver cr = getContentResolver();
Uri uri = Contacts.CONTENT_URI;
String[] projection = { Contacts._ID,
    Contacts.DISPLAY_NAME };
String selection = null;
String[] selectionArgs = null;
String sortOrder = Contacts.DISPLAY_NAME + " ASC";
Cursor cursor = cr.query(uri, projection, selection,
    selectionArgs, sortOrder);
```

Nel caso in cui volesse rendere disponibile una tipologia di dato differente da quelle nativamente presenti nelle API Android, lo sviluppatore sarebbe libero di definire la propria implementazione della classe *ContentProvider*.

2.3 Threading Model

L'ultimo aspetto fondamentale da conoscere per un corretto sviluppo di applicazioni in ambiente Android è il modo con cui vengono gestiti i thread. Come anticipato durante la trattazione, è spesso utile, e in alcuni casi perfino necessario, ricorrere a particolari astrazioni per separare il flusso di esecuzione di una particolare operazione da quello del resto dell'applicazione, in modo da garantirne un buon livello di reattività e velocità di esecuzione.

2.3.1 Main Thread

Come spiegato nella descrizione dell'ambiente di esecuzione di Android, ad ogni applicazione viene assegnato un processo, il quale contiene originariamente un solo thread, a cui ci si riferisce con il termine *Main Thread* o anche *UI Thread*. Tale thread è l'unico responsabile del flusso d'esecuzione di una applicazione e di tutte le sue componenti. Se non diversamente specificato, il Main Thread è chiamato a gestire:

- l'interfaccia grafica dell'applicazione; questo comporta l'aggiornamento delle componenti visive e la gestione delle interazioni con l'utente;

- l'esecuzione delle operazioni definite all'interno delle componenti dell'applicazione; in particolar modo si occupa della gestione del loro *life cycle* gestendo direttamente la chiamata ai metodi di *callback*.

La comprensione dei limiti che questo modello ad un solo thread comporta è immediata. Dal momento che tutto avviene su un singolo flusso d'esecuzione, nessun evento può esser inviato o gestito dal Main Thread se occupato in altre operazioni; questo porta l'utente ad un'esperienza di utilizzo negativa, poiché l'applicazione può risultare lenta o addirittura bloccata.

Per sopperire a questo problema è dunque necessario ricorrere ad un approccio *multithread*, che permette allo sviluppatore di delegare l'esecuzione di operazioni, particolarmente onerose dal punto di vista della complessità di calcolo, ad altri flussi di controllo; ciò costituisce un importante supporto al Main Thread, permettendo una gestione più tempestiva ed efficiente degli eventi che di volta in volta vengono intercettati e processati. Su questo fronte Android impone una rigida politica di gestione dell'interfaccia; quest'ultima è infatti editabile solo dal Main Thread, e inoltre è stato reso obbligatorio l'utilizzo di alcune *best practices*, definite allo scopo di mantenere il Main Thread libero tramite la segnalazione del già citato *ANR*. Un errore di *Activity Not Responding* altro non è che un sistema di controllo, utilizzato da Android per evitare casi di cattiva programmazione nelle proprie applicazioni; la segnalazione di tale errore avviene quando il Main Thread è tenuto occupato troppo a lungo. Il tempo effettivamente necessario per la segnalazione non è dichiarato con precisione: si aggira intorno ai 5 secondi ma può dipendere dalla componente che sta bloccando il sistema. Ad ogni modo, è chiaro che il dubbio di incorrere in questa evenienza è sintomo di un errore a livello di progettazione dell'applicazione.

Oltre ai classici strumenti per la programmazione multithread presenti nelle librerie Java, Android mette a disposizione una serie di nuove componenti, ideate per facilitare il lavoro dello sviluppatore e per instradarlo verso *pattern* di sviluppo compatibili con i vincoli imposti. La caratteristica più interessante di queste componenti risiede nella facilità con cui è possibile gestire lo scambio di informazioni tra thread, grazie ai particolari meccanismi che implementano. Le componenti principali sono *AsyncTask* e *Handler*; verrà però fornita anche una panoramica sull'utilizzo del classico thread Java, nell'ambito della programmazione Android.

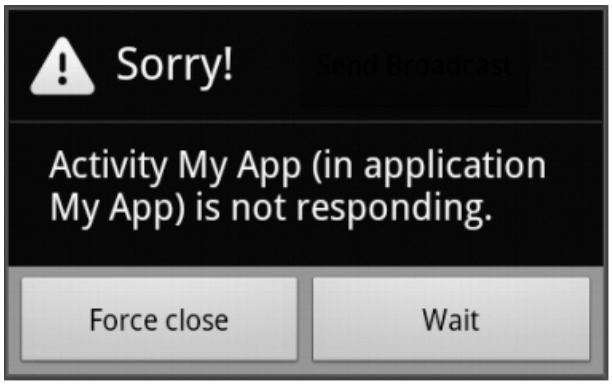

Figura 2.7: Errore ANR

2.3.2 Thread Java

L'utilizzo dei thread definiti nelle librerie Java rappresenta il primo possibile approccio alla programmazione multithread in ambiente Android. Poichè il loro utilizzo nelle applicazioni è limitato allo svolgimento di operazioni non direttamente legate all'interfaccia grafica, prendono il nome di *Worker Threads* o *Background Threads*. I modi per lanciare l'esecuzione di un nuovo worker thread sono:

- tramite l'estensione della classe *java.lang.Thread*;
- tramite l'implementazione dell'interfaccia *java.lang.Runnable*.

Il lancio del nuovo flusso d'esecuzione inizia con la chiamata al metodo *start()*; ad esso segue l'esecuzione del codice contenuto all'interno del metodo *run()*, e al termine di quest'ultimo avviene la chiusura del thread.

L'utilizzo di un worker thread permette di separare efficacemente i flussi di esecuzione; non permette, tuttavia, di poter sfruttare i risultati derivati dalla sua computazione per poter aggiornare l'interfaccia grafica, poiché al tentativo del thread di aggiornare un componente grafico il sistema segnalerà un errore. Per risolvere tale problema vengono messi a disposizione alcuni metodi per poter richiere al main thread, o ad altri thread in generale, di eseguire una serie di istruzioni:

- *Activity.runOnUiThread(Runnable)*;

- *View.post(Runnable)*;
- *View.postDelayed(Runnable, long)*.

Si fornisce un esempio di utilizzo del metodo *post(Runnable)*:

```
public void onClick(View v) {
  new Thread(new Runnable() {
    public void run() {
      //Effettua il download dell'immagine dalla rete
      final Bitmap b =
        loadImageFromNetwork('http://test.com/image.png');
      mImageView.post(new Runnable() {
        public void run() {
          mImageView.setImageBitmap(b);
        }
      });
    }
  }).start();
}
```

Nell'esempio si vuole eseguire all'interno del flusso del Main Thread, che gestisce il metodo *onClick()*, un'operazione potenzialmente lunga come il download di un'immagine, e successivamente visualizzarla sullo schermo non appena questa sia disponibile. La soluzione presentata è corretta, poiché viene delegato ad un nuovo thread l'esecuzione dell'operazione di download, mentre attraverso il metodo *post()* viene effettuato l'aggiornamento dell'interfaccia sfruttando il flusso del Main Thread, così come imposto da Android.

2.3.3 Async Task

L'utilizzo dei worker threads, anche se corretto, non rappresenta un metodo di gestione dei threads adattato per l'utilizzo in ambiente Android, e ciò diventa evidente nel momento in cui si eseguono operazioni più complesse, in cui sono richiesti frequenti aggiornamenti dell'interfaccia. L'alternativa offerta da Android, e creata appositamene per adattarsi ai limiti imposti dal sistema, è l'*Async Task*, il quale facilita la separazione dei flussi di controllo al momento di eseguire operazioni particolarmente lunghe, e implementa metodi per l'aggiornamento delle componenti grafiche.

Per utilizzare un Async Task è necessario estendere la classe *android.os.AsyncTask<Params,Progress,Results >*; i tre *generics* dovranno esser specificati dallo sviluppatore al momento della scrittura della propria classe e rappresentano:

- *Params*: il tipo dei parametri passati al task utili per l'esecuzione;
- *Progress*: il tipo di dato utilizzato per la pubblicazione degli aggiornamenti durante la computazione;
- *Results*: il tipo di risultato restituito al termine della computazione.

Un Async Task è caratterizzato da un proprio life cycle, gestito tramite quattro metodi di callback:

- *onPreExcecute()*: primo metodo ad esser chiamato all'invocazione del task, viene eseguito sfruttando il flusso del Main Thread, e si utilizza per inizializzare le variabili necessarie per le operazione a seguire;
- *doInBackground(Params...)*: invocato immediatamente dopo l'esecuzione di *onPreExcecute()*. Questo metodo è l'unico nel life cycle del task ad esser eseguito su di un thread separato, ed è perciò utilizzato per eseguire operazioni di background o a lungo termine. La computazione eseguita all'interno di questo metodo può sfruttare un insieme di parametri passati in ingresso; il valore di *return* dovrà esser dello stesso tipo di quello specificato alla voce *Results*, e costituirà il parametro in ingresso al metodo *onPostExcecute(Result)*. Se si volesse aggiornare l'interfaccia grafica ad ogni *step* eseguito sarebbe possibile farlo tramite il metodo *publishProgress(Progress..)* la cui chiamata comporta l'esecuzione del metodo *onProgressUpdate(Progress..)*;
- *onProgressUpdate(Progress...)*: eseguito dal main thread, permette di visualizzare sull'interfaccia grafica i progressi ed i risultati intermedi conseguiti dalla computazione all'interno di *doInBackground(Params..)*. Un tipico utilizzo di questo metodo riguarda l'aggiornamento di una *progress* bar durante un download;
- *onPostExcecute(Result)*: invocato al termine di *doInBackground(Params..)*, da cui riceve in ingresso il risultato della sua computazione. Viene utilizzato per rappresentare graficamente il risultato finale della computazione del task.

Una volta istanziato, è possibile lanciare in esecuzione un Async Task tramite il metodo *execute()*, la cui chiamata, che può essere effettuata esclusivamente all'interno del main thread, genererà l'invocazione in sequenza dei primi due metodi sopra descritti. Da ricordare che è possibile utilizzare l'istanza di Async Task una sola volta; le eventuali chiamate ad *execute()*, successive alla prima, genereranno un'eccezione. Un Async Task può terminare la propria esecuzione in due modi:

- esaurendo le istruzioni all'interno del metodo *doInBackground()* e quindi eseguendo *onPostExecute(Result)*;
- attraverso la chiamata al metodo *cancel()*, che comporta un cambiamento dello stato interno dell'oggetto. Se si pensa di utilizzare questo metodo è necessario rivisitare l'implementazione di *doInBackground()*, aggiungendo un controllo periodico allo stato di cancellazione del task tramite il metodo *isCancelled()*, in modo da garantirne la terminazione il prima possibile. Tramite l'utilizzo di *cancel()*, ed una volta terminata l'esecuzione di *doInBackground()*, verrà eseguito dal Main Thread il metodo di callback *onCancelled(Result)*, che sostituirà *onPostExecute(Result)* implementando eventualmente una politica di ripristino dello stato delle componenti grafiche dell'interfaccia.

Per mostrare un utilizzo pratico di Async Task si ripropone l'esempio precedente del download di un'immagine dalla rete; l'invocazione avviene attraverso la gestione dell'evento *onClick()* di un ipotetico pulsante all'interno di una activity:

```
//Eseguito all'interno di una activity
public void onClick(View v) {
  DownloadImageTask task=new DownloadImageTask(this);
  task.execute("http://example.com/image.png");
}
```

La classe *DownloadImageTask*, creata estendendo AsyncTask, è così definita:

```
public class DownloadImageTask
    extends AsyncTask<String, Void, Bitmap> {

  private Context mContext;

  DownloadImageTask(Context context) {
```

```
    mContext = context;
  }

  protected Bitmap doInBackground(String... urls) {
    //Effettua il download dell'immagine dalla rete
    return loadImageFromNetwork(urls[0]);
  }

  protected void onPostExecute(Bitmap result) {
    ImageView mImageView;
    /*Ottiene il riferimento alla view
    * ed imposta l'immagina appena ottenuta */
    mImageView =
        ((Activity) mContext).findViewById(R.id.image);
    mImageView.setImageBitmap(result);
  }
}
```

La classe così definita eseguirà il download mediante il metodo *doInBackground()*, sfruttando quindi un nuovo thread, mentre all'interno di *onPostExecute()*, e quindi attraverso il main thread, avverrà l'aggiornamento della view dell'interfaccia, il cui riferimento è stato ottenuto mediante l'oggetto *Context* rappresentante la activity chiamante e passato al costruttore.

2.3.4 Handler e Looper

Quando si è parlato del Main Thread ci si è concentrati sul suo flusso di esecuzione; tuttavia, è necessario precisare che non è l'unico elemento rilevante per il suo corretto utilizzo. Al concetto di Main Thread, per poterne garantire il funzionamento, è associata anche una *message queue*, che viene utilizzata dal sistema per immagazzinare le richieste in arrivo. Se il Main Thread si trovasse impossibilitato a soddisfare una richiesta perché già occupato nell'esecuzione di un'altra operazione, questa verrebbe posta in coda in attesa del proprio turno. Un *Handler* rappresenta lo strumento attraverso il quale è possibile per lo sviluppatore interagire con la message queue del Main Thread, che è l'unico thread di un'applicazione ad esserne dotato nativamente. Un generico worker thread non nasce con una message queue associata; è possibile assegnarne una ad esso tramite l'utilizzo dei cosidetti *Loopers*, che verranno mostrati più avanti in questa sezione.

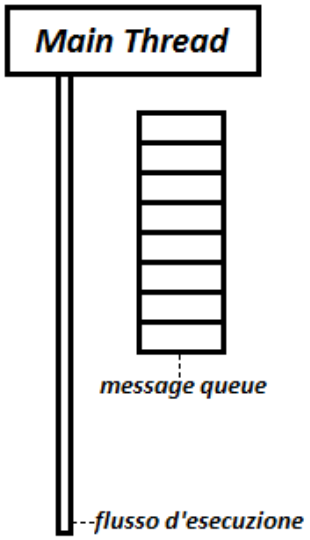

Figura 2.8: Struttura del Main Thread

Per definire il proprio Handler è possibile estendere la classe *android.os.Handler*, ridefinire il metodo *handleMessage(Message msg)* che permette di ricevere messaggi ed infine implementare la politica di gestione di quest'ultimi. Quando un Handler viene istanziato, esso fa sempre riferimento al thread all'interno del quale è stato creato e alla message queue ad esso associata. Si fornisce un esempio di utilizzo in cui si istanzia un handler all'interno del flusso di esecuzione del main thread, associandolo quindi alla sua message queue:

```
MyHandler handler = new MyHandler();
```

Successivamente, in seguito alla pressione di un pulsante, viene lanciato un worker thread il cui compito consiste nel contare fino a 3, ad intervalli di un secondo, e nel notificare il main thread spedendo dei messaggi tramite l'handler ad esso associato:

```
public void startHandlerTest(View view){
  Log.i("HandlerExample", "[CounterActivity]
      Crazione del thread {"+Utils.getThreadId()+"} ");
```

```
  Thread backgroundThread = new Thread(new Runnable() {
    @Override
    public void run() {
    for (int i = 0; i <=3; i++) {
      try {
        Thread.sleep(1000);
        Message msg = handler.obtainMessage();
        Bundle b = new Bundle();
        b.putString("My Key", "" + i);
        msg.setData(b);
        Log.i("HandlerExample", "[BGThread] Invio
            messaggio n°"+i+" {"+Utils.getThreadId()+"} ");

        //Invio di un messaggio al main thread
        handler.sendMessage(msg);
      } catch (Exception e) {}
    }
    Log.i("HandlerExample", "[BGThread] Conteggio finito
        {"+Utils.getThreadId()+"} ");
    }
  });
  backgroundThread.start();
}
```

Come si può vedere, l'invio del messaggio avviene tramite il metodo *sendMessage(Message msg)*, e i dati all'interno del messaggio vengono inseriti tramite *setData(Bundle data)* utilizzando l'oggetto *Bundle* descritto in precedenza. E' importante notare come l'oggetto *Message* non sia ottenuto creandone uno nuovo, bensì venga ottenuto tramite il metodo *handler.obtainMessage()*, che comporta la restituzione di un riferimento ad un messaggio presente in un *message pool* associato allo stesso handler. In questo modo il sistema può 'riciclare' gli stessi messaggi, ottimizzando la gestione delle risorse. Una volta inviato il messaggio, questo verrà depositato nella message queue del Main Thread e gestito dal metodo *handleMessage(Message msg)* della classe *MyHandler*, così definita:

```
public class MyHandler extends Handler {
  private final static String TAG = "HandlerExample";
  /*Eseguito dal main thread nel momento in cui
  * non è occupato */
```

```
  @Override
  public void handleMessage(Message msg) {
    Bundle b = msg.getData();
    String val = b.getString("My Key");
    Log.i(TAG, "[MyHandler] Valore nel messaggio : "+val+"
        {"+Utils.getThreadId()+"}");
  }
}
```

All'interno del metodo *handleMessage()* ci si limita a leggere il contenuto del messaggio e stamparlo tramite log; l'esecuzione di questo *snippet* comporta il seguente risultato:

```
38.298: [CounterActivity] Crazione del thread {main:(id)1}
39.363: [BGThread] Invio messaggio n°0 {Thread-10:(id)10}
39.368: [MyHandler] Valore nel messaggio : 0 {main:(id)1}
40.371: [BGThread] Invio messaggio n°1 {Thread-10:(id)10}
40.371: [MyHandler] Valore nel messaggio : 1 {main:(id)1}
41.391: [BGThread] Invio messaggio n°2 {Thread-10:(id)10}
41.391: [MyHandler] Valore nel messaggio : 2 {main:(id)1}
42.408: [BGThread] Invio messaggio n°3 {Thread-10:(id)10}
42.408: [MyHandler] Valore nel messaggio : 3 {main:(id)1}
```

Come è possibile notare, la gestione dei messaggi da parte di *MyHandler* avviene all'interno del Main Thread; questo significa che, nel medesimo contesto, sarebbe ugualmente possibile effettuare modiche anche all'interfaccia grafica. Oltre al tipo *Message*, è possibile accodare nella message queue anche oggetti di tipo *Runnable* che, una volta giunto il loro turno, saranno automaticamente posti in esecuzione all'interno del thread al quale la coda è associata.

Per dare la possibilità anche ad un generico thread di avere una message queue associata e di poter ricevere messaggi asincroni oltre che inviarli, si sfruttano le funzionalità della classe *android.os.Looper*. Tale classe permette in modo molto semplice di creare ed associare una message queue a qualunque thread, mediante l'utilizzo dei seguenti metodi:

- *public static void Looper.prepare()*: serve a creare una message queue e ad associarla al thread all'interno del quale è chiamato;
- *public static void Looper.loop()*: finalizza il processo di creazione della coda, rendendola di fatto pronta a ricevere messaggi;

- *public void quit()*: utilizzato per terminare il *Looper Thread*, ovvero il thread al quale è associata una message queue. Tale metodo non è statico e deve esser chiamato sull'oggetto Looper associato al flusso d'esecuzione ottenibile tramite *Looper.myLooper()*.

Occorre richiamare tali metodi all'interno del metodo *run()* del nuovo thread; tra la chiamata a *prapare()* e quella a *loop()* è necessario istanziare l'handler che si occuperà di gestire i messaggi inseriti nella message queue.

Il *design pattern* implementato grazie all'uso della classe *Looper* possiede una caratteristica particolarmente degna di nota: la business logic del thread viene di fatto spostata dal metodo *run()* del thread al metodo *handleMessage()* dell'handler ad esso associato; ciò trasforma l'handler stesso in una specifica entità con un proprio flusso di esecuzione, un personale *asyncronous event listener* e un sistema di gestione a pila dei messaggi ricevuti.

Capitolo 3

Caso di studio: SMS Backup+

In questo capitolo si andrà ad effettuare una analisi critica del modello di programmazione di Android, al fine di verificare in maniera pratica il supporto che esso fornisce per lo sviluppo di *smart mobile applications*, ovvero applicazioni che implementano il livello di *context awareness* descritto nel capitolo introduttivo della presente Tesi. A tal scopo è fondamentale immergersi nel contesto di un caso di studio reale, relativo ad una applicazione usata su larga scala. Per conseguire tale obiettivo, è stato scelto un prodotto consolidato sul mercato, con oltre un milione di downloads all'attivo, evitando in questo modo di cadere nella trappola comune offerta da un semplice studio di una applicazione di carattere accademico, priva di alcun interesse pratico.

L'applicazione *SMS Backup+* nasce dalle ceneri del precedente progetto *SMS Backup*, il cui sviluppo è stato interrotto nei primi mesi del 2010. La principale funzione che offre questo prodotto consiste nella possibilità di salvare gli SMS, presenti sul proprio smartphone, direttamente sulla casella di posta personale di *Gmail*, organizzandoli in conversazioni separate a seconda del destinatario. Un ulteriore vantaggio che offre *SMS Backup+* consiste nella possibilità per l'utente di richiedere manualmente il salvataggio oppure di schedularlo automaticamente, secondo intervalli di tempo regolari e di periodo variabile. La versione presa come caso di studio è la 1.4.6; quest'ultima, oltre alle funzionalità appena descritte, offre anche:

- Il ripristino degli sms dalla casella di posta al proprio dispositivo;
- Il salvataggio degli mms;

- Il salvataggio, e corrispondente ripristino, del registro delle chiamate, grazie all'integrazione del servizio *Google Calendar*;
- La possibilità di utilizzare altri server di posta IMAP differenti da Gmail;
- La possibilità, per le altre applicazioni, di chiedere a loro volta il backup di sms, mms e del registro delle chiamate.

3.1 Struttura del progetto

Come suggerito dalle *best practices* proposte in ambiente Android, il progetto dell'applicazione segue un preciso schema strutturale; osservando il *package explorer* di *Eclipse*, come mostrato in figura 3.1, risultano evidenti le componenti più importanti del progetto, all'interno della cartella principale:

- */src*: contiene i codici sorgenti del progetto; tutte le classi sono racchiuse all'interno dell'unico *package com.zegoggles.smssync*;
- */gen*: contiene la classe autogenerata *R.java*, che associa al nome di ogni risorsa dell'applicazione un Id univoco;
- */res*: contiene tutte le risorse statiche dell'applicazione, quali immagini, stringhe o layout definiti tramite xml e divisi nelle rispettive sottocartelle;
- *AndroidManifest.xml*: file contenente alcune *meta-informations*, riguardanti l'applicazione e le sue componenti.

Il progetto, dal punto di vista dello sviluppatore, è purtroppo molto carente in quanto a documentazione tecnica. L'architettura di default di un'applicazione Android, tuttavia, risulta un più che valido aiuto al processo di analisi al quale il progetto verrà sottoposto nei capitoli seguenti. Ad esempio, il file *AndroidManifest.xml* fornisce un chiaro e organizzato elenco delle componenti coinvolte nel funzionamento dell'applicazione, facilitando notevolmente l'identificazione dei ruoli di ciascun elemento all'interno del progetto.

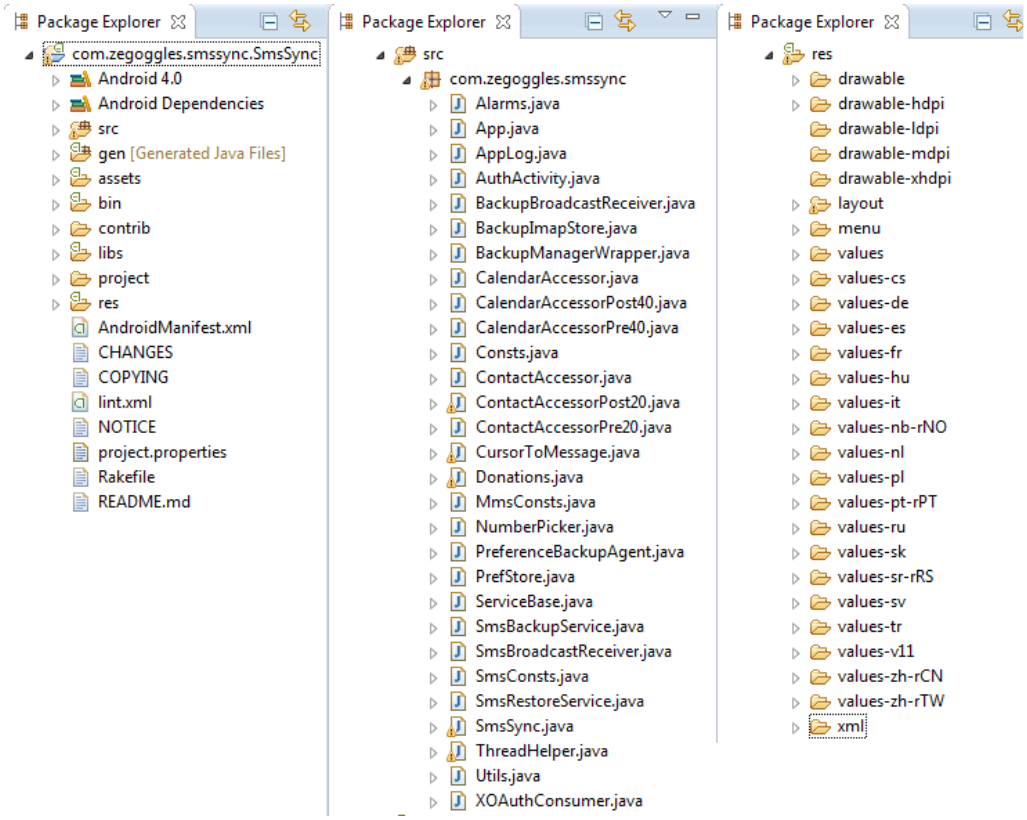

Figura 3.1: Struttura del progetto e dettaglio di /src e /res

3.2 Utilizzo dell'applicazione

Le operazioni che un utente può effettuare sono identificabili in tre categorie principali:

- operazione di *backup*;
- operazione di *restore*;
- impostazione delle preferenze.

3.2.1 Backup

L'operazione di backup costituisce prevedibilmente la funzionalità principale dell'applicazione. Tramite questa funzione, l'utente può salvare gli sms, gli mms e il registro delle chiamate, presenti sul proprio smartphone, direttamente sulla propria casella di posta. Ciò permette di mantenere uno storico completo senza dover occupare inutilmente la memoria del dispositivo e, ancora più importante, garantisce la disponibilità costante di una copia di backup, in caso di malfunzionamento del dispositivo stesso.
Una richiesta di backup può avvenire in diversi modi:

- Può essere richiesta direttamente dall'utente, tramite pressione dell'apposito pulsante sull'interfaccia grafica;
- Può essere avviata automaticamente dall'applicazione, anche quando questa non è attiva, grazie alla funzione di backup periodico;
- Può avvenire in seguito alla ricezione di un sms;
- Può essere richiesta da una applicazione sviluppata da terze parti, tramite la diffusione di un apposito messaggio in *broadcast*.

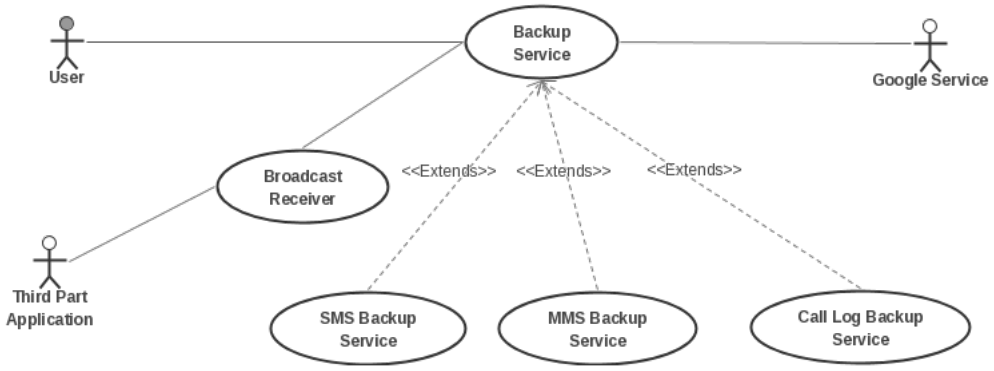

Figura 3.2: Use Case Diagram dell'operazione di Backup

3.2.2 Restore

Dualmente alla funzionalità di Backup, l'opzione di Restore permette all'utente di ripristinare sul proprio dispositivo gli sms ed il registro chiamate precedentemente memorizzati mediante Backup. E' da notare come, nella versione dell'applicazione attualmente disponibile, non sia possibile ripristinare gli mms salvati in precedenza. A differenza del Backup, l'operazione di Restore può essere avviata esclusivamente dall'utente, selezionando la relativa opzione presente nell'interfaccia.

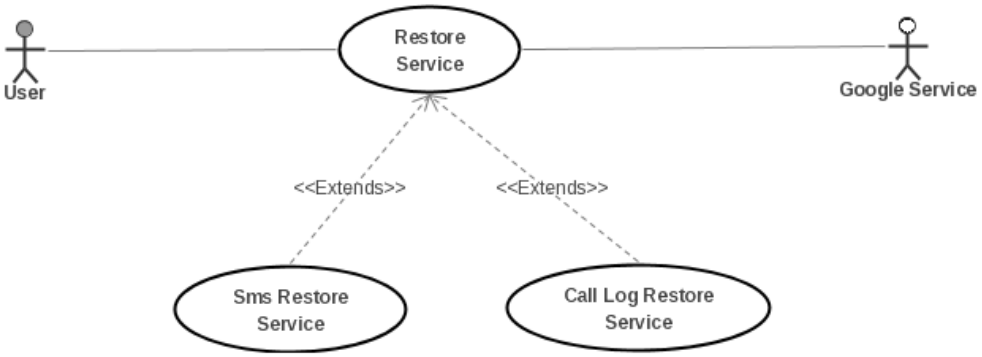

Figura 3.3: Use Case Diagram dell'operazione di Restore

3.2.3 Modifica Impostazioni

L'ultima, ma non meno importante, funzionalità offerta dall'applicativo consiste nell'interfaccia di configurazione del prodotto. Grazie a questo set di opzioni, l'utente può definire il comportamento delle operazioni di Backup e Restore, oltre a poter gestire i parametri di connessione e a poter decidere se tener traccia dei flussi di esecuzione tramite il log applicativo.

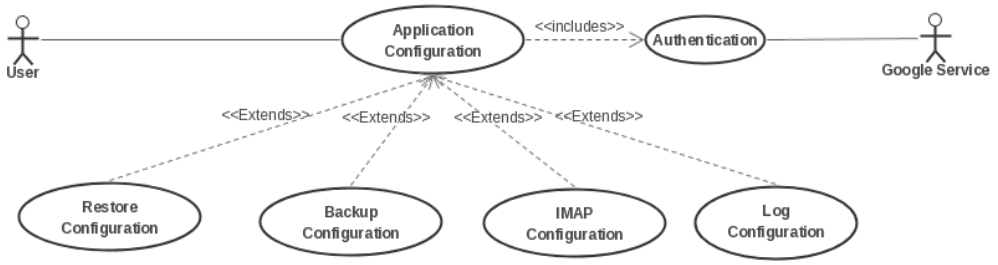

Figura 3.4: Use Case Diagram dell'operazione di configurazione

Occorre sottolineare come, in tutte le operazioni messe a disposizione dall'applicazione, giochi un ruolo fondamentale l'*Actor* denominato *Google Service*. Questo servizio, offerto direttamente da Google tramite *API* ben definite e consolidate, permette un'interazione piuttosto semplice e diretta con il *Web Service REST* che gestisce l'accesso e la manipolazione dell'account Gmail dell'utente.

3.3 Analisi della funzione di Backup

Come già specificato, la funzionalità di backup offerta dall'applicazione *SMS Backup+* ne costituisce il cuore, oltre ad essere la parte più complessa dell'intero progetto, sia dal punto di vista architetturale, per quanto concerne le classi e le risorse, sia dal punto di vista concettuale e operativo, poiché in essa vengono svolte operazioni articolate e vengono gestiti numerosi eventi. Data la sua complessità, questa operazione contiene tutti gli elementi d'interesse per questo lavoro di analisi, si focalizzarà quindi l'attenzione su questa funzione.

3.3.1 Analisi della struttura

La logica della operazione di Backup è definita interamente all'interno della classe *SmsBackupService.java*; tuttavia, attorno a quest'ultima orbitano numerose altre classi (mostrate in figura 3.5 nel class diagram corrispondente), con le quali interagisce costantemente. Occorre, pertanto, fare chiarezza sull'organizzazione e le funzionalità da esse offerte durante l'esecuzione di un'operazione di backup.

- **SmsSync**: rappresenta la *activity* principale dell'applicazione; in essa riscontriamo una differenza fondamentale, rispetto alla *best practice* prevista dal manuale Android, e illustrata nel capitolo precedente: a questa activity corrispondono direttamente tutte le schermate a disposizione per l'utilizzo dell'applicazione. La singolarità della scelta effettuata nella progettazione di *SMS Backup+* è giustificabile dalla particolare struttura dell'applicazione, che consiste, in sostanza, in un unico menu di preferenze organizzato su livelli differenti; in un contesto di sviluppo Android, questa situazione è perfettamente gestibile tramite una specifica tipologia di activity, definita dalla classe *android.preference.PreferenceActivity*, dalla quale *SmsSync* discende. Durante l'operazione di backup, questa activity si occupa di aggiornare la visualizzazione dello status dell'operazione, e di gestire gli eventi derivanti dalla pressione del pulsante di avvio o di arresto da parte dell'utente.

- **PrefStore**: rappresenta un'entità di supporto per l'accesso ai dati permanenti dell'applicazione, salvati tramite la classe *android.content.SharedPreferences*, che permette la persistenza dei dati all'interno di un *hashmap* predefinito. Un esempio di coppia chiave-valore definita in questo contesto è rappresentato dalla definizione della data di salvataggio dell'ultimo sms elaborato in fase di backup.

- **Alarms**: Classe che costituisce un'interfaccia per l'applicazione verso il sistema di gestione delle notifiche *Alarm Manager*, interno al sistema operativo. Questa entità, costituita interamente da metodi e proprietà statiche, permette all'applicazione di impostare i *timer* di esecuzione delle attività schedulate, programmandole tramite le API fornite dall'Alarm Manager. La classe *Alarms* fornisce le basi per una distinzione netta fra le differenti fonti dalle quali provengono le

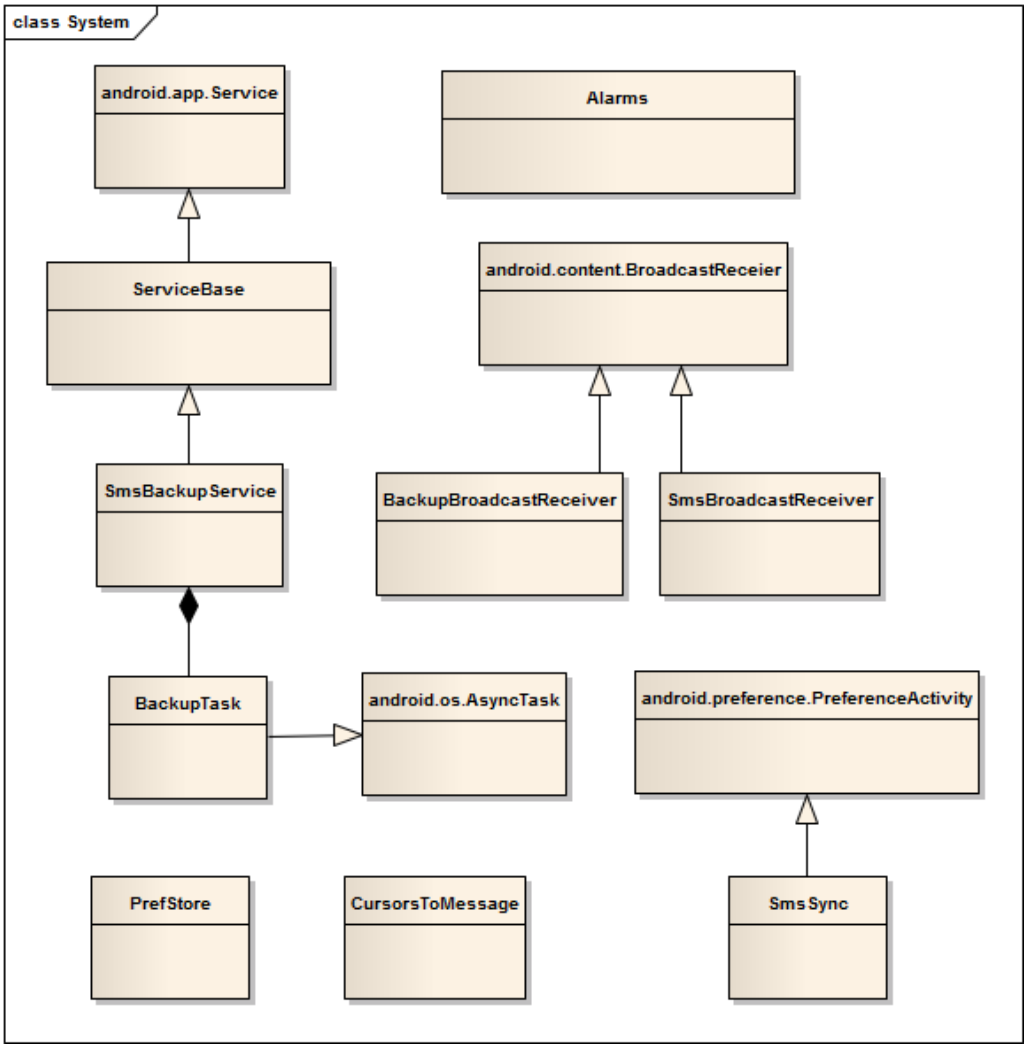

Figura 3.5: Class Diagram delle principali classi utilizzate in un'operazione di backup

richieste di programmazione, quali ad esempio l'arrivo di un sms o la schedulazione manuale da parte dell'utente.

- **BackupBroadcastReceiver/SmsBroadcastReceiver**: Queste classi, che ereditano entrambe dalla superclasse *BroadcastReceiver*, si occupano di intercettare rispettivamente l'evento di richiesta di Backup dagli applicativi di terze parti, e l'evento di ricezione degli Sms da parte del dispositivo. L'esecuzione del backup non è, tuttavia, eseguita automaticamente: la notifica della richiesta viene veicolata verso la classe *Alarms*, per verificare che, secondo le impostazioni selezionate dall'utente, sia permesso effettuare realmente l'operazione, e con quali tempistiche.

- **SmsBackupService**: Classe chiave dell'applicazione, è l'entità che si occupa della corretta esecuzione delle operazioni di Backup; in particolare, questo servizio viene avviato quando c'è la richiesta, da parte dell'utente o della classe *Alarms*, di effettuare l'operazione sopracitata. *SmsBackupService* effettua un controllo sullo stato del sistema operativo: se il sistema è in stato di *Sleep*, e non è abilitata la trasmissione dati in background, la richiesta non verrà processata; in caso contrario, il servizio verificherà l'assenza di un backup o di un restore in esecuzione, e infine avvierà un'istanza della classe *BackupTask*, la quale si occuperà dell'operazione di backup.

- **BackupTask**: Questa classe, che sfrutta le funzionalità di *AsyncTask* per la definizione di un nuovo flusso di esecuzione parallelo al Main Thread, si occupa di preparare gli elementi dei quali effettuare il backup, e successivamente di eseguire materialmente il backup stesso. Durante la fase di preparazione, *BackupTask* si occupa di definire quali sono gli elementi da memorizzare (sms, mms, e registro chiamate) in base ai parametri di configurazione dell'applicazione; nella fase successiva verrà effettuata la memorizzazione sul repository esterno scelto nelle impostazioni. Questa classe si occupa anche di un'attività non meno importante: gestisce le interruzioni derivanti da errori o richieste esplicite del sistema o dell'utente, e agisce di conseguenza garantendo l'integrità dei dati.

- **CursorsToMessage**: La prima funzione di questa classe è quella di trasformare ognuna delle differenti entità gestite dall'applicazione, in-

capsulandole una alla volta in un unico tipo di oggetto, il *Message*. Lo sviluppatore ha in realtà predisposto un parametro che permette di incapsulare più oggetti alla volta in un singolo Message; l'utilizzo di questa opzione è tuttavia sconsigliata da un breve commento nel codice, in cui si avverte che potrebbe causare problemi. Per ovvi motivi questa possibilità è nascosta all'utilizzatore finale dell'applicazione, il quale non ha la possibilità di modificare questa impostazione tramite interfaccia grafica. La mancanza di documentazione sull'argomento lascia intendere che tale opzione sia stata introdotta internamente a scopi di debug, oppure che sia stata predisposta in anticipo per una futura implementazione definitiva.

La descrizione appena fornita delle classi principali dell'applicazione consente di osservare facilmente come alcuni dei concetti di buona programmazione, illustrati nel capitolo precedente, siano stati applicati con successo in questo progetto:

- l'operazione di backup vera e propria, che può richiedere anche diversi minuti a seconda della quantità di oggetti da salvare, viene effettuata su di un nuovo thread, grazie alle funzionalità di AsyncTask;

- la generazione di una *istanza* di *BackupTask* avviene all'interno di un service; ciò garantisce all'istanza stessa di poter terminare il suo lavoro a prescindere dalla chiusura dell'applicazione, poiché esso è legato ad una componente dotata di un ciclo di vita indipendente;

- i due oggetti che ereditano da BroadcastReceiver, vengono utilizzati esclusivamente per notificare l'applicazione del relativo evento sul quale sono in ascolto, garantendo una veloce esecuzione del metodo *onReceive()*; ciò si traduce in una rapida terminazione del componente e un'immediata restituzione del controllo;

- la activity *SmsSync* si occupa esclusivamente dell'aggiornamento dell'interfaccia grafica, sfruttando le altre classi per qualsiasi tipo di computazione; questo pattern di programmazione, seppur ignorato per motivi di comodità negli esempi dello stesso manuale di Android [7], viene comunque sottolineato più volte al fine di promuovere una politica di separazione dei concetti.

La semplice applicazione di queste *best practices* permette allo sviluppatore di ottenere con uno sforzo minimo un'applicazione stabile, reattiva e concettualmente ben organizzata.

3.3.2 Analisi dell'operazione di Backup

Esaurita la descrizione architetturale degli elementi interessati all'operazione di backup, si esaminano ora, in modo più dettagliato, i passaggi che ne caratterizzano l'esecuzione. Per farlo ci si avvale di uno dei *sequence diagrams* elaborati a conclusione del processo di *reverse engineering*, effettuato in sede di analisi dell'applicazione e reso necessario dalla mancanza di documentazione ufficiale.

Parallelamente alla descrizione oggettiva dei processi, saranno esaminate le scelte implementative intraprese dagli sviluppatori, allo scopo di valutare quanto tali scelte risultino conformi al modello di programmazione di Android, o al contrario se ne discostino, e per quali motivazioni.
Qui di seguito, una breve descrizione delle entità che risultano coinvolte nel processo di Backup, indicate graficamente nel *sequence diagram* di figura 3.6:

- *SMS Backup*: rappresenta l'applicazione nella sua interezza, senza effettuare distinzione tra le sue componenti interne;

- *Android System*: rappresenta il sistema operativo Android; è necessario rappresentarlo per descrivere le interazioni che l'applicazione ha con esso durante il recupero di alcune informazioni di sistema;

- *k9-imapstore*: rappresenta l'omonima libreria estrapolata dal progetto del *client* di posta *K-9 Mail*, che fornisce un set di funzioni per una interazione facilitata con i server di posta *IMAP*. Il tipo di dato *Message* illustrato precedentemente è definito all'interno di questa libreria, e rappresenta il generico messaggio che viaggerà attraverso la rete, formattato secondo quanto richiesto dallo standard IMAP;

- *Google Service*: rappresenta il server di posta IMAP offerto dal servizio *Gmail*. Occorre sottolineare che, in tutti i casi illustrati fino ad ora, è stato sempre preso in considerazione l'utilizzo dei servizi di posta di Google, in quanto costituiscono la principale via d'utilizzo

dell'applicazione; si rammenta che è comunque possibile utilizzare altri server di posta, poiché questa scelta non comporterebbe significativi cambiamenti nei diagrammi presentati.

Backup request

E' intuitivo notare dal sequence diagram riportato in figura 3.6 che una generica operazione di backup comincia con una *Backup request*; tale richiesta, come spiegato in precedenza, può derivare da fonti di diversa natura, ma indipendentemente dalla sua origine essa verrà sempre effettuata tramite l'invio di un *Explicit Intent* al sistema, all'interno del quale si richiede l'avvio di *SmsBackupService*. La creazione ed invio dell'intent avrà, in ogni caso, la seguente struttura:

```
Intent intent = new Intent(this, SmsBackupService.class);
startService(intent);
```

Come conseguenza di questa operazione, il sistema avvierà un'istanza di *SmsBackupService*, eseguendo il metodo di callback *onStartCommand()*. Occorre rimarcare come lo sviluppatore sia stato costretto, per mantere la compatibilità anche con le versioni precedenti ad *Eclair* (api level 5), a fornire una implementazione del metodo deprecato *onStart()*; ciò avviene convogliando l'esecuzione dei due metodi attraverso l'utilizzo di *handleIntent(final Intent intent)*, definito dallo sviluppatore stesso, coerentemente a quanto suggerito nelle API di Android.
Successivamente viene effettuato un controllo per verificare che non vi siano in corso altre operazioni di backup o restore; di seguito si riporta il codice utilizzato per svolgere tale controllo:

```
if (!sIsRunning) {
  if (!SmsRestoreService.isWorking()){
    sIsRunning = true;
    new BackupTask().execute(intent);
  } else {
  appLog(R.string.app_log_skip_backup_already_running);
  }
}
```

La soluzione adottata, il cui uso è largamente diffuso fra i progettisti Android, prevede l'utilizzo delle variabili statiche *sIsRunning* e del metodo

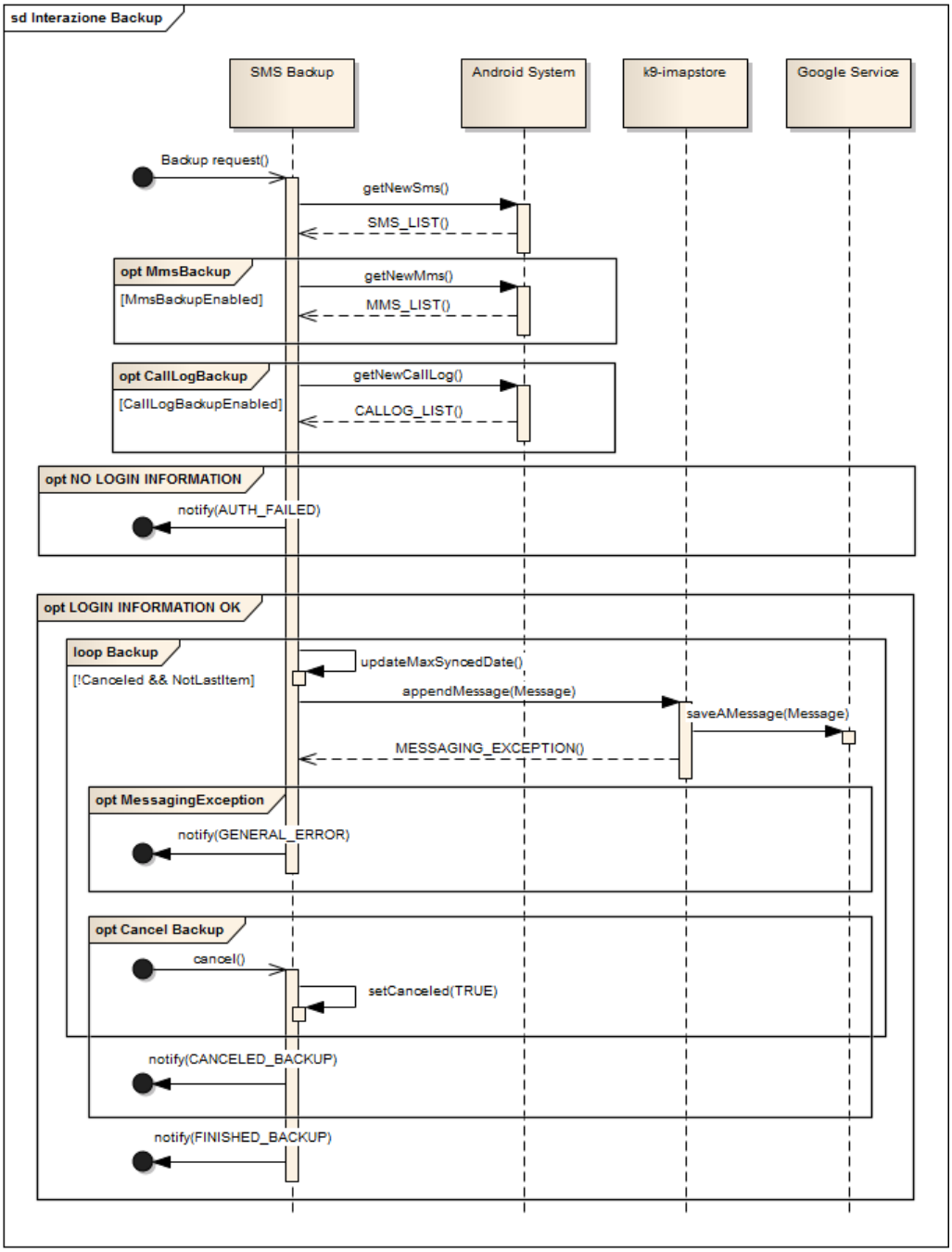

Figura 3.6: Sequence diagram dell'operazione di backup

statico *isWorking()* di *SmsRestoreService*; tale metodo, coerentemente con la soluzione scelta, ritorna semplicemente il valore di una variabile statica definita all'interno della classe. Nel caso in cui non vi siano altri task attivi, verrà creata ed eseguita una nuova istanza di *BackupTask*. Questa soluzione ha, tuttavia, un difetto fondamentale, che ne mina irreparabilmente l'affidabilità: nel caso di una interruzione non gestita dei processi di backup e restore, la variabile *slsRunning* risulterebbe non aggiornata correttamente, con il risultato di mantenere 'appesa' l'immagine di esecuzione del thread e di impedire ad ulteriori task di essere eseguiti, pur avendone il diritto in termini di funzionamento logico dell'applicazione. In questo caso, solamente un riavvio dell'applicazione o una modifica forzosa della variabile potrebbe risolvere il problema e riportare il flusso di esecuzione allo stato corretto.

Recupero Sms, Mms e Call logs

All'interno del metodo *doInBackground()* di *BackupTask* viene effettuata l'operazione di recupero degli sms, mms e registro delle chiamate da salvare, sfruttando il flusso di esecuzione di un thread separato dal Main Thread. L'utente ha la possibilità di scegliere quali categorie di oggetti (sms, mms, registro chiamate) includere nell'operazione di backup; all'interno di *SharedPreferences* è memorizzata questa scelta. Una semplice analisi del codice ha permesso di rivelare un *bug* dell'applicazione, esplicitato nel *sequence diagram* di figura 3.6: il flusso di controllo sulla scelta degli oggetti sui quali effettuare il backup non include gli sms, sebbene esista l'opzione per l'utente di escluderli del tutto dall'operazione di salvataggio; in tal senso, l'opzione verrà ignorata e il software includerà in ogni caso gli sms nel backup.
Il recupero degli elementi da salvare avviene interrogando i rispettivi *Content Provider*, utilizzando il metodo *query()* di oggetti *ContentResolver*. Di seguito è riportato il codice adibito al recupero degli sms:

```
String sortOrder = SmsConsts.DATE;
  return getContentResolver().query(SMS_PROVIDER, null,
    String.format("%s > ' AND %s <> ' %s", SmsConsts.DATE,
      SmsConsts.TYPE, groupSelection(DataType.SMS, group)),
      new String[] {
        String.valueOf(
          PrefStore.getMaxSyncedDateSms(context)),
        String.valueOf(SmsConsts.MESSAGE_TYPE_DRAFT) },
    sortOrder);
```

Traducendo la query in linguaggio SQL, essa diventa:

```
SELECT *
FROM SMS
WHERE DataSms > MaxSyncData AND type <> 3
ORDER BY  Date
```

dove 'type <>3' indica di tralasciare i messaggi salvati come bozze. Si intuisce che l'applicazione utilizza un campo data per discriminare quali sms siano stati già precedentemente salvati e quali invece siano ancora da memorizzare. L'utilizzo di questo semplicissimo stratagemma è reso necessario dal fatto che Android non prevede nativamente operazioni di backup per questi tipi di dati, e di conseguenza, all'interno del relativo Content Provider, non esiste un campo utilizzabile come *flag* per questo tipo di operazione, analogo a quello utilizzato per marcare i messaggi letti.

Recuperati tutti gli oggetti di cui effettuare il backup, si effettua un conteggio e si salva il numero complessivo all'interno della variabile statica *sItemsToSync*, che in seguito verrà utilizzata come condizione di uscita dal ciclo di backup.

L'esecuzione di queste operazioni all'interno di un nuovo thread sono da considerarsi un'ottima pratica, in quanto possono rivelarsi piuttosto lunghe, e come più volte ribadito è consigliabile evitare di bloccare il Main Thread anche se non si pensa di incorrere in un *ANR*.

Notify

Con la voce *Notify* si intende il metodo generico con il quale l'applicazione avvisa l'utente di un particolare evento all'interno dell'esecuzione del backup, come ad esempio il completamento dell'operazione, oppure un generico errore. Coerentemente a quanto detto nella descrizione degli AsyncTask è molto semplice propagare informazioni all'interfaccia grafica, nonostante l'obbligo imposto da Android di effettuare determinate modifiche all'interno del Main Thread, attraverso l'uso del metodo *publishProgress()* e del metodo di callback *onProgressUpdate()*.

Mentre la comunicazione da un thread all'altro è incredibilmente semplificata, la vera difficoltà è data dall'ottenere un riferimento alla activity *SmsSync* in modo da poterne aggiornare gli elementi grafici e notificare così l'utente. Poiché buona parte delle richieste di backup che arrivano non provengono direttamente dall'interfaccia, risulta evidente come sia difficol-

toso mantenere il riferimento diretto alla activity che si vuole aggiornare. Il presentarsi di differenti problematiche di questo genere, insieme alla sorprendente carenza di indicazioni, da parte della manualistica ufficiale, sulle *best practices* da adottare per risolvere tali problematiche, ha provocato nel corso del tempo l'elaborazione da parte degli sviluppatori di soluzioni del tutto varie, e spesso addirittura fantasiose o molto articolate. Si consideri per esempio l'arrivo di un sms: tutto il processo di gestione dell'evento e richiesta del backup avviene senza mai passare dalla activity *SmsSync*; in questa situazione, senza riferimento alla *view* da aggiornare, è necessario adottare una soluzione *custom*, quale ad esempio l'utilizzo di metodi o attributi statici. Qui di seguito si riporta la soluzione adottata all'interno dell'applicazione:

```
@Override
protected void onProgressUpdate(SmsSyncState... progress) {
  if (progress != null && progress.length > 0) {
    if (smsSync != null) {
      smsSync.statusPref.stateChanged(progress[0]);
    }
    sState = progress[0];
  }
}
```

Dove *smsSync* è una variabile statica, che la activity stessa si occupa di impostare come segue, al momento della sua creazione:

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  ServiceBase.smsSync = this;
  . . .
}
```

Questo tipo di soluzione, che appare piuttosto semplice ed efficace, è in realtà da evitare; il riferimento alla activity, salvato in una variabile statica, impedirà al *Garbage Collector* di effettuare l'operazione di rimozione per liberare la memoria di *heap*, nel caso in cui la activity venga distrutta. L'errore è aggravato proprio dal fatto che si tratta di una activity, il cui riferimento viene in genere passato alle *views* che la compongono, oltre che ad ulteriori componenti, generando quindi una gerarchia di oggetti che il Garbage Collector non può ripulire dalla memoria.

L'esempio appena riportato dimostra chiaramente come, se da un lato esistono numerose soluzioni di diverso genere al problema sottoposto, d'altra parte queste soluzioni non siano sponsorizzate ufficialmente dalle specifiche di programmazione in ambiente Android, poiché di fatto manca un supporto tecnico nelle API, che permetta di scegliere una delle soluzioni e implementarla senza incorrere in *side effects*, che possono risultare particolarmente ostili alla buona realizzazione del prodotto.
Ad integrazione di quanto detto finora, si riportano alcune altre soluzioni specifiche che vengono comunemente utilizzate nel mondo della programmazione Android:

- Utilizzare un sistema di comunicazione organizzato in *Intents*; tale soluzione è in linea con il paradigma di utilizzo delle logiche di *broadcasting*, che risulta particolarmente efficace ed è ottimamente gestito dalle API di Android. L'implementazione, in un progetto mediamente complesso, di un sistema così articolato di generazione di messaggi è tuttavia difficile da organizzare e gestire, poiché all'aumentare del codice scritto e delle funzionalità richieste, l'incremento di messaggi broadcast generati risulterebbe esponenziale, rendendo estremamente complessi e difficoltosi il debug, lo studio di nuove features e il refactoring di quelle vecchie.

- Dichiarare un metodo statico all'interno di *SmsSync* ed utilizzarlo per richiamare il metodo *stateChanged* dell'oggetto *statusPref*. Questa soluzione risulta decisamente più semplice e intuitiva della prima, e comporta una gestione più snella nel caso di un progetto di medie o grandi dimensioni; tuttavia, la dichiarazione di istanze di oggetti esterni all'interno di metodi dell'applicazione va contro le buone regole di programmazione, legando inesorabilmente e rendendo dipendenti fra loro le classi interessate. Contrariamente a pattern quali la *dependency injection* e l'uso di interfacce e di metodi generici, una soluzione come quella proposta renderebbe molto difficoltoso il *refactoring*, e pressoché impossibile la riusabilità del codice all'interno dell'applicazione stessa.

Backup Loop

Questo ciclo costituisce il cuore dell'operazione di salvataggio; al suo interno avviene l'invio effettivo, tramite connessione internet, dei messaggi da salvare sul server IMAP. L'esecuzione di tale ciclo si divide nelle seguenti fasi:

1. Conversione di un *sms*, *mms* o *call log* nel più generico *Message* specificato dalla libreria *k9-imapstore*; tale conversione avviene tramite l'utilizzo del metodo *cursorToMessage()* dell'omonima classe, che si occupa di rimappare le informazioni contenute, ad esempio in un sms, all'interno di un oggetto Message.

2. Aggiornamento della classe *PrefStore* tramite i metodi statici *setMaxSyncedDateSms()*, *setMaxSyncedDateMms()*, *setMaxSyncedDateCallLog()*; grazie ad essi, l'applicazione tiene traccia in modo persistente della data di ricezione o di invio dell'ultimo oggetto salvato. L'operazione avviene per ogni categoria, ovvero verrà memorizzata una data per l'ultimo sms, una per l'ultimo mms ed un'altra per l'ultimo call log. Tale tecnica, combinata con una strategia di backup che prevede il salvataggio partendo dagli elementi più vecchi e terminando con i più recenti, permette di conoscere con sufficiente precisione quali elementi sono stati salvati fino a questa fase, e fornisce inoltre, come si vedrà in seguito, un buon livello di robustezza nella gestione di interruzione del processo.

3. Invio effettivo del Message al server IMAP; questa operazione è avviata tramite il metodo *appendMessages()* della libreria k9-imapstore, ed è completamente gestita da quest'ultima. L'aspetto importante è la possibilità durante questa fase che venga generata l'eccezione *MessagingException*, anch'essa definita all'interno della libreria esterna, che rappresenta un errore generico avvenuto durante la trasmissione, come ad esempio una caduta della connessione.

4. Aggiornamento, all'interno della variabile statica *sCurrentSyncedItems*, del contatore degli elementi sino ad ora salvati, e notifica all'utente dello stato di progresso dell'operazione mediante le metodologie descritte nella sezione precedente.

Tralasciando l'analisi dettagliata della gestione di interruzioni ed eccezioni, la quale sarà il tema della prossima sezione, è possibile ad ogni modo notare la presenza di un ulteriore *bug* piuttosto grave, rilevato durante l'analisi di quest'applicazione e verificato sperimentalmente; gli sviluppatori hanno implementato l'operazione di aggiornamento della data dell'ultimo elemento salvato prima di effettuarne l'effettivo salvataggio su server. Si consideri il caso in cui si stia salvando un sms ricevuto; aggiornando il campo *PREF_MAX_SYNCED_DATE_SMS* con la data di ricezione, l'applicazione considererà il backup avvenuto e non lo terrà più presente per le future operazioni. Tuttavia, se per qualsiasi ragione l'esecuzione del metodo *appendMessages()* non dovesse andare a buon fine, il messaggio non verrebbe salvato sulla casella di posta, né ora né con future operazioni di backup; ad aggravare la situazione, l'utente non sarebbe in alcun modo notificato del problema. Tale errore è ovviamente imputabile solo agli sviluppatori di *SMS Backup+*, ed in nessun modo al modello di programmazione Android.

3.3.3 Analisi della gestione delle interruzioni dell'operazione di Backup

Terminata l'analisi dell'operazione di backup in caso di corretto funzionamento, cioè quando l'applicazione termina positivamente e tutti gli elementi considerati nelle liste sono stati salvati, occore considerare il comportamento di *SMS Backup+* in caso di interruzione del processo. L'interruzione dell'operazione di salvataggio può essere suddivisa in due categorie principali, a seconda dell'evento che l'ha causata:

- *Interruzione richiesta*: in questa categoria rientrano le interruzioni generate da entità esterne al *BackupTask*, e richieste tramite il metodo *cancel()*. L'attuale implementazione dell'applicazione prevede un unico evento di questa categoria, causato esplicitamente dall'utente; tale notifica può avvenire solo tramite la pressione dell'apposito pulsante, presente sull'interfaccia grafica. Pur non potendo prevedere quando la richiesta d'interruzione possa arrivare, questa tipologia di evento risulta particolarmente semplice da gestire, poiché si trova convogliata all'interno di un unico metodo. Nel proseguo della trattazione, verranno dettagliati i meccanismi di gestione suggeriti dal modello di programmazione di Android;

- *Interruzione imprevista*: questo tipo di interruzione è il risultato di un evento avvenuto al di fuori del raggio di controllo dell'applicazione; un esempio può esser dato da una caduta improvvisa della connessione. Questi tipi di eventi sono generalmente rappresentati tramite eccezioni; nel caso specifico di *SMS Backup+*, all'interno del ciclo di backup viene gestita soltanto l'eccezione *MessagingException*.

Interruzione richiesta dall'utente

La richiesta di chiusura del processo da parte dell'utente ha origine all'interno della activity *SmsSync*, in risposta alla pressione dell'apposito pulsante di interruzione. La segnalazione che ne scaturisce viene recapitata a *SmsBackupService* tramite il metodo statico *cancel()*, che si occupa di impostare la variabile *sCanceled* al valore *true*. Tale variabile viene controllata ad ogni interazione del ciclo di backup; per completezza, di seguito se ne riporta l'implementazione:

```
while(!sCanceled && (sCurrentSyncedItems < sItemsToSync)){
  /*Operazioni per il salvataggio di un messaggio*/
}
```

La chiamata a *cancel()* permette quindi di terminare l'operazione di salvataggio corrente del singolo elemento della lista, causando l'uscita dal ciclo *while* e, dopo aver effettuato alcune operazioni di chiusura, anche la terminazione del metodo *doInBackground()*. Per chiarire i passaggi della procedura di interruzione, e per una migliore comprensione del funzionamento di un AsyncTask, si faccia riferimento al diagramma in figura 3.7.

Riassumendo, è da notare come gli sviluppatori dell'applicazione abbiano implementato una strategia di interruzione *custom*, basata su una variabile booleana e sul suo controllo periodico all'interno del ciclo di backup. Tuttavia, le API di Android, in merito alla classe *AsyncTask*, offrono un meccanismo apposito per interrompere l'esecuzione delle istanze di quest'ultima; il concetto espresso da tale meccanismo è del tutto simile a quello utilizzato nell'applicazione. Viene infatti suggerito l'utilizzo del metodo, non statico, *cancel(boolean mayInterruptIfRunning)* il cui utilizzo deve esser combinato ad un controllo ciclico del metodo *isCancelled()*. La logica utilizzata nell'applicazione è esattamente identica a quella proposta da Android, ed è peraltro nata nella versione del sistema operativo che ha introdotto il concetto stesso di AsyncTask (1.5); è quindi curioso come gli

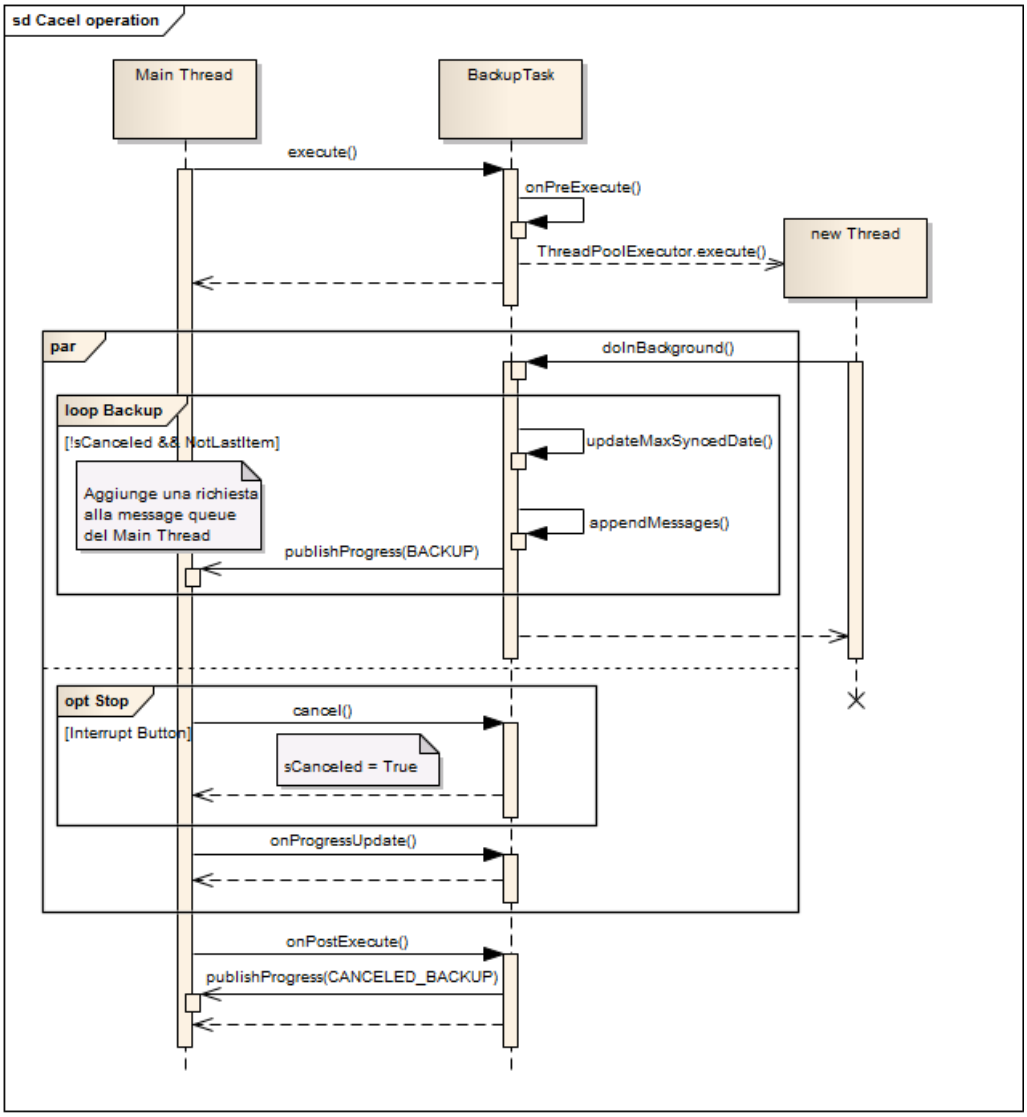

Figura 3.7: Sequence diagram di una richiesta di interruzione

sviluppatori di *SMS Backup+* abbiano scelto di utilizzare gli AsyncTask, decidendo tuttavia di non far uso della strategia di interruzione fornita con essi, sobbarcandosi peraltro il carico della gestione del parametro *sCanceled*.

Il meccanismo proposto dalle API offre inoltre la possibilità di gestire separatamente la terminazione dovuta alla chiamata del metodo *cancel()*, tramite il metodo di callback *onCanceled(Result result)*, che si sostituisce al metodo *onPostExecute(Result result)*. L'attributo *mayInterruptIfRunning*, passato al momento della chiamata a *cancel()*, indica se si desidera terminare immediatamente l'esecuzione del thread associato, senza quindi dover aspettare il controllo del parametro *isCancelled()* per poter terminare il task. E' da sottolineare come Android lasci al programmatore la scelta della granularità delle operazioni eseguite all'interno del metodo AsyncTask, suggerendo soltanto la necessità di terminare il task il prima possibile, una volta eseguito il metodo *cancel()*. Nel caso specifico di *SMS Backup+*, questa direttiva è stata decisamente seguita: l'applicativo esegue il salvataggio di un singolo elemento alla volta, rendendo l'applicazione il più possibile reattiva alle eventuali richieste di interruzione.

Risulta interessante esaminare anche i meccanismi di interruzione di operazioni in background, messi a disposizione da astrazioni che poggiano sull'utilizzo di una *message queue*. Si prenda come riferimento la classe *IntentService*; si ricorda che questa classe rappresenta un *local service*, a cui è associato un proprio thread ed una coda per le richieste, la quale gestisce automaticamente i messaggi in arrivo, uno alla volta. Un IntentService è pensato per terminare automaticamente non appena vengano esaurite le richieste presenti nella coda; tuttavia, esso non mette a disposizione meccanismi per un'interruzione forzata. L'utilizzo del metodo *stopService(Intent service)* risulta a conti fatti inutile, in quanto aggiunge alla coda di IntentService una richiesta di interruzione, che verrà servita solo dopo aver soddisfatto tutte le richieste che erano giunte precedentemente. Ipotizzandone l'utilizzo in *SMS Backup+*, la richiesta di interruzione verrebbe servita solo dopo aver eseguito l'intera richiesta di backup, risultando quindi inefficace.
Un modo di interrompere forzatamente un IntentService è quello di effettuare l'*override* del metodo di callback *onStartCommand(Intent intent, int flags, int startId)*, secondo la seguente implementazione:

```
@Override
public int onStartCommand(Intent intent,
```

```
        int flags, int startId){
    if (intent.getAction().equals("Stop"))
      stopSelf();
    super.onStartCommand(intent, flags, startId);
}
```

Sfruttando la proprietà secondo cui tale metodo viene comunque chiamato ad ogni richiesta inviata al service, occupandosi solo successivamente di inserirla nella coda, si invia un intent contenente un'esplicita richiesta di interruzione, e si sfrutta il metodo *stopSelf()* della classe padre *Service*. Tale tecnica risulta tuttavia essere un'inutile forzatura, in quanto si sta cercando di cambiare il comportamento di una componente ideata per scopi differenti. Se ne deduce quindi che, in caso sia necessario interrompere tempestivamente un'operazione eseguita in background, l'utilizzo di un AsyncTask risulta esser la miglior opzione. In alternativa, è sempre possibile creare un proprio *Worker Thread* implementando una gestione personalizzata delle richieste di interruzione; in questo modo, ovviamente, non verranno utilizzati i meccanismi di programmazione offerti dal *framework* di Android.

Interruzione causata da una eccezione

Ogni applicazione, in particolare quelle che interagiscono con componenti remoti tramite la rete, deve prevedere che il proprio comportamento possa esser influenzato da eventi esterni e non controllabili; questo implica la necessità di implementare una politica di gestione delle eccezioni, nel caso in cui tali eventi causino il malfunzionamento di un'operazione.

Nell'applicazione *SMS Backup+*, l'unica eccezione che può esser generata all'interno del ciclo di backup, che rappresenta l'operazione di maggior interesse, è rappresentata dalla classe *MessagingException*. Tale tipo di eccezione viene generata dal metodo *appendMessages()*, e può quindi essere motivata da una molteplicità di ragioni, quali un temporaneo malfunzionamento del servizio di posta al quale si tenta di accedere, oppure la caduta della connessione, dovuta a problemi di ricezione del dispositivo. Qualsiasi sia la causa scatenante, tale eccezione rappresenta un impedimento al normale svolgimento dell'operazione di backup, e deve quindi essere gestita opportunamente per consentire all'operazione di terminare senza causare ulteriori problemi, come il blocco dell'applicazione o la perdita di dati; nel caso dell'applicazione in analisi, questi inconvenienti invece avvengono, co-

me si è potuto verificare in precedenza. Per comprendere come viene gestito l'arrivo di un *MessagingException*, si faccia riferimento al diagramma in figura 3.8, rappresentante il flusso di esecuzione relativo ad un caso generale.

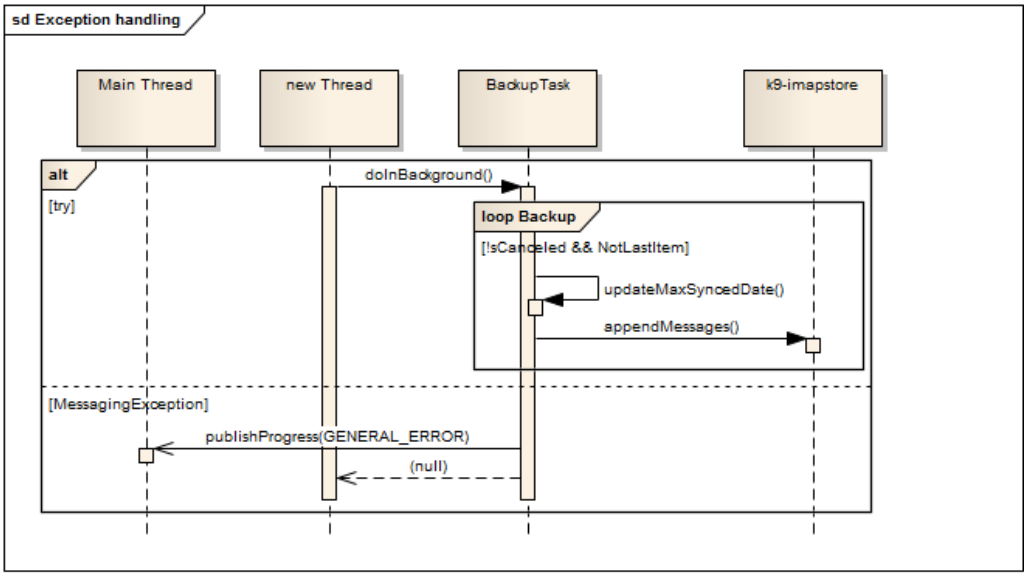

Figura 3.8: Gestione dell'eccezione MessagingException

Come si può notare, la gestione dell'eccezione si limita ad un semplice aggiornamento dell'interfaccia grafica, per notificare l'utente della situazione. Non risulta necessaria nessuna operazione di *rollback*, grazie alla gestione unitaria dei singoli *Messages* all'interno del ciclo di backup; ogni chiamata al metodo *appendMessages()* comporta la finalizzazione dello storage del singolo Message, pertanto non è prevista una memorizzazione parziale dell'elemento. Unitamente alla logica funzionale dell'applicazione, che ammette come *successful* lo storage parziale della lista di elementi da processare, è possibile considerare *safe* la gestione dell'eccezione fornita durante l'esecuzione.

In un caso più generale e meno banale di quello appena presentato, la gestione delle eccezioni è lasciata allo sviluppatore, che se ne farà carico utilizzando i pattern standard della programmazione Java; le API Android non forniscono quindi metodologie innovative per la gestione delle eccezioni.

3.4 Analisi dell'estensibilità e modularità di applicazioni scritte in Android

Al fine di verificare il suppporto all'estensibilità e alla modularità del codice fornito dal modello di programmazione Android, si è deciso di implementare alcune estensioni dell'applicazione *SMS Backup+*.
Le modifiche proposte sono:

- Implementazione di una funzione di arresto automatico dell'operazione di backup nel caso in cui la batteria scenda sotto un livello critico;
- Implementazione della gestione degli sms in arrivo durante un'operazione di backup, permettendone l'inserimento all'interno dell'operazione di salvataggio in corso senza doverne eseguire successivamente un'altra.

3.4.1 Low Battery Level

La batteria costituisce il vero tallone di Achille della maggior parte dei dispositivi mobile, che in numerosi casi resistono poco più di ventiquattr'ore senza essere ricaricati. Ha quindi senso pensare che un utente medio sia più interessato a preservare la rimanente carica della propria batteria, piuttosto che a completare un'eventuale operazione di backup, la quale fa uso intensivo di cicli di CPU e della rete internet.

La proposta di questa nuova funzionalità è quindi basata su una necessità reale e piuttosto comune, ed ha buone probabilità di essere implementata in una futura versione dell'applicazione; le premesse fatte la rendono quindi un caso di studio ideale. Per ottenere tale funzione è necessario, prima di tutto, implementare una componente sensibile agli appositi messaggi diffusi in *broadcast* dal sistema Android. Per farlo si utilizza la componente *MyBatteryReceiver*, già presentata nella sezione sui *BroadcastReceiver*; per semplicità, se ne riporta il codice qui di seguito:

```
public class MyBatteryReceiver extends BroadcastReceiver {
  private final static String TAG = "BatteryMonitor";

  @Override
```

```
  public void onReceive(Context context, Intent intent) {
    Log.i(TAG, "[MyBatteryReceiver] Low battery level!");
    /*Qui è possibile inserire il codice per la gestione
    * dell'applicazione in reazione all'evento */
  }
}
```

Il passo successivo consiste nel registrare nell'applicazione il receiver appena descritto; come esposto in precedenza, ci sono differenti modi per effettuare la registrazione; per ridurre la complessità dell'esempio, si farà ricorso al metodo più semplice, che prevede la configurazione del file *AndroidManifest.xml*:

```
<receiver android:name="MyBatteryReceiver">
  <intent-filter>
    <action android:name=
        "android.intent.action.BATTERY_LOW"/>
  </intent-filter>
</receiver>
```

L'applicazione è ora in grado di ricevere gli avvisi del sistema operativo riguardanti il livello critico della batteria; occorre infine implementare l'effettiva interruzione del task di backup. Il metodo più immediato per farlo consiste nello sfruttare il metodo statico *cancel()* di *SmsBackupService* che, come già illustrato, si occupa di far terminare *BackupTask* senza la necessità di effettuare ulteriori controlli. Applicando questa modifica a *MyBatteryReceiver*, il metodo *onReceive()* apparirà come segue:

```
@Override
public void onReceive(Context context, Intent intent) {
  Log.i(TAG, "[MyBatteryReceiver] Low battery level!");
  SmsBackupService.cancel();
}
```

L'implementazione della nuova funzionalità è ultimata e perfettamente funzionante; per chiarire la sequenza di metodi interessati in seguito alla ricezione dell'evento *BATTERY_LOW* si può fare riferimento al *sequence diagram* rappresentato in figura 3.9.

La soluzione proposta tramite l'utilizzo di un metodo statico è di facile e veloce implementazione, tuttavia tale pratica non rispecchia i principi di buona ingegneria del software. Anche se è stato possibile aggiungere una

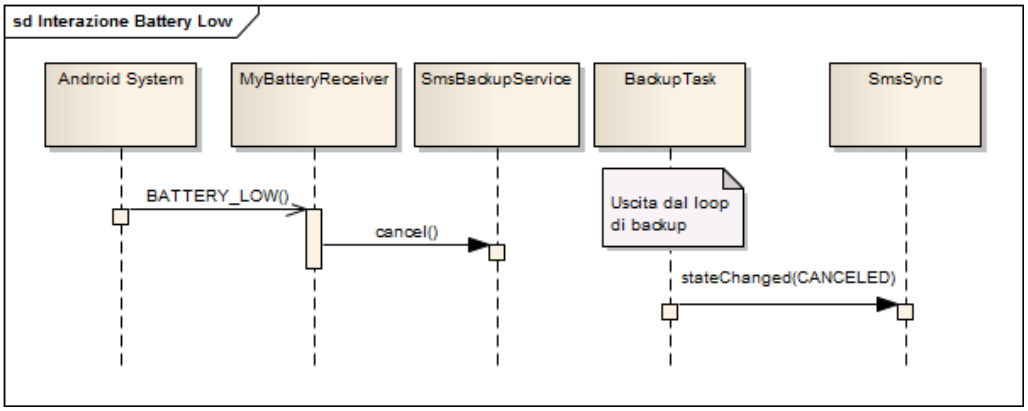

Figura 3.9: Sequence diagram della gestione dell'evento Battery Low

nuova funzionalità all'applicazione senza dover modificare le classi già esistenti, l'utilizzo di variabili o metodi statici rende il codice poco modulare. Ad esempio, se si volesse sostituire la activity *SmsSync* con un'altra per personalizzare l'interfaccia definendo un nuovo layout, sarebbe necessario modificare direttamente il codice di *SmsBackupService*, poiché in essa vengono chiamati esplicitamente i metodi di *SmsSync*. In questo caso lo sforzo sarebbe minimo, grazie anche alle funzioni di refactoring fornite dall'*IDE*; tuttavia, come insegna l'ingegneria del software, tale operazione resta uno sforzo evitabile ed in alcuni casi potrebbe esser addirittura un lusso che non ci si può permettere, ad esempio nel caso in cui i sorgenti non fossero più disponibili per la modifica.

In un'applicazione di questo tipo, evidentemente non progettata per fornire un solido supporto all'estensibilità, quello delle variabili e metodi statici rappresenta solo uno dei problemi; è palese, ad esempio, la totale mancanza di interfacce, che costituiscono uno degli elementi fondamentali per scrivere del codice ben formato. Considerando, tuttavia, che il modello di programmazione di Android stesso spinge gli sviluppatori verso soluzioni di tipo statico, è possibile ipotizzare che gli obiettivi principali del sistema operativo di Google divergano dai criteri standard di progettazione per gli applicativi a sviluppo in *team*, quali estensibilità e riuso del codice, per orientarsi maggiormente verso soluzioni più snelle, a sviluppo singolo e facilmente sostituibili da versioni successive completamente rivoluzionate.

3.4.2 New Sms

Come nel caso precedentemente analizzato, è opportuno approcciarsi al problema partendo dalla definizione di un *BroadcastReceiver*, che sia in grado, questa volta, di intercettare i messaggi diramati dal sistema in caso di arrivo di un nuovo sms. Come descritto precedentemente in questo capitolo, l'applicazione implementa già un receiver che può esser utilizzato a tal scopo; esso è definito dalla classe *SmsBroadcastReceiver*, di cui si riporta il codice qui di seguito:

```
public class SmsBroadcastReceiver
      extends BroadcastReceiver {
  public static final String SMS_RECEIVED =
      "android.provider.Telephony.SMS_RECEIVED";

  @Override
  public void onReceive(Context ctx, Intent intent) {
    if (LOCAL_LOGV) Log.v(TAG,
        "onReceive("+ctx+","+intent+")");

    if (intent.getAction().equals
        (Intent.ACTION_BOOT_COMPLETED)){
      bootup(ctx);
    } else if (intent.getAction().equals(SMS_RECEIVED)){
      incomingSMS(ctx);
    }
  }

  private void bootup(Context ctx) {...}
  private void incomingSms(Context ctx) {...}
}
```

E' rilevante notare come *SmsBroadcastReceiver* si occupi in realtà anche di ricevere e gestire le notifiche di avvio del sistema, in modo da poter riprogrammare il prossimo backup automatico andato perso con lo spegnimento del dispositivo. Non è ben chiaro il motivo per cui gli sviluppatori abbiano scelto di inserire tale controllo in questo punto del sistema; questa pratica di programmazione ingenera una sorta di incoerenza nella *naming convention*: *SmsBroadcastReceiver* non si occupa solo della ricezione di eventi legati agli sms, contrariamente a quanto è lecito pensare dal nome del receiver stesso.

Fatte queste considerazioni, occorre ora definire le modalità di implementazioni della modifica che si vuole apportare all'applicazione; a questo scopo, si procede quindi passo dopo passo, affrontando uno alla volta i problemi che si presentano:

1. Ottenimento dell'sms appena ricevuto. Questa problematica è di facile risoluzione: può essere affrontata eseguendo una *query* al content provider, oppure sfruttando le informazioni contenute all'interno dell'intent ricevuto, come proposto dal manuale:

   ```
   Object[] pduArray=
       (Object[]) intent.getExtras().get("pdus");
   SmsMessage[] messages=new SmsMessage[pduArray.length];
   for (int i = 0; i<pduArray.length; i++) {
     messages[i]=
         SmsMessage.createFromPdu((byte[]) pduArray[i]);
   }
   ```

   La sigla *PDU (Protocolo Description Unit)* presente nel codice indica uno standard industriale per la rappresentazone di messaggi sms.

2. Permettere la comunicazione tra il thread che sta effettuando il backup e il Main Thread che sta gestendo l'evento: una soluzione, di uso piuttosto comune, per permettere a diversi thread di comunicare è l'utilizzo di oggetti condivisi. Attualmente l'oggetto di tipo *Cursor* contenente gli sms da salvare viene dichiarato all'interno del metodo *doInBackground* di *BackupTask*; risulta quindi necessario spostarne almeno la dichiarazione, ad esempio all'interno di *SmsBackupService*, dando in questo modo la possibilità anche al MainThread di accedervi.

3. Gli oggetti di tipo *Cursor* non permettono l'aggiunta di nuovi elementi: occore quindi rimapparne il contenuto in un'altra struttura dati che ne permetta una gestione più dinamica, come ad esempio una *List*; sarebbe opportuno anche implementare una politica di gestione della concorrenza, tramite l'utilizzo di una classe *wrapper*.

4. Adattare il codice alla nuova struttura dati: avendo modificato la struttura dati su cui si lavora è necessario modificare manualmente quelle parti di codice che ne fanno uso.

5. Fornire un metodo per l'aggiunta del nuovo sms alla lista: anche in questo caso si può ricorrere, per una veloce risoluzione del problema, all'utilizzo di un metodo statico su *SmsBackupService*, che permetta l'aggiunta del messaggio alla lista tramite il metodo fornito dalla classe *wrapper*.

Le due modifiche implementative proposte possono essere inquadrate in un più generale *pattern* di programmazione mediante broadcast e corrispondente broadcast receiving; in particolare, nella declinazione offerta dal modello di programmazione Android, questo pattern risulta notevolmente favorito e pertanto agevolato nell'implementazione. Infatti, il modello offre la possibilità di aggiungere una componente BroadcastReceiver mediante due semplici passaggi: la configurazione del file *AndroidManifest.xml*, nel quale viene descritto il receiver da implementare nelle sue caratteristiche specifiche, e l'override del metodo *onReceive()*, per definire le operazioni da eseguire a seguito dell'intercettazione del broadcast. Agendo secondo questo pattern, e quindi sfruttando la modularità nativa offerta dal framework, otteniamo il notevole risultato di rendere una qualsiasi applicazione sensibile ad una nuova categoria di eventi, senza la necessità di modificare parti preesistenti in modo invasivo ed eccessivamente verboso.

L'analisi di risoluzione delle specifiche implementative, discusse in dettaglio nei precedenti paragrafi, mette in luce due differenti approcci rispetto al modello di progettazione ideale proposto da Android. Appare evidente, infatti, come le due modifiche proposte siano del tutto diverse quanto a complessità di risoluzione del problema: la progettazione del *MyBatteryReceiver* risulta decisamente più semplice e lineare, rispetto all'analisi effettuata per la modifica dell'*SmsBroadcastReceiver*. Le motivazioni che si trovano dietro a questa differenza sono riscontrabili in molteplici dettagli, tuttavia il *leitmotiv* che le unisce è la maggiore o minore personalizzazione richiesta dal codice scritto dallo sviluppatore, rispetto all'estensione diretta delle ottime API del framework di Android. Nel caso del *MyBatteryReceiver*, infatti, l'applicazione del pattern sopra indicato è avvenuta in modo lineare e congruente con le specifiche; è stato aggiunto l'elemento di configurazione nell'apposito file, ed è stata estesa la classe BroadcastReceiver per gestire opportunamente i broadcast di tipo *BATTERY_LOW*, generati nativamente dal sistema operativo. Nel caso della modifica di *SmsBroadcastReceiver*, al contrario, si è resa necessaria un'analisi decisamente più approfondita, poiché è stato inevitabile studiare il codice scritto direttamente

dagli sviluppatori di *SmsBackup+*. Al di là della complessità della modifica richiesta, l'analisi è risultata difficoltosa, e viziata dalla generale cattiva ingegnerizzazione del progetto, evidentemente condotto senza l'interesse a fornire una buona riusabilità e comprensione delle sue parti.

Per completezza della trattazione, è possibile ipotizzare una reingegnerizzazione massiva dell'architettura dell'applicativo, basata sull'utilizzo di *IntentService* al posto degli AsyncTask correntemente utilizzati. Secondo questo scenario, la modifica appena proposta risulterebbe decisamente più semplice; sarebbe infatti sufficiente aggiungere in coda una richiesta contenente l'sms appena ricevuto, per far sì che questo venga salvato nell'operazione di backup già in corso, senza peraltro la necessità di aggiungere alcun oggetto condiviso, evitando anche lo sforzo di gestirne gli accessi concorrenti. La motivazione di questa differenza di difficoltà implementativa è da ricercarsi nella natura delle componenti utilizzate; secondo il modello di sviluppo in ambiente Android, un AsyncTask è una componente ideata per svolgere un compito preciso, senza dover sostenere interazioni con altre entità; per questo motivo, la modifica del suo comportamento risulta particolarmente macchinosa. Al contrario, un IntentService, e in generale le componenti basate su una message queue, sono creati con lo scopo di ricevere messaggi di varia natura, e comportarsi differentemente in funzione di essi; in questo modo, quindi, promuovono un comportamento di tipo proattivo, pur rimanendo legate alle limitazioni derivanti dalla gestione di una coda.

Capitolo 4

Conclusioni

Il lavoro svolto nell'ambito della presente Tesi ha portato ad una esaustiva trattazione del modello di programmazione di Android, grazie anche ad una analisi, talvolta sperimentale, delle metodologie che questo mette a disposizione per lo sviluppo di applicazioni.

Coerentemente con gli obiettivi prefissati nel capitolo introduttivo, si è focalizzata l'attenzione su aspetti quali la gestione di eventi asincroni e l'estensibilità effettiva del codice, scritto secondo i canoni imposti dal modello.

Nella prima parte della trattazione è stata analizzata la struttura del sistema Android ed il suo modello di esecuzione; quest'ultimo, basato su di una politica di virtualizzazione dei processi, si discosta da quello classico della programmazione Java, in quanto offre una molteplicità di *entry point* alle applicazioni, elemento che ne condiziona fortemente lo sviluppo. Successivamente, è stata data particolare attenzione alla descrizione delle componenti offerte dalle API di Android, studiate per agevolare il lavoro degli sviluppatori, evidenziando il ruolo che queste ricoprono all'interno di un'applicazione; sono stati inoltre forniti alcuni esempi di utilizzo, allo scopo di evidenziarne le principali caratteristiche. Nella fase finale è stata affrontata una analisi maggiormente votata alla sperimentazione, attraverso lo studio dell'applicazione *SMS Backup+*; sono state valutate, nello specifico, le potenzialità del modello di programmazione Android, nel soddisfare i bisogni degli sviluppatori alle prese con *smart mobile applications*, le quali sono fortemente votate allo scambio di informazioni con l'esterno e sensibili ai cambiamenti dell'ambiente in cui operano.

A fronte del lavoro svolto è possibile affermare che Android offre un modello di sviluppo che non si limita ai concetti propri della programmazione *Object Oriented*; in particolare, nelle sue componenti principali, esso si differenzia notevolmente da quello standard di Java. Chi dovesse affacciarsi per la prima volta a questo mondo, si ritroverà a scontrarsi con un sistema di gestione del flusso di controllo fortemente pilotato dal sistema operativo, tramite metodi di *callback*, e con un meccanismo di comunicazione basato sullo scambio di messaggi, rappresentati dagli *Intent*. Tuttavia, una volta superato questo scoglio iniziale, non appena ci si impadronisce dello stile di progettazione per dispositivi mobile, si riesce ad apprezzare la qualità del modello proposto: quest'ultimo, infatti, permette di sviluppare molto rapidamente applicazioni, dotate di interfaccia grafica e capaci di interagire con la rete internet e con eventi di origine esterna, pagando il solo prezzo del rispetto di alcune semplici regole di progettazione.

L'estensione dell'analisi al caso reale ha tuttavia portato alla luce anche alcuni aspetti critici del modello. E' stata riscontrata, infatti, una certa difficoltà nel permettere lo scambio di informazioni tra le componenti, nel caso in cui il passaggio diretto di un riferimento non sia possibile; tale difficoltà, alimentata per lo più dalla scarsità di *best practices* ufficiali, ha portato gli sviluppatori all'adozione di soluzioni custom che, pur svolgendo in qualche modo il loro compito, nascondono inevitabilmente errori concettuali di diversa natura e gravità.

In relazione alla capacità delle applicazioni di reagire e modificare il proprio comportamento in risposta ad eventi asincroni, sono stati riscontrati diversi livelli di difficoltà, derivanti sia dal tipo di azione intrapresa, sia dalla natura della componente a cui viene assegnato il compito di eseguire suddetta azione. In particolare, è risultato evidente che gli oggetti come *AsyncTask* sono più inclini ad interrompere l'operazione svolta piuttosto che modificarla; essi, peraltro, offrono appositi metodi per la gestione delle richieste di interruzione, che sollevano lo sviluppatore da qualsiasi compito di gestione della concorrenza, e di conseguenza limitano il verificarsi di pericolose corse critiche. Altre categorie di componenti quali gli *IntentService*, al contrario, risultano migliori per modellare situazioni in cui è richiesta una maggiore flessibilità, offrendo la possibilità di aggiungere richieste a *runtime* e modificandone di conseguenza il comportamento finale. Dal punto di vista dello sviluppatore, la predisposizione di alcune componenti a svolgere certi compiti piuttosto che altri, si traduce nella necessità di una conoscenza

approfondita del modello, al fine di poter effettuare da principio la scelta che meglio si adatta alle proprie esigenze di progetto.

A chiusura dell'analisi del modello di programmazione offerto da Android, e come conseguenza delle considerazioni fatte durante lo svolgimento della Tesi, sono stati analizzati gli aspetti relativi ai concetti di estensibilità e riusabilità del codice, i quali costituiscono da sempre due elementi fondamentali per agevolare le implementazioni successive, nell'ambito di qualsiasi progetto. L'analisi svolta ha messo in forte evidenza due situazioni contrastanti: il modello di programmazione, nell'ambito dell'interoperabilità fra differenti applicazioni, fornisce il formidabile meccanismo degli *Intent* impliciti, che adempie al compito appena descritto con semplicità e trasparenza; tuttavia, nel caso di applicativi complessi e composti da differenti elementi che interagiscono vicendevolmente, il modello stesso spinge lo sviluppatore ad adottare soluzioni poco flessibili, e decisamente troppo chiuse in loro stesse; infatti, soluzioni quali l'uso di variabili e metodi statici, o l'utilizzo degli *Intent* espliciti, risultano necessarie per la corretta interazione fra parti differenti all'interno di una unica applicazione, e sono purtroppo caratterizzate da un basso indice di riusabilità e di modularità.

Ringraziamenti

Ringrazio la mia famiglia, per tutti gli sforzi che ha sostenuto in questi anni in modo da permettermi di conseguire questo traguardo.

Ringrazio la mia ragazza, Alessia, per avermi supportato e sopportato durante questi mesi.

Ringrazio tutti gli amici conosciuti all'università, che hanno reso piacevoli questi anni, e quelli conosciuti in erasmus, con cui ho condiviso una bellissima esperienza di vita.

Ringrazio Alessio, per la sua infinità pazienza e disponibilità.

Bibliografia

[1] Oha website. http://www.openhandsetalliance.com.

[2] W. Alex. Get smart. In *Communications of the ACM*, 2009.

[3] F. Enrico. E' un mondo di cellulari. sono oltre 4 miliardi, 2009. http://www.repubblica.it/2007/08/sezioni/tecnologia/-cellulari/mezzo-mondo-cell/mezzo-mondo-cell.html.

[4] Google. Android developer's guide, 2007. http://developer.android.com/guide/index.html.

[5] L. James A., J. Anthony D., and R. Franklin. Smarter phones. In *Pervasive Computing*. IEEE CS, 2009.

[6] L. Ramon, R. Kevin, and S. Michael. Idc, 2012. http://www.idc.com/getdoc.jsp?containerId=prUS23771812.

[7] K. Satya and M. Dave. In Apress, editor, *Pro Android 4*, 2012.