DEPARTMENT OF
INFORMATION
ENGINEERING
UNIVERSITY OF PADOVA

Interfacce utente per tablet nella piattaforma Android

Tesi di

Ilenia Gasparini

Relatore: Carlo Fantozzi

Indice

Un ringraziamento speciale ai miei genitori e a Luca per essermi stati vicino.
Ringrazio inoltre il Professor C. Fantozzi per l'opportunità della tesi.

Capitolo 1

Introduzione

1.1 Scopo della tesina

QUESTA tesina illustra le componenti per lo sviluppo di interfacce utente per tablet nella piattaforma Android. Tale piattaforma è stata introdotta da Google nel 2007 per proporre una soluzione libera ed aperta rivolta ai dispositivi mobili, programmabile tramite il popolare linguaggio Java. In particolare, la tesina si sofferma sull'uso dei frammenti e dell'Action Bar; al termine della trattazione è infine descritto ed implementato un semplice esempio che mostra l'utilizzo di queste componenti.

1.2 Le interfacce utente

L'interfaccia utente è l'insieme dei mezzi con cui le persone (gli utenti) interagiscono con una particolare macchina, dispositivo, programma informatico o un altro strumento complesso (il sistema). L'interfaccia utente fornisce i mezzi per:

- ◇ fornire dati e comandi al sistema (input);
- ◇ permettere al sistema di informare gli utenti (output o feedback).

Nelle scienze informatiche e nella interazione uomo-computer, parlando di interfaccia utente (di un programma informatico) si fa riferimento alle informazioni grafiche, testuali e sonore che il programma presenta all'utente e alle sequenze di controllo (come la pressione di tasti sulla tastiera del computer, i movimenti del mouse e le selezioni con touchscreen) che l'utente usa per controllare il programma.

1.2.1 Storia delle interfacce utente

La storia dell'evoluzione delle interfacce utente può essere suddivisa in quattro principali generazioni distinguibili in base al tipo predominante di interfaccia utente.

Batch System (1945-1968)

Le prime generazioni di interfacce utente non erano ancora interattive. Negli anni '60 non esistevano monitor o tastiere: gli unici dispositivi di input erano le schede perforate, gli unici dispositivi di output le stampanti. Il batch system prevedeva un'interfaccia nella quale l'interazione fra l'utente e il sistema si limitava a un singolo momento: l'invio di una sequenza di comandi batch come unità indivisibile. Tutti i comandi utente dovevano essere specificati prima di conoscere il risultato dello stesso: l'utente specificava tutti i dettagli del batch job prima dell'elaborazione e riceveva l'output quando tutta l'elaborazione era stata terminata; il computer non chiedeva ulteriori input una volta cominciata l'elaborazione. Attualmente i comandi batch sono ancora usati, ad esempio, nella gestione dei server (aggiunta e rimozione di utenti, iscrizione a mailing list, ecc.). Questo stile di interazione può avere una sua utilità per inviare comandi ripetitivi senza la necessità di una supervisione, ma non si dimostra valido quando c'è la necessità di prevedere un intervento dell'utente in qualsiasi momento per interrompere o modificare le operazioni.

CLI: Command Line Interface (1969-1983)

I sistemi time sharing furono inventati attorno agli anni '60 per permettere a più utenti di accedere simultaneamente allo stesso mainframe: questo per non sprecare la capacità di calcolo e ammortizzare i costi elevati dei primi sistemi. In questo modo però non rimanevano molte risorse disponibili per l'interfaccia utente che era così 'line oriented'. Le interfacce line oriented sono principalmente interfacce unidimensionali, nella quale l'utente può interagire solo con l'ultima riga che serve per l'immissione dei comandi (command line). Una volta che l'utente preme il tasto di invio comando, l'istruzione non è più modificabile. Ugualmente la risposta del sistema non può essere modificata. Dato che le interfacce line oriented non permettevano all'utente di muoversi sullo schermo, la tecnica di interazione era per lo più limitata a una struttura di domanda e risposta: il dialogo era condotto dal computer.

WIMP GUI (1984-oggi)

La terza generazione delle interfacce utente vede la nascita delle interfacce utente grafiche denominate WIMP (dalle iniziali delle corrispondenti parole inglesi Windows, Icons, Menu, Pointing devices) secondo quelli che sono i componenti da esse utilizzati. L'interfaccia grafica, nota anche come GUI (dall'inglese Graphical User Interface), è un tipo di interfaccia che consente all'utente di interagire con la macchina manipolando oggetti grafici convenzionali visibili sullo schermo. Il modello classico di GUI, è basato sulla presentazione di elementi grafici all'interno di finestre, sull'interazione con icone e sulla selezione di servizi mediante menu utilizzando il mouse per posizionare un puntatore su questi elementi. L'introduzione delle GUI ha notevolmente semplificato l'utilizzo del computer perché non è più richiesta la conoscenza preventiva di specifici comandi per eseguire determinate operazioni. Essa ha reso possibile l'interazione diretta con rappresentazioni grafiche degli oggetti d'interesse, permettendo anche a utenti non esperti in Informatica di interagire con programmi complessi sfruttando rappresentazioni e concetti a loro familiari. Nei sistemi operativi moderni l'interfaccia grafica è concepita tramite la metafora di un piano di lavoro rappresentato dallo schermo (detto scrivania o desktop), con le icone a rappresentare i file (di cui alcune a forma di cartellina per le directory) e le finestre a rappresentare le applicazioni. Tale ambiente di lavoro, in cui si opera attraverso il puntatore comandato con il mouse, fu definito nei laboratori Xerox (progetto Alto) e commercializzato per la prima volta nel 1981 dalla Xerox stessa con il costoso Xerox Star. In seguito, tale paradigma venne ripreso da Apple nel 1983, con il poco fortunato Apple Lisa, e nel 1984, con il più fortunato Macintosh. La prima versione di GUI WIMP a colori venne introdotta nel 1985 da Atari con l'Atari 520ST, seguita a distanza di due mesi da Commodore International con l'Amiga 1000. Le GUI WIMP sono state progressivamente introdotte nei diversi sistemi operativi, prima sotto forma di ambiente operativo (cioè come software integrativo del sistema operativo) acquistabile separatamente (nel dicembre 1983 Visi On della VisiCorp, nel febbraio 1985 GEM della Digital Research, e nel novembre 1985 Microsoft Windows della Microsoft), poi all'interno dei SO stessi. Attualmente tutti i sistemi operativi diffusi nel settore dei personal computer sono dotati di una GUI che opera secondo gli stessi principi di quella originariamente studiata da Xerox. Ciò ha causato una evoluzione significativa nell'interazione tra computer e utente: grazie all'interfaccia grafica è possibile compiere molti comuni compiti senza il bisogno di un'approfondita conoscenza del funzionamento del computer. Con la GUI è possibile fornire al software una grafica accattivante e una rinnovata semplicità d'uso che permettono un'interazione più diretta e

naturale. Il modello WIMP è ancora alla base della maggior parte delle odierne GUI; quelle più familiari alla maggior parte delle persone oggi sono: Microsoft Windows, Mac OS X, e interfacce XWindow System per computer desktop e laptop.

Vantaggi WIMP. A parte la facilità d'uso, un'interfaccia utente è un intermediario tra l'intenzione dell'utente e l'esecuzione di tale intenzione. Uno scopo da raggiungere è fare del computer un perfetto maggiordomo, che conosce i contesti, i gusti, le eccentricità e che esegue discretamente gli ordini dell'utente anticipando le sue necessità senza aver bisogno di una direzione esplicita. Lo scopo che si cerca di raggiungere nelle interfacce utente consiste nel minimizzare i meccanismi di manipolazione e la distanza cognitiva tra l'intenzione e l'esecuzione dell'intenzione. Ovviamente l'utente si focalizza sul task ma non sulla tecnologia utilizzata per il task. Le WIMP GUI hanno abilitato classi di utenti che prima del loro avvento non erano in grado di rapportarsi con i computer: ad esempio i bambini che non sanno ancora leggere o scrivere, i manager e gli utenti casalinghi non professionali. Il concetto 'punta e clicca' delle WIMP GUI è diventato parte della cultura moderna. Quello che le WIMP GUI hanno reso possibile è, di fatto, uno standard per l'interfaccia dell'applicazione che, confrontata con le interfacce delle linee di comando, offre una elevata facilità apprendimento e d'uso; inoltre facilita il trasferimento della conoscenza acquisita attraverso l'uso di un'applcazione ad un'altra applicazione, grazie alle consistenze nelle sembianze e nell'aspetto tra le applicazioni stesse (ad esempio, l'icona 'Salva' è ormai uno stereotipo utilizzato in tutte le applicazioni desktop).

Svantaggi WIMP. Gli svantaggi delle WIMP si possono riassumere in quattro punti:

1. Al crescere della complessità dell'applicazione cresce in maniera superlineare la difficoltà di apprendimento dell'interfaccia a causa dell'abbondanza di widget e funzionalità, ognuno dei quali preso singolarmente è di facile comprensione ma nell'insieme crea complessità. Alcune applicazioni desktop moderne sono talmente vaste che l'utente rifiuta alcune funzionalità e addirittura non vuole aggiornarsi alle ultime versioni per paura di perdere quel sottoinsieme di componenti che conosce.

2. Gli utenti tendono a spendere troppo tempo nella manipolazione delle interfacce rispetto alla esecuzione di compiti veri e propri. Gli utenti esperti sono spesso frustrati da troppi livelli di 'punta e clicca' e dalla confusione dello schermo dovuta alla presenza di troppi widget ricorrendo così alla scorciatoia della tatiera.

3. Le WIMP GUI sono state progettate appositamente per applicazioni bidimensionali quali word processor e fogli di calcolo. Quando invece l'informazione è tridimensionale il passaggio tra l'applicazione 3D e i controlli 2D è meno naturale introducendo una distanza cognitiva. In generale, le applicazioni tridimensionali tendono ad avere una complessità visiva più grande delle applicazioni bidimensionali.

4. L'utilizzo del mouse e della tastiera non è sempre adatto a tutte le tipologie di utenti in quanto non sono progettati per gestire attività ripetitive, per non menzionare le speciali necessità degli utenti con disabilità. Inoltre non tutti i dispositivi possono essere dotati di mouse e tastiera: si pensi ad esempio ai telefoni cellulari.

Inoltre un'altra limitazione delle WIMP GUI risiede nel fatto che sono state progettate per un singolo utente e non facilitano l'interazione fra utenti. L'interazione avviene tipicamente con un canale che risponde ad eventi di input discreti che consistono in semplice pressioni di tasti e movimenti del mouse. Infine le interfacce WIMP non supportano parola, udito e tocco: infatti le interfacce di tipo WIMP non sono ottimali per gestire compiti quali CAD, lavori su grandi moli di dati in maniera simultanea, giochi interattivi e in generale quelle che richiedono interfacce non standard.

Post-WIMP (1997-oggi)

Nelle applicazioni per le quali le tecniche WIMP non sono adatte è possibile usare nuove tecniche di interazione, che vanno sotto il nome di interfacce post-WIMP. Una interfaccia post-WIMP è una interfaccia contenente almeno una tecnica di interazione diversa dai classici elementi WIMP come menu e icone, ad esempio il linguaggio di comunicazione naturale. Un altro esempio di interazione naturale uomo-computer che non usa dispositivi o tecniche WIMP sono i giochi come i simulatori di guida con volante e cambio e simulatori di golf nei quali il giocatore si muove in un club reale per battere una palla la cui traiettoria è poi simulata e visualizzata. Un altro esempio attuale riguarda il riconoscimento vocale implementato nei recenti smartphone Android e iPhone, che permette di rispondere al telefono, dettare sms o avviare conversazioni senza usare le mani. I dispositivi mobili come gli smartphone e i PDA tipicamente usano elementi WIMP con differenti metafore dovute alle limitazioni nello spazio e ai dispositivi disponibili. Dal 2007, alcuni sistemi operativi basati sul touch-screen come iOS e Android usano la classe di interfacce GUI denominate post-WIMP. Queste supportano stili di interazione usando più di un dito a contatto con un display, che permette azioni come lo 'stringimento' e la 'rotazione', che non sono possibili usando un mouse. Inoltre, lo schermo

può fornire un feedback aptico al tocco, rendendo ad esempio più realistica la sensibilità di una tastiera virtuale visualizzata a schermo.

1.2.2 Il futuro

Nella storia dell'informatica si è assistito ad un progressivo spostamento dell'attenzione dalla macchina all'uomo e in particolare all'interazione dell'uomo con la macchina. La ricerca attuale nel campo delle interfacce e dell'interazione con il calcolatore sta studiando metodi ancora più naturali ed espressivi per comunicare con la macchina. Sono ad esempio allo studio tecniche affidabili per interagire con comandi vocali, con il movimento delle mani e del corpo e perfino attraverso le caratteristiche emotive ed espressive del proprio comportamento. Le interfacce della prossima generazione sono già in sviluppo e probabilmente estenderanno il numero di dimensioni dalle attuali 2 dimensioni, a 3 o più. Le vie più comuni per aggiungere dimensioni alle interfacce sono l'aggiunta del tempo (in forma di animazioni) dei suoni, della voce, cosi come ovviamente la creazione di un vero e proprio spazio fisico tridimensionale con l'utilizzo di sistemi per la realtà virtuale. L'obiettivo principale è però quello di rendere le interfacce usabili e semplici da utilizzare. Le due caratteristiche su cui più si punta sono la multimedialità, includendo informazioni di varia natura, e la connettività, unendo fra loro i vari sistemi. Un altro obiettivo è quello dell'unione tra applicazioni: permettere all'utente di lavorare allo stesso compito con strumenti diversi. Un altro possibile sviluppo potrebbe consistere nella creazione di interfacce che non necessitano di un dialogo esplicito con l'utente, ma che agiscono in funzione di messaggi non espliciti dell'utente, come movimento degli occhi, riconoscimento dei gesti e della postura, analisi semi intelligente delle azioni. Un'altra caratteristica importante potrebbe essere quella di avere un'interfaccia che si adegua a seconda della posizione dell'utente: ad esempio potrebbe ingrandire le dimensioni del testo quando percepisce l'utente lontano dal monitor; potrebbe avvertire l'utente con messaggi di allarme sonori nel caso in cui sia dall'altro lato della stanza; infine, nel caso in cui l'utente riceva un messaggio di posta e non sia al computer, il sistema potrebbe decidere autonomamente di inviare messaggi al suo cellulare, mandare un fax all'hotel dove si trova o inviare il messaggio al computer dell'ufficio.

1.3 Struttura della tesina

Il capitolo 1 ha introdotto le interfacce utente parlando della loro evoluzione. Il capitolo 2 fornirà una sommaria introduzione alla piattaforma Android, il-

lustrando da dove trae origine e come si è evoluta nel tempo. Il capitolo 3 descriverà la piattaforma com'è attualmente, illustrandone i componenti chiave quali activity, risorse ed intenti. Il capitolo 4 spiegherà come avviene la progettazione dell'interfaccia utente e dei principali controlli utilizzabili. Il capitolo 5, infine, presenterà un piccolo esempio con una semplice applicazione che fa uso dei frammenti e dell'Action Bar. La lettura della tesina non richiede particolari conoscenze tecniche, tuttavia è necessario aver chiaro il concetto di interfaccia utente, ed avere familiarità con i fondamenti del linguaggio di programmazione Java e con la sintassi dell'XML.

Capitolo 2

Storia di Android

ANDROID è un sistema operativo open source per smartphone, tablet e dispositivi mobili in generale, sviluppato da Google; con l'SDK messo a disposizione consente in modo relativamente semplice e potente di progettare e programmare applicazioni. Allo stato attuale Android è rilasciato in due versioni: la prima, per smartphone, è denominata Gingerbread ed è arrivata alla release 2.3, mentre la seconda, per tablet, è chiamata Honeycomb ed è giunta alla release 3.2; molto recentemente però è stata rilasciata la versione 4, che funzionerà sia per tablet che con smartphone.

2.1 La storia

Android fu sviluppato inizialmente nel 2003 da una startup californiana di nome Android Inc., fondata da Andy Rubin, Rich Miner, Nick Sears e Chris White. Siamo agli inizi del nuovo millennio, ogni telefonino ha il proprio sistema operativo e gli smartphone più evoluti sono quelli prodotti da Palm e quelli con a bordo Windows Mobile. In questo scenario, la visione di Andy Rubin era di creare un sistema operativo aperto, basato su Linux, conforme agli standard, con un'interfaccia semplice e funzionale che mettesse a disposizione degli sviluppatori strumenti efficaci per la creazione di applicazioni. E soprattutto, a differenza di tutti gli altri sistemi operativi allora disponibili sul mercato, la sua adozione doveva essere gratuita. La svolta arriva nel luglio del 2005 quando Google acquista Android Inc. trasformandola nella Google Mobile Division con a capo sempre Andy Rubin. L'acquisizione fornì a Rubin i fondi e gli strumenti per portare avanti il suo progetto. Il passo successivo fu la fondazione nel novembre dello stesso anno della Open Handset Alliance (OHA). L'Open Handset Alliance, capeggiata da Google, è formata da 35 mem-

bri fra cui alcuni operatori telefonici come Vodafone, T-Mobile, Telecom Italia; produttori di dispositivi mobili come HTC, Motorola, Samsung; produttori di semiconduttori come Intel, Texas Instruments, Nvidia; compagnie di sviluppo software e di commercializzazione. Il loro scopo è di creare standard aperti per dispositivi mobili. Il 5 novembre 2007, l'Open Handset Alliance viene istituita ufficialmente e presenta il sistema operativo Android. Qualche giorno dopo viene rilasciato anche il primo Software Development Kit (SDK) per gli sviluppatori che include: gli strumenti di sviluppo, le librerie, un emulatore del dispositivo, la documentazione, alcuni progetti di esempio e un tutorial. Nel giugno dello stesso anno era arrivato sul mercato il primo iPhone di Apple che aveva rivoluzionato il modo di concepire gli smartphone. In molti si aspettavano che Google rispondesse con un proprio smartphone per cui la sorpresa fu grande quando presentò un intero ecosistema, un sistema operativo capace di funzionare su dispositivi anche molto diversi tra loro. Al momento del lancio Android presenta un numero molto ridotto di applicazioni: un browser basato su webkit, una rubrica e un calendario sincronizzati con Gmail e poco altro. Per invogliare i programmatori a cimentarsi con questa nuova piattaforma,nel gennaio del 2008 Google istituisce un concorso con un montepremi di 10 milioni di dollari per le migliori 50 applicazioni.

2.1.1 Smartphone

Nell'agosto del 2008 viene presentata la versione 0.9 dell'SDK di Android sulla cui base l'operatore T-Mobile annuncia il primo smartphone con sistema operativo Android: il T-Mobile G1. Questo terminale, che in Italia sarà conosciuto come HTC Dream, raggiunge il mercato americano nel settembre del 2008: se ne sono venduti un milione nei primi 60 giorni! Il T-Mobile G1 (immagine 2.1) è uno smartphone con display TFT da 3,2 pollici touchscreen, un processore da 528 MHz, 256 MB di ROM e 192 MB di RAM. Ha una tastiera fisica QWERTY a scomparsa a 5 righe che costituisce l'unico metodo di input. La tastiera software verrà introdotta solo con Android 1.5 ad aprile del 2009.

A differenza di quanto avviene con l'iPhone, Android punta da subito sulla personalizzazione dell'interfaccia, nasconde alla vista l'elenco delle applicazioni installate lasciando libero l'utente di configurare a suo piacimento le schermate a sua disposizione con widget, shortcut a programmi e cartelle. Però, analogamente ad Apple, è presente un app store: l'Android Market. A differenza dell'iPhone, il sistema di notifiche non è gestito tramite pop-up ma tramite un menu a tendina che scende dall'alto. Il multitasking è pienamente supportato, infatti tenendo premuto il tasto home compare un elenco delle ultime applicazioni aperte che permette di passare dall'una all'altra. La versione di Android

Figura 2.1 – Il primo smartphone con Android

1.0 presente a bordo di questo terminale mostra un sistema operativo ancora acerbo. Mancano infatti le API per il Bluetooth (funziona solo l'auricolare), la tastiera virtuale e le API per GTalk. Lo sviluppo del software avviene però a ritmi vertiginosi e nel giro di pochi mesi vengono rilasciate numerose versioni.

Una delle caratteristiche più simpatiche di Android è il fatto che le sue diverse versioni sono si indicate a livello ufficiale con un numero di versione secondo gli standard informatici ma di norma vengono distinte per il proprio codename, tradizionalmente ispirato alla pasticceria e rigorosamente in ordine alfabetico: Cupcake (Android 1.5), Donut (1.6), Eclair (2.0/2.1), Froyo (ossia Frozen Yogurt 2.2), Gingerbread (2.3), Honeycomb (3.x), Ice Cream Sandwich (4.0) e così via.

Android 1.5 Cupcake

Nell'aprile del 2009 viene rilasciato l'SDK di Android 1.5 chiamato Cupcake. Come anticipato, a partire da questa versione Google assegna ad ogni versione del suo sistema operativo il nome di un dolce, procedendo in ordine alfabetico. Cupcake presenta delle novità sostanziali per Android: oltre alla già citata tastiera virtuale dà la possibilità di scattare foto e registrare filmati caricandoli direttamente su Picasa o Youtube, introduce il riconoscimento vocale, migliora i widget e le animazioni tra le schermate, fornisce le API per lo sfruttamento dell'accelerometro e il Bluetooth stereo.

Android 1.6 Donut

Nel settembre del 2009 viene rilasciato Android 1.6 Donut. Si tratta di una release minore che migliora la funzione integrata di ricerca rendendola trasversale ed estendendola a tutti i dati del terminale. Su Donut viene inoltre migliorata la velocità della fotocamera e della videocamera ed aggiornata l'interfaccia della galleria e dell'app store Android Market.

Android 2.0/2.1 Eclair

Nel novembre del 2009 viene rilasciato Android 2.0 Eclair. Si tratta di un aggiornamento importante per il SO di Google che permette l'integrazione dei contatti con Facebook e Twitter. Le immagini dei contatti vengono prese direttamente da Facebook e cliccandoci sopra appare un piccolo menu che presenta i modi in cui è possibile contattare la persona. Nella fotocamera viene aggiunto il supporto allo zoom digitale, alla possibilità di scelta tra alcune scene prestabilite, al bilanciamento del bianco, all'effetto colori e alla modalità macro focus. La tastiera virtuale diventa multitouch e viene fornita con dizionario più ricco. Il browser permette lo zoom con il doppio tap ed è garantita la conformità allo standard HTML5. Finalmente arrivano le API per utilizzare direttamente il Bluetooth all'interno delle applicazioni. Fa inoltre la sua comparsa Google Navigator, il navigatore satellitare integrato in Google Maps. Ed ancora: supporto alla sincronizzazione di account multipli Gmail, supporto ad altre dimensioni dello schermo e supporto ad Exchange; vengono infine introdotti i live wallpaper.

Android 2.2 Froyo

A maggio 2010 esce l'aggiornamento Android 2.2 FroYo. Grazie ad un nuovo compilatore (Dalvik) JIT si riescono ad ottenere prestazioni fino a 7 volte superiori rispetto a quelli di Android 2.1 Eclair: i programmi si aprono più velocemente e il sistema è molto più fluido. Grazie all'Engine V8 Javascript, lo stesso di Chrome, la navigazione in internet è fulminea. Viene introdotto il tethering Wi-Fi e USB che trasforma gli smartphone in hotspot mobili che condividono la connessione con altri dispositivi. Le applicazioni del Market hanno la possibilità di aggiornarsi automaticamente (auto-update) e adesso possono essere installate sulla memory card per risparmiare la memoria interna. C'è il pieno supporto ad Adobe Flash Player per tutti i siti internet che ne fanno uso. La tastiera virtuale supporta più lingue indipendentemente da quella impostata nel terminale. Viene introdotta la funzione 'send to phone' per inviare contenuti dal browser del PC direttamente allo smartphone.

Android 2.3 Gingerbread

Nel dicembre del 2010 viene introdotto Android 2.3 Gingerbread. Alcune ottimizzazioni del codice rendono Gingerbread ancora più veloce. Oltre ad un generale restyling dell'interfaccia c'è il supporto alla tecnologia Near Field Communication (NFC) sia in lettura che in scrittura. La tastiera virtuale ridisegnata, risulta molto più veloce; il meccanismo del copia e incolla è reso più agevole

e i giochi possono sfruttare nuovi driver che migliorano l'accelerazione hardware. Inoltre è ora integrato nel sistema il supporto alle telefonate via internet tramite i protocolli SIP e VOIP.

2.1.2 Tablet

Durante il 2010 vengono presentati molti tablet che utilizzano Android come sistema operativo. Quello che riceve più interesse dei media è il Samsung Galaxy Tab con la versione 2.2 FroYo che si pone in diretta concorrenza con l'iPad di Apple. Il prodotto è presentato durante l'IFA di Berlino 2010 ed è messo in commercio a partire dalla fine di Settembre 2010.

Android 3.0 Honeycomb

Nel gennaio del 2011 Google lancia Android 3.0 Honeycomb, un sistema operativo sviluppato appositamente per i tablet. Il primo dispositivo a montare Android 3.0 Honeycomb è il Motorola XOOM (si veda Figura 2.2), messo in vendita negli Stati Uniti a Febbraio 2011. Questa nuova versione del sistema operativo mette al bando ogni tasto fisico, sostituendoli con tre tasti virtuali presenti nell'angolo in basso a sinistra a prescindere dall'orientamento del tablet. Il multitasking sfrutta delle immagini di anteprima delle applicazioni aperte; le notifiche vengono ulteriormente migliorate e spostate nell'angolo in basso a destra. Sono rivisti anche i widget che, sfruttando il maggiore spazio a disposizione, riescono a mostrare gran parte delle informazioni utili senza dover aprire l'applicazione. È fornito il supporto nativo alle videochiamate e ai processori multi-core. Molte applicazioni sono ridisegnate, come pure il Market. In alto a destra è presente una barra delle azioni dinamica il cui contenuto cambia in base all'applicazione che si sta utilizzando.

Figura 2.2 – Il primo tablet con Android 3.0

Più in dettaglio, le nuove caratteristiche di Android 3.0 sono le seguenti.

1. Interfaccia utente

L'interfaccia è completamente rivoluzionata. Il sistema è ottimizzato per dispositivi dotati di uno schermo più ampio dove c'è più spazio da sfruttare ed è meglio organizzata e più personalizzabile. Google ha introdotto non solo un miglioramento del modello di interazione con i contenuti ma anche un nuovo design dell'interfaccia grafica denominato Holographic. Le applicazioni scritte per le precedenti piattaforme non dovrebbero avere problemi di portabilità mentre le prossime potranno avvalersi della nuova serie di oggetti e delle nuove funzionalità multimediali e 3D.

2. Multitasking

   Il multitasking è la caratteristica principale di questa release di Android, che permette di avviare e gestire più applicazioni contemporaneamente, passare da un'applicazione all'altra attraverso la barra di sistema e visualizzare un elenco delle applicazioni aperte di recente.

3. Action Bar

   Nella parte superiore dello schermo è presente una barra delle azioni (Action Bar) che consente di accedere alle opzioni contestuali, alla navigazione ed all'attivazione dei widget. Naturalmente questa barra è accessibile agli sviluppatori che possono aggiungere le opzioni e le funzionalità addizionali delle loro applicazioni.

4. System Bar

   Nella parte inferiore delle schermo si trova la barra di sistema (System Bar), in cui vengono visualizzate informazioni come le notifiche e gli aggiornamenti sullo stato del sistema, ma anche i pulsanti per la navigazione (indietro, home, menu). La barra è sempre visibile sullo schermo fatta eccezione per la modalità 'Lights Out' - essenzialmente una modalità full screen - che potrà essere attivata in ambiti quali la riproduzione video.

5. Schermata Home personalizzabile

   La nuova piattaforma Honeycomb consente di avere 5 schermate home personalizzabili in base alle diverse necessità. Ogni schermata ha i propri widget e le proprie applicazioni, oltre che degli sfondi esclusivi. In comune hanno l'accesso alle dock di avvio e di ricerca (delle app, dei file, dei contatti e via dicendo). Ombre e sfumature nel layout contribuiscono inoltre ad aumentarne la visibilità e la leggibilità.

6. Apps recenti

   E' disponibile un'applicazione che consente di avere un accesso diretto alle app più utilizzate. Questa è presente nella barra di sistema e sarà

davvero indispensabile in alcune circostanze, come in modalità multitask dove ci sarà bisogno di avere un accesso veloce all'app che interessa in un certo momento.

7. Tastiera

   La tastiera è stata rimodellata e riposizionata per aumentare la velocità di battitura e le funzionalità. Sono stati aggiunti nuovi pulsanti - come il tasto TAB - per rendere migliore l'esperienza della scrittura. Anche la barra di sistema supporta la nuova tastiera permettendo di visualizzare il menu dei caratteri speciali e di poter commutare tra la digitazione del testo e la modalità di input vocale.

8. Selezione del testo e copia incolla

   La selezione del testo è stata notevolmente migliorata, rendendo l'attività di copia e incolla molto più semplice e intuitiva. Basta tenere premuta la zona dove si vuole effettuare la selezione ed appariranno due freccette a delimitarla. Una volta selezionata la porzione di testo desiderata, sulla barra delle azioni verrà visualizzato il menu contestuale (copia, incolla, condividi, ricerca).

9. Connettività

   Anche in quest'ambito sono presenti delle novità. C'è la possibilità di sincronizzare file multimediali (magari tramite fotocamera collegata via USB) sfruttando il Media / Photo Transfer Protocol senza la necessità di montare il dispositivo come periferica di massa. Google promette di abbattere totalmente la connessione tramite cavo nelle future versioni del suo OS. Le funzioni di connettività permettono inoltre di collegare, tramite USB o Bluetooth, tastiere o altri dispositivi esterni. Per quanto riguarda il Wi-Fi, nuovi algoritmi di scansione promettono di ridurre il tempo di attesa per la scansione e la connessione alle reti senza fili. Migliorata, infine, la gestione delle connessioni multiple tramite Bluetooth.

10. Apps Standard

    Arricchito anche il bagaglio di applicazioni standard ottimizzate per tablet: browser, gestione fotocamera, galleria, gestione contatti e client di posta elettronica.

    - Browser

      Novità sul versante browser che implementa adesso le schede di navigazione (tab). Interessante la modalità incognito che permette di navigare senza lasciare tracce nella cronologia; cronologia e segnalibri sono accessibili da un'unica pagina.

- ◇ Fotocamera e Gallery

  L'applicazione della fotocamera è stata rielaborata per usufruire dello schermo più grande e per avere un accesso più rapido a funzioni come zoom, focus e flash. Anche l'app per la gallery è stata rivista, ma ha un look simile alla precedente fatta eccezione per una leggera ottimizzazione dovuta allo schermo più grande.

- ◇ Contatti

  L'applicazione per la gestione dei contatti è composta da 2 pannelli principali, questo per facilitare lo scorrimento tra i nomi in rubrica. Migliorata anche la formattazione dei numeri telefonici internazionali e la possibilità di classificazione del contatto (collega, amico, familiare, e via dicendo).

Android 3.1 Honeycomb

Nel maggio del 2011, viene annunciato l'aggiornamento di Android Honeycomb alla versione 3.1; esso non introduce nuovi elementi ma migliora diversi aspetti caratteristici di questo sistema operativo per tablet. Il primo miglioramento riguarda il task switcher che gestisce il multitasking: prima limitato alle ultime 5 applicazioni, adesso mostra tutte le ultime applicazioni utilizzate. Alcune applicazioni saranno ancora aperte (vedi Capitolo 3), altre potranno essere state chiuse ma verranno riaperte immediatamente nel caso fossero selezionate. Il sistema provvederà automaticamente a gestire la memoria disponibile così che l'utente non si vedrà mai costretto a forzare la chiusura di un'applicazione. Il secondo miglioramento riguarda i widget a scorrimento introdotti con Android Honeycomb, che adesso possono essere ridimensionati a piacere per mostrare un maggior quantitativo di dati senza dover aprire l'applicazione vera e propria. Il terzo miglioramento riguarda le porte USB dei tablet che adesso Android gestisce nativamente come USB Host, così da poter gestire un gran numero di periferiche come tastiere, mouse, fotocamere e controller di gioco.

Android 3.2 Honeycomb

Questa nuova release, annunciata a pochissima distanza dal rilascio della precedente Android 3.1, ottimizza di fatto le prestazioni di Android sui tablet con display da 7", introducendo al contempo il supporto per i processori Qualcomm, che cercano di imporsi come rivali delle CPU NVIDIA Tegra 2, attualmente le più utilizzate sui tablet Android. Questa

versione aggiunge inoltre nuove funzionalità per utenti e sviluppatori, come il supporto per una più vasta gamma di risoluzioni per i display e per il caricamento di contenuti multimediali dalle memory card, una nuova modalità Smart Zoom per eseguire le app sempre a schermo intero e nuove API per lo schermo, specifiche per gestire in modo più preciso l'interfaccia utente.

Il 31 gennaio 2011 viene ufficialmente dichiarato che Android è il sistema operativo di smartphone e tablet più diffuso nel mondo: infatti nell'ultimo trimestre del 2010 Android è riuscito a superare Symbian, l'incontrastato sistema operativo di Nokia per oltre 10 anni, vendendo nel mondo ben 32,9 milioni di smartphone contro i 30,6 milioni di Symbian. Dal 2008 Android è cresciuto, anno su anno, del 615.1% (tratto da [Wik4]).

Capitolo 3

Panoramica su Android

3.1 Architettura

A livello tecnico, la piattaforma open source Android è di fatto uno stack software, ovvero un insieme di strati software ognuno dei quali usa i servizi forniti dallo strato sottostante. Il primo strato è basato sul kernel 2.6 del sistema operativo Linux; l'ultimo è composto da applicazioni Java che vengono eseguite su uno speciale framework, basato anch'esso su Java e orientato agli oggetti, a sua volta eseguito su un nucleo costituito da librerie Java eseguite tramite la macchina virtuale Dalvik, specifica per dispositivi mobili, dotata (a partire dalla versione 2.2) di compilatore just-in-time (JIT).

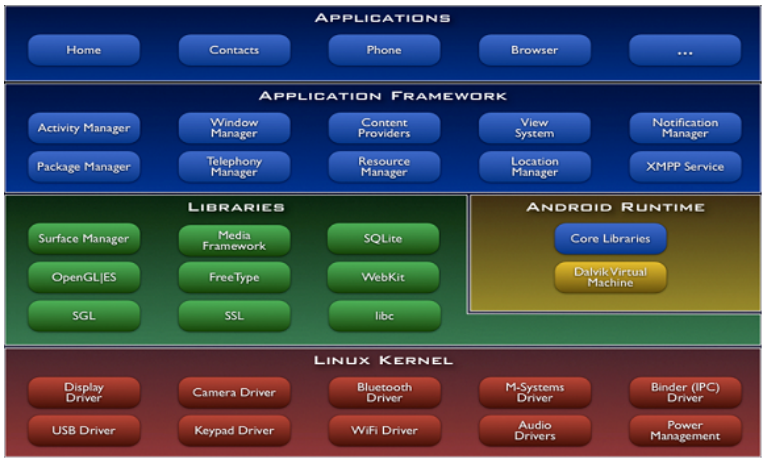

Figura 3.1 – L'architettura di Android

Linux Kernel

Internamente Android usa Linux per la gestione a basso livello del sistema (device, processi, memoria, ecc). In particolare, come si può vedere dalla Figura 3.1, direttamente nel kernel sono inseriti i driver per il controllo dell'hardware del dispositivo: driver per la tastiera, lo schermo, il touchpad, il Wi-Fi, il Bluetooth, il controllo del'audio e così via.

Libraries

Le librerie, sviluppate in C o C++, sono costituite da numerosi componenti e servono per gestire la grafica sia 2D che 3D, lo strato di interfacciamento verso un db relazionale, API per il supporto multimediale (mp3, jpeg e H.264) e per il supporto a connessioni sicure (SSL).

Android Runtime

L'architettura prevede poi una macchina virtuale e una libreria Java che, insieme, costituiscono la piattaforma di sviluppo per le applicazioni Android. Questa macchina virtuale, denominata Dalvik, è stata dedicata e ottimizzata per l'uso sui dispositivi mobili; essa è simile a una Java Virtual Machine, ma in realtà presenta alcune differenze, una Java Virtual Machine esegue del codice bytecode, invece la Dalvik Virtual Machine non esegue bytecode standard, ma un altro linguaggio, chiamato DEX (Dalvik EXecutable), studiato appositamente per una migliore resa in uno smartphone. L'ambiente di sviluppo provvede automaticamente alla generazione del codice DEX, ri-compilando il bytecode che a sua volta è frutto di una prima comune compilazione Java; il tutto avviene quindi in modo trasparente al programmatore. Un'altra peculiarità della Dalvik Virtual Machine riguarda la libreria di base che la affianca, la quale, come si può facilmente verificare aprendo l'indice dei package al percorso docs/reference/packages.html nel proprio Android SDK, differisce sia dalla Standard Edition di Java (di cui però ci sono tutti i pacchetti fondamentali ma non ad esempio AWT e Swing), sia dalla Java Mobile Edition, snobbata da Android. Non passano poi inosservati i tanti package con prefisso android che, naturalmente, sono esclusivi di questa speciale piattaforma e che servono per l'interazione diretta con le funzionalità del sistema sottostante.

Application Framework

Nel penultimo strato dell'architettura è presente un framework speciale tramite il quale è possibile rintracciare i gestori e le applicazioni di base del sistema. Ci sono gestori per le risorse, per le applicazioni installate, per le telefonate,

il file system e altro ancora: tutti componenti di cui difficilmente si può fare a meno. Le più importanti parti del framework sono le seguenti.

- ◇ Activity Manager : gestisce il ciclo di vita e il backstack delle applicazioni.
- ◇ Content Providers: memorizzano e recuperano dati e li rendono condivisibili tra le applicazioni.
- ◇ Resource Manager: manipola l'accesso alle risorse dentro ai pacchetti.
- ◇ Location Manager: fornisce la posizione del dispositivo.
- ◇ Notification Manager: colleziona eventi che avvengono in background e li notifica all'utente.
- ◇ Package Manager: manipola le informazioni sui package dell'applicazione correntemente installati nel dispositivo.
- ◇ View System: gestisce componenti dell'interfaccia utente (UI).
- ◇ Window Manager: crea finestre, spedisce eventi UI alle applicazioni.
- ◇ Telephony Manager: fornisce l'accesso ai servizi telefonici.

Applications

Infine, nello strato più alto dell'architettura, Android è fornito con una serie di applicazioni destinate all'utente finale già incluse con l'installazione di base: un player multimediale, un browser, una rubrica e un calendario. A questo livello si insericscono anche le applicazioni sviluppate dall'utente.

La Dalvik Virtual Machine

Tramite l'SDK (o meglio: tramite gli strumenti utilizzati mediante l'SDK) un'applicazione Android viene trasformata in un codice intermedio chiamato bytecode; questo è esattamente quello che accade abitualmente in Java, ossia:
Codice Java ▷ compilazione ▷ bytecode ▷ VM ▷ esecuzione reale del programma
Questo bytecode viene eseguito da un programma chiamato macchina virtuale (Virtual Machine, VM). Negli ambienti Android non viene utilizzata la Macchina virtuale Java: è stata scritta una nuova VM chiamata Dalvik Virtual Machine (DVM). Ogni terminale Android ha la sua DVM, come descritto nell'architettura del sistema; il suo compito è quello di eseguire il bytecode. Si ha quindi la seguente catena di esecuzione:
Codice Java ▷ compilazione ▷ bytecode ▷ DVM ▷ esecuzione reale dell'applicazione Android
L'idea è questa: essendo la DVM uguale per tutti i dispositivi Android, ogni

applicazione può essere eseguita su ogni terminale, indipendentemente dal costruttore e dall'architettura HW. La conseguenza di tale idea è stata che molti costruttori di dispositivi mobili scelgono Android: in questo modo possono infatti fornire ai propri utenti un ambiente condiviso da moltissimi altri utenti.

Le Applicazioni

Le applicazioni sono i programmi che vengono eseguiti dall'utente tramite l'interfaccia grafica del terminale Android. Esse sono Event Driven, ovvero sono guidate dagli eventi in quanto lo scopo è programmare un terminale mobile che risponde necessariamente ad eventi (touch schermo, azioni da tastiera). Nella tipica applicazione Android, dunque, tutto viene pilotato dall'utente.

3.2 Sviluppo delle applicazioni

Le applicazioni Android sono caratterizzate da una certa dualità: parti dinamiche scritte in Java e parti statiche scritte in XML. Parti statiche possono essere quelle caratteristiche che non cambiano durante l'esecuzione dell'applicazione, come per esempio il colore dello sfondo. Parti dinamiche sono gli aspetti programmatici come per esempio la gestione degli eventi. Questa dualità è però solo apparente. Durante l'esecuzione, infatti, la Dalvik Virtual Machine (DVM) esegue sempre un programma. Per lo sviluppo delle applicazioni è disponibile una completa documentazione la quale, anche graficamente, riprende la struttura tipica della documentazione Java del sito Oracle.

Per sviluppare applicazioni in grado di girare su sistemi Android, è necessario installare sul proprio PC:

- ◇ Java;
- ◇ un IDE (Intgrated Development Enironment), come Eclipse, che costrusice automaticamente lo scheltro dell'applicazione e ne semplifica lo sviluppo;
- ◇ un apposito kit di sviluppo (SDK), completo di emulatore, librerie e documentazione.

Benché Android SDK disponga di script che automatizzano l'installazione delle applicazioni, il lancio dell'emulatore e il debug del codice, lavorare in un ambiente integrato, con ogni opzione a portata di clic, è sicuramente più facile, specie quando l'ambiente integrato si chiama Eclipse. Per questo è disponibile anche un plug-in che Google ha scritto per la celebre piattaforma di sviluppo Open Source che è chiamato Android Development Tools for Eclipse (ADT).

3.3 Gestioni degli AVD

Il kit di sviluppo comprende un emulatore che consentirà di provare le applicazioni sul PC, prima di installarle su un reale dispositivo equipaggiato con Android. Per sviluppare le applicazioni, quindi, si deve imparare a interagire con un emulatore. Il primo concetto che è necessario assimilare è quello denominato Android Virtual Device (AVD), cioè dispositivo virtuale Android. Nel prorpio PC si possono creare e configurare quanti dispositivi virtuali si vuole. È come avere tanti differenti smartphone o tablet da utilizzare per i propri test, solo che invece di dispositivi di plastica e silicio si tratta di macchine virtuali, fatte cioè di puro software, da eseguire attraverso l'emulatore. In questo modo è anche possibile avviare contemporaneamente sullo stesso PC due o più dispositivi virtuali, ad esempio per testare un'applicazione che fa interagire più smartphone o tablet, come una chat o un gioco multiplayer. Per creare e configurare un AVD basta compilare alcune voci:

- ◇ Name: il nome che si vuole attribuire al dispositivo virtuale, ad esempio 'Android1'.
- ◇ Target: la tipologia del dispositivo. Scegliendo Android 3.2 si creerà così un dispositivo virtuale compatibile con la versione 3.2 delle specifiche di Android.
- ◇ SD Card: qui è possibile dotare il dispositivo virtuale di una scheda di memoria virtuale. È possibile specificare sia il percorso di un file di immagine di una scheda di memoria, se si vuole riutilizzare una memoria virtuale esistente, sia una dimensione di spazio, per creare una nuova memoria virtuale. Percorriamo quest'ultima strada e specifichiamo il valore 64M. Verrà così creata una scheda di memoria virtuale di 64 MB.
- ◇ Skin: dall'elenco è possibile scegliere la risoluzione del dispositivo. Le principali scelte possibili sono HVGA-P (equivalente a 480x320), HVGA-L (320x480), QVGA-P (320x240) e QVGAL (240x320). C'è poi una scelta di default chiamata semplicemente HVGA, che corrisponde comunque a HVGA-P. Vi sono poi anche WVGA (480x800), WXGA (1280x800), WVGA854 (480x854), WQVGA400 (240x400) e WQVGA432 (240x432).

Il nuovo dispositivo virtuale entrerà a far parte dell'elenco gestito dal manager, e potrà essere utilizzato per eseguire il debug e il test delle applicazioni.

3.4 Tipi di applicazione

In Android esistono due tipi di applicazioni.

Figura 3.2 – Un emulatore per tablet

- ◇ Applicazioni: occupano tutto lo schermo principale come per esempio il browser web standard di Android.

- ◇ Widget: occupano una piccola e fissata porzione dello schermo principale come per esempio l'orologio standard di Android.

3.5 Isolamento di un'applicazione

Ogni applicazione gira nel proprio processo Linux con il proprio user ID. Il sistema operativo Linux assicura che le applicazioni non accedano a privilegiati componenti del sistema operativo stesso o alla memoria e ai dati di un'altra applicazione. Inoltre ogni applicazione viene eseguita da una propria copia della Dalvk Virtual Machine, per cui eventuali malfunzionamenti non possono propagarsi da un'applicazione all'altra. Le applicazioni possono comunicare tra loro attraverso i Content Provider.

3.6 Componenti di un'applicazione

Ciascun ambiente, Java e non, dispone dei suoi mattoni fondamentali, che lo sviluppatore può estendere e implementare per trovare un punto di aggancio con la piattaforma. Android non sfugge alla regola: a seconda di quel che si intende fare è disponibile un diverso modello. Le applicazioni Android, si compongono di questi quattro componenti fondamentali: le attività (activity), i servizi (service), i broadcast receiver e i content provider. Ogni applicazione è formata da uno o più di questi elementi e non è detto che li contenga tutti: ad esempio potrebbe essere costituita da due activity e da un servizio, senza avere broadcast receiver né content provider. Nella stragrande maggioranza dei casi, comunque, le applicazioni comprendono almeno un activity. Le activity, di conseguenza, sono l'elemento fondamentale tra componenti di base delle ap-

plicazioni Android. I principali componenti di un'applicazione Android sono quindi:

◇ Activity

Le activity sono quei blocchi di un'applicazione che interagiscono con l'utente utilizzando lo schermo e i dispositivi di input messi a disposizione dal dispositivo. Comunemente fanno uso di componenti UI già pronti, come quelli presenti nel pacchetto android.widget. Le activity sono probabilmente il modello più diffuso in Android, e si realizzano estendendo la classe base android.app.Activity.

◇ Servizio

Un servizio gira in backgrund e non interagisce direttamente con l'utente. Ad esempio può riprodurre un brano MP3, mentre l'utente utilizza delle activity per fare altro. Un servizio si realizza estendendo la classe android.app.Service.

◇ Broadcast Receiver

Un Broadcast Receiver viene utilizzato quando si intende intercettare un particolare evento di sistema. Ad esempio lo si può utilizzare se si desidera compiere un'azione quando si scatta una foto o quando parte la segnalazione di batteria scarica. La classe da estendere è android. content. BroadcastReceiver.

◇ Content Provider

I Content Provider sono utilizzati per esporre dati e informazioni. Costituiscono un canale di comunicazione tra le differenti applicazioni installate nel sistema. Si può creare un Content Provider estendendo la classe astratta android.content.Content Provider.

3.6.1 Activity vs applicazione

Spesso questi due concetti vengono confusi. In generale:

◇ Activity: è associata a una singola e precisa cosa che l'utente può fare. Ad esempio: un'activity per scrivere una nota in una casella di testo editabile.

◇ Applicazione: contiene activity (e altri componenti). Ad esempio: l'applicazione Notepad ha un'activity per modificare una nota e un'activity per gestire una lista di note.

3.7 Come il sistema gestisce le activity

Stando alla documentazione ufficiale, un'activity implementa una singola e precisa cosa che l'utente può fare. L'utente, per fare qualcosa, deve interagire con il dispositivo. Un ruolo essenziale nell'interazione tra l'uomo e il dispositivo è svolto dai meccanismi di input, come la tastiera e il touchscreen, che permettono all'utente di specificare il proprio volere. Le periferiche di input, tuttavia, da sole non bastano. Affinché l'utente sappia cosa può fare e come farlo, ma anche affinché il software possa mostrare all'utente il risultato elaborato, è necessario un canale aggiuntivo. Nella maggior parte dei casi questo canale è il display. Sulla superficie dello schermo il software disegna tutti quegli oggetti con cui l'utente può interagire (bottoni, menu, campi di testo), e sempre sullo schermo viene presentato il risultato dell'elaborazione richiesta. Quindi, per fare qualcosa con il dispositivo, è necessario usare lo schermo. Esiste perciò un parallelo tra il concetto di activity in Android e quello di finestra in un sistema desktop, benché non siano esattamente la stessa cosa. In generale, si può quindi dire che le activity sono quei componenti di un'applicazione Android che fanno uso del display e che interagiscono con l'utente. In maniera più pragmatica, un'activity è una classe che estende android.app.Activity. L'autore del codice, realizzando l'activity, si serve dei metodi ereditati da Activity per controllare cosa appare sul display, per assorbire gli input dell'utente, per intercettare i cambi di stato e per interagire con il sistema sottostante.

3.8 Ciclo di vita di un activity

In un sistema desktop il monitor è sufficientemente spazioso da poter mostrare più finestre simultaneamente, perciò è possibile lavorare con più programmi contemporaneamente in esecuzione e visibili, le cui finestre vengono affiancate o sovrapposte. Gli smartphone e i tablet, invece, funzionano diversamente. Poiché le risorse di calcolo sono modeste, le activity di Android hanno carattere di esclusività. Un'activity passa attraverso i seguenti stati:

- ◇ ACTIVE (o RUNNING): visibile, riceve l'input dall'utente
- ◇ PAUSED: parzialmente visibile, non riceve l'input dall'utente
- ◇ STOPPED: non visibile (ma ancora in esecuzione)
- ◇ DESTROYED: rimossa dalla memoria di Android

È possibile mandare in esecuzione più activity simultaneamente, ma soltanto un'activity alla volta può essere in primo piano e occupare l'intero display.

L'activity che occupa il display è in esecuzione e interagisce direttamente con l'utente. Le altre, invece, sono tenute nascoste in sottofondo. L'utente, naturalmente, può ripristinare un'activity e riprenderla da dove l'aveva interrotta, riportandola in primo piano. Il cambio di activity può anche avvenire a causa di un evento esterno. Il caso più ricorrente è quello della telefonata in arrivo: se il telefono squilla mentre si sta usando la calcolatrice, quest'ultima sarà automaticamente mandata in sottofondo. L'utente, conclusa la chiamata, potrà richiamare l'activity interrotta, riprendendo i calcoli esattamente da dove li aveva interrotti. In Android il concetto di chiusura delle activity è secondario e tenuto nascosto all'utente. Le activity di Android non dispongono di un bottone 'x', o di un tasto equivalente, con il quale è possibile terminarle. L'utente, di conseguenza, non può chiudere un'activity, ma può solo mandarla in sottofondo. Questo, comunque, non significa che le activity non muoiano mai, anzi! Per prima cosa le activity possono morire spontaneamente, perché hanno terminato i loro compiti. Insomma, anche se il sistema non fornisce automaticamente un bottone 'chiudi', è sempre possibile includerlo nelle proprie applicazioni. In alternativa, la distruzione delle activity è completamente demandata al sistema. I casi in cui un'activity può terminare sono due:

- ◇ L'activity è ibernata e il sistema, autonomamente, decide che non è più utile e perciò la distrugge.
- ◇ Il sistema è a corto di memoria, e per recuperare spazio inizia a uccidere bruscamente le activity in sottofondo.

Non necessariamente un'activity non visibile viene fermata dal sistema, anche se ciò può accadere, e quindi bisogna progettare l'applicazione assumendo che accada sempre 'il peggio'. Esistono poi dei task manager di terze parti che permettono di terminare le activity in sottofondo, ma non sono previsti nel sistema di base. I differenti passaggi di stato di un'activity attraversano alcuni metodi della classe Activity che si possono ridefinire per intercettare gli eventi di nostro interesse. La Figura 3.3 illustra la sequenza di chiamate ai metodi di Activity eseguite durante i passaggi di stato dell'activity. Nel dettaglio:

- ◇ `protected void onCreate(android.os.Bundle savedInstanceState)`: richiamato non appena l'activity viene creata. L'argomento `savedInstanceState` serve per recuperare un eventuale stato dell'activity salvato in precedenza da un'altra istanza che è stata terminata. L'argomento è null nel caso in cui l'activity non abbia uno stato salvato.
- ◇ `protected void onStart()`: richiamato per segnalare che l'activity sta per diventare visibile sullo schermo.

- ◇ `protected void onResume()`: richiamato per segnalare che l'activity sta per iniziare l'interazione con l'utente.
- ◇ `protected void onPause()`: richiamato per segnalare che l'activity non sta più interagendo con l'utente.
- ◇ `protected void onStop()`: richiamato per segnalare che l'activity non è più visibile sullo schermo.
- ◇ `protected void onRestart()`: richiamato per segnalare che l'activity sta venendo riavviata dopo essere stata precedentemente arrestata.
- ◇ `protected void onDestroy()`: richiamato per segnalare che l'applicazione sta per essere terminata.

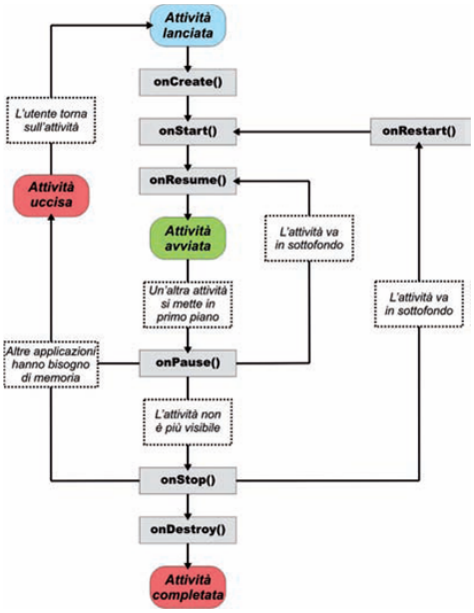

Figura 3.3 – Ciclo di vita di un'activity

Le transizioni di stato sono una conseguenza delle decisioni dell'utente o del sistema. Un'activity non può evitare transizioni di stato: può solo gestirle eseguendo appropriate azioni.

3.9 Sottoattività

Come spiegato sopra, un'applicazione Android può contenere più di un'activity. In questo caso una soltanto sarà marcata come activity principale di lancio. Le altre saranno, invece, delle sottoattività, che l'activity principale potrà

lanciare quando ce n'è bisogno. Realizzare una sottoattività è semplice tanto quanto realizzare l'activity principale: ancora una volta è sufficiente estendere android.app.Activity. Tutte le activity, sia quella principale che quelle secondarie, vanno poi registrate nel file AndroidManifest.xml, affinché il sistema sappia della loro esistenza.

3.10 Back Stack

Quando l'utente avvia un'applicazione, Android mette l'activity principale in primo piano sullo schermo del dispositivo. Da tale applicazione l'utente può invocare un'altra activity della stessa applicazione, e poi un'altra, e così via. L'Activity Manager di Android sistema le activity che fanno parte di una stessa applicazione in uno stack, detto anche Back Stack, secondo l'ordine con cui esse vengono aperte. In qualsiasi momento, l'utente può premere il tasto BACK per ritornare alla schermata precedente.

3.11 Il manifesto

Il manifesto descrittore dell'applicazione è un file .xml, AndroidManifest.xml, che viene generato automaticamente via Eclipse e sul quale all'atto, della creazione di un progetto, sono già state eseguite alcune configurazioni iniziali secondo le proprietà specificate dal programmatore. Questo file descrive l'applicazione nel suo complesso al dispositivo; in particolare al suo interno vanno dichiarati i componenti del software e precisamente le activity, i servizi, i provider e i receiver eventualmente presenti nell'applicazione, in modo che il sistema possa agganciarli e azionarli correttamente. Inoltre vanno specificate le caratteristiche hardware e software richieste dall'applicazione (ad esempio il livello delle API, le librerie API addizionali) ed elencati i permessi per accedere a determinate risorse od operazioni critiche quali la rubrica, il GPS, il web.

3.12 Le risorse

Le risorse sono file, di cui l'applicazione necessita, che non contengono codice, come ad esempio immagini, suoni, video, icone, preferenze e altro ancora. Nel momento in cui viene creato un progetto, Eclipse predispone un albero di cartelle, all'interno del quale vengono generati automaticamente diversi file. Oltre al manifesto descrittore dell'applicazione, visto prima, c'è il file default.properties, che serve esclusivamente al sistema di build automatico. Ci sono poi delle directory: src, assets, res e gen. La prima, src, deve contenere i

package e le classi dell'applicazione che si vuole realizzare, ovvero i sorgenti. Le cartelle res e assets servono per ospitare le risorse esterne necessarie all'applicazione, come le immagini, i file audio e altro ancora. La cartella res, in particolar modo, gode di una speciale struttura predefinita, formata dalle tre sotto-directory drawable, layout e values. La prima, drawable, serve per le immagini utilizzate dal software, mentre layout e values ospitano alcuni speciali file XML utili per definire in maniera dichiarativa l'aspetto dell'applicazione e i valori utilizzati al suo interno. Infine, c'è la cartella gen, che contiene la speciale classe chiamata `R`, probabile abbreviazione di Resources, che viene generata automaticamente dal sistema e non deve mai essere modificata a mano. Essa contiene gli identificatori che vengono usati per riferirsi alle risorse. Invocando questa classe, infatti, è possibile richiamare via codice le risorse memorizzate sotto la directory res. Inoltre è possibile aggiungere suffissi alle sottocartelle, ad esempio per il supporto multilingue e gli orientamenti dello schermo.

3.12.1 Differenza tra res e assets

La differenza tra le cartelle res e assets è poco evidente, eppure c'è. La directory res è pensata per gestire le risorse in maniera strutturata, ed infatti è suddivisa in sottocartelle. Tutte le risorse posizionate in res vengono prese in esame dal sistema di build e riferite nella speciale classe R. Quelle dentro res, dunque, sono delle risorse gestite. Sotto assets, invece, è possibile depositare qualsiasi file si desideri senza che il sistema di build esegua un'analisi preventiva e crei il riferimento in R. Le risorse esterne conservate nella directory assets possono essere caricate servendosi della classe android.content.res.AssetManager. Nella maggior parte dei casi, comunque, non c'è bisogno di ricorrere alla cartella assets, poiché res offre una maniera semplificata e completa per l'accesso alle risorse.

3.13 Intenti

In Android, activity, servizi e broadcast receiver sono attivati tramite intent. Nel dizionario di Android, un intent è 'la descrizione di un'operazione che deve essere eseguita'. Più semplicemente, gli intent sono dei messaggi asincroni che il sistema manda a un'applicazione quando si aspetta che questa faccia qualcosa. Ad esempio, l'activity principale può lanciare delle sotto-activity ricorrendo al metodo startActivity() che accetta come argomento un oggetto di tipo android.content.Intent, che rappresenta un intent. La sotto-activity verrà lanciata e occuperà lo schermo al posto di quella principale.

Capitolo 4

Componenti dell'interfaccia utente

4.1 Costruire interfacce utente: i componenti

Un'applicazione Android è una collezione di activity ognuna delle quali definisce una schermata dell'interfaccia utente. Per la costruzione e la gestione delle interfacce utente Android mette a disposizione alcuni strumenti, indispensabili in ogni applicazione, come widget e layout di base. L'SDK di Android fornisce infatti vari controlli che il programmatore può usare per costruire oggetti grafici in grado di interagire con chi impugna il dispositivo. Questo è un passaggio cruciale: tutti i dispositivi mobili di nuova generazione puntano tantissimo sull'interazione con l'utente.

4.1.1 View e ViewGroup

Alla base della costruzione delle interfacce grafiche e dei componenti che le attività possono usare per interagire con l'utente ci sono due tipi di oggetto chiave, chiamati View e ViewGroup, che tramite le rispettive classi, rappresentano il modo in cui Android classifica ed organizza ciò che è sullo schermo. Le View sono oggetti come i bottoni, le caselle di testo, le icone e tutti gli altri congegni di un'interfaccia grafica. I ViewGroup, invece, sono dei contenitori che possono mettere insieme più oggetti View. I ViewGroup, inoltre, sono a loro volta degli oggetti View, e di conseguenza possono contenere altri ViewGroup. Grazie a questa intuizione è possibile organizzare i componenti sullo schermo secondo uno schema ad albero, come quello di Figura 4.1

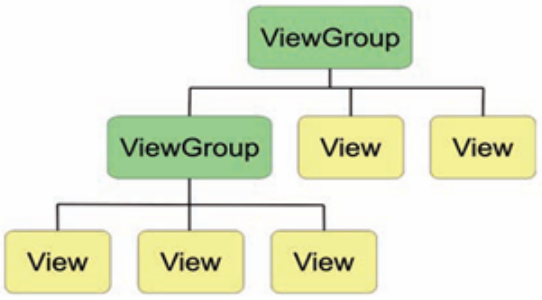

Figura 4.1 – ViewGroup

Tutti i componenti View estendono la classe base android.view.View. Nella libreria standard di Android ci sono già molti componenti di questo tipo, soprattutto nel pacchetto android.widget. Oltre ai componenti di base, ad ogni modo, è sempre possibile estendere la classe View e realizzare i propri componenti custom. Il più delle volte non c'è bisogno di farlo, poiché quelli forniti da Android bastano per tutte le principali necessità. È comunque importante che sia data questa possibilità.

La classe android.view.ViewGroup è una speciale estensione di View. Come accennato in precedenza, e come rappresentato in Figura 4.1, un ViewGroup è una speciale View che può contenere altre View. Per questo motivo gli oggetti ViewGroup dispongono di diverse implementazioni del metodo `addView()`, che permette proprio di aggiungere una nuova View al gruppo:

- `public void addView(View child)`: aggiunge un oggetto View al gruppo.
- `public void addView(View child, int index)`: aggiunge un oggetto View al gruppo, specificandone la posizione attraverso un indice (index).
- `public void addView(View child, int width, int height)`: aggiunge un oggetto View al gruppo, specificandone larghezza (width) ed altezza (height).
- `public void addView(View child, View Group.LayoutParams params)`: aggiunge un oggetto View al gruppo, applicando una serie di parametri di visualizzazione ed organizzazione del componente (params).
- `public void addView(View child, int index, ViewGroup.LayoutParams params)`: aggiunge un oggetto View al gruppo, specificando la posizione attraverso un indice (index) ed applicando una se-

rie di parametri di visualizzazione ed organizzazione del componente (params).

ViewGroup è una classe astratta, pertanto non può essere istanziata direttamente. Come nel caso di View, è possibile realizzare il proprio ViewGroup custom, ma il più delle volte conviene scegliere fra le tante implementazioni messe a disposizione dalla libreria Android. Queste implementazioni differiscono nella maniera di presentare i componenti che sono al loro interno: alcune li mettono uno dopo l'altro, altre li organizzano in una griglia, altre ancora possono essere usate per avere una gestione a schede dello schermo, e così via.

Una volta che, combinando oggetti View e ViewGroup, si è ottenuta l'interfaccia utente che si desidera, è necessario che questa venga mostrata sullo schermo. Le attività (cioè gli oggetti android.app.Activity) mettono a disposizione un metodo `setContentView()`, disponibile nelle seguenti forme:

- `public void setContentView(View view)`: mostra sullo schermo l'oggetto View specificato.
- `public void setContentView(View view, ViewGroup.LayoutParams params)`: mostra sullo schermo l'oggetto View specificato, applicando una serie di parametri di visualizzazione ed organizzazione del componente (params).

4.1.2 Widget

Con il termine widget (congegno) si indicano quei componenti di base per l'interazione con l'utente, come le caselle di testo, i bottoni, le check box, le liste, e così via. I widget intesi come i componenti delle interfacce utente sono una cosa diversa dai widget intesi come applicazioni. I widget predefiniti di Android estendono tutti (direttamente o indirettamente) la classe View, e sono conservati nel package android.widget. Alcuni di questi widget sono:

- `android.widget.TextView`: permette di mostrare del testo all'utente. Il messaggio da visualizzare può essere impostato con il metodo `setText()`, che può accettare come parametro sia una stringa sia un riferimento a risorsa preso dal gruppo R.string.

Figura 4.2 – TextView

◇ `android.widget.EditText`: estende TextView e permette all'utente di immettere del testo. Il testo digitato può essere recuperato con il metodo `getText()`, che restituisce un oggetto di tipo android.text.Editable. Gli oggetti Editable sono simili alle stringhe, ed infatti implementano l'interfaccia `java.lang.Char.Sequence`.

Figura 4.3 – EditText

◇ `android.widget.Button`: realizza un bottone che l'utente può premere o cliccare. Il componente espande TextView, e per questo è possibile impostare il testo mostrato al suo interno con il metodo `setText()`, sia con parametro stringa sia con riferimento a risorsa del gruppo R.string.

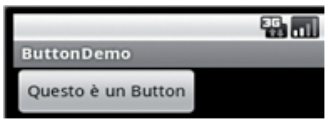

Figura 4.4 – Button

◇ `android.widget.ImageView`: permette di mostrare un'immagine proveniente da un file, da un content provider oppure da una risorsa. Metodi utili sono:

- `setImageBitmap()`, che accetta un oggetto di tipo android.graphics.-Bitmap;
- `setImageDrawable()`, che accetta un argomento android.graphics.-drawable.Drawable;
- `setImageResource()`, che accetta un riferimento a risorsa drawable.

Figura 4.5 – ImageView

◇ `android.widget.ImageButton`: un bottone con un'immagine. Estende la classe ImageView, e quindi espone gli stessi metodi di quest'ultima per impostare l'immagine mostrata.

Figura 4.6 – ImageButton

◇ `android.widget.CheckBox`: questo componente realizza una casella di spunta (check box, appunto). Estende Button e TextView, pertanto il testo a fianco della casella può essere impostato con i metodi `setText()` già noti.

Figura 4.7 – CheckBox

◇ `android.widget.RadioButton`: questo componente realizza un bottone radio. Come nel caso di CheckBox, le classi base Button e TextView forniscono i metodi necessari per l'impostazione del testo visualizzato. Un bottone radio, da solo, non ha senso. Due o più bottoni radio, pertanto, possono essere raggruppati all'interno di un `RadioGroup`. L'utente, così, potrà attivare soltanto una delle opzioni del gruppo.

Figura 4.8 – RadioButton

◇ `android.widget.ToggleButton`: un bottone 'ad interruttore', che può essere cioè 'on' o 'off'. Può essere usato per indicare l'attivazione o disattivazione di opzioni.

Figura 4.9 – ToggleButton

◇ `android.widget.DatePicker`: un componente che permette di scegliere una data selezionando giorno, mese ed anno. La data impostata dall'utente può essere recuperata servendosi dei metodi `getDayOfMonth()`, `getMonth()` e `getYear()`.

Figura 4.10 – DatePicker

◇ `android.widget.TimePicker`: un componente che permette di scegliere un orario selezionando ora e minuto. L'orario impostato dall'utente può essere recuperato servendosi dei metodi `getCurrentHour()` e `getCurrentMinute()`.

Figura 4.11 – TimePicker

◇ `android.widget.AnalogClock`: un componente che mostra all'utente un orologio analogico.

Figura 4.12 – AnalogClock

◇ `android.widget.DigitalClock`: un componente che mostra all'utente un orologio digitale.

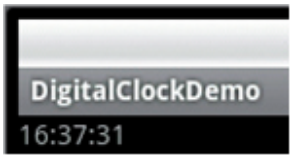

Figura 4.13 – DigitalClock

Tutti gli oggetti discussi finora richiedono, nei loro costruttori, un oggetto che estenda la classe astratta android.content.Context. Si tratta di una struttura che permette l'accesso al sistema e che costituisce il contesto di esecuzione dell'applicazione. Per ottenere oggetti di questo tipo basta sapere che android.app.Activity estende indirettamente `Context`, per cui dall'interno di un'attività, sarà sufficiente usare la parola chiave `this`. La considerazione vale per le attività, ma anche per tanti altri contesti della programmazione Android: più o meno tutte le classi che sono mattoni fondamentali del sistema estendono direttamente o indirettamente la classe astratta android.content.Context.

4.1.3 Layout

Con il termine layout (disposizione, impaginazione), in Android, si identificano tutti quei ViewGroup utilizzabili per posizionare i widget sullo schermo. Android fornisce una serie di layout predefiniti. I più comuni sono i seguenti:

- `android.Widget.FrameLayout`. È il più semplice e basilare dei layout: accetta un widget, lo allinea in alto a sinistra e lo estende per tutta la dimensione disponibile al layout stesso. La Figura 4.14 mostra un semplice esempio di utilizzo, che espande un bottone all'intera area a disposizione di un'attività.

- `android.widget.RelativeLayout`. Come FrameLayout vuole un solo componente al suo interno, ma a differenza di quest'ultimo lo disegna nelle sue dimensioni ideali, senza allargarlo per ricoprire l'intera area a disposizione. Per default, il componente viene allineato in alto a sinistra, ma è possibile controllare l'allineamento servendosi del metodo `setGravity()`, che accetta un argomento di tipo int, che è bene scegliere fra le costanti messe a disposizione nella classe android.view.Gravity.

Figura 4.14 – FrameLayout

- `android.widget.LinearLayout`. È un layout utile per disporre più componenti uno di seguito all'altro, sia orizzontalmente sia verticalmente. Una volta creato il layout, il suo orientamento può essere stabilito chiamando il metodo `setOrientation()`, con argomento pari a `LinearLayout.HORIZONTAL` o `LinearLayout.VERTICAL`. Con l'orientamento orizzontale i componenti verranno messi tutta sulla stessa riga, uno

dopo l'altro. Con l'allineamento verticale, invece, si procede lungo una colonna, e quindi i widget saranno uno sopra l'altro. Nel caso dell'allineamento orizzontale i componenti vengono introdotti lungo una sola linea. Il sistema accetta di aggiungere componenti finché c'è spazio. Se si va di poco oltre la dimensione della riga, il sistema tenta un aggiustamento restringendo i componenti al di sotto delle loro dimensioni ideali. Raggiunto un certo limite, comunque, il sistema si rifiuta di andare oltre, ed i componenti di troppo non saranno più visualizzati. Il metodo `setGravity()`, nell'allineamento orizzontale, può essere usato per decidere dove posizionare e come organizzare la riga dei componenti rispetto allo spazio disponibile. Nei LinearLayout verticali i componenti vengono aggiunti uno sopra l'altro, ed espansi in orizzontale fino ad occupare tutto lo spazio a disposizione del layout. In questo caso `setGravity()` può essere usato per decidere se allineare la colonna in alto, in basso o al centro. Il sistema aggiunge componenti finché c'è spazio nella colonna. Superato il limite, i componenti di troppo non vengono visualizzati.

Figura 4.15 – LinearLayout

◇ `android.widget.TableLayout`. È un layout che permette di sistemare i componenti secondo uno schema a tabella, suddiviso cioè in righe e colonne. I TableLayout vanno costruiti aggiungendo al loro interno degli oggetti TableRow, ciascuno dei quali forma una riga della tabella. Ogni riga è suddivisa in colonne. In ciascuna cella può essere inserito un componente. La gravità, cioè il metodo `setGravity()`, può essere usato sia su TableLayout che su TableRow, per stabilire gli allineamenti relativi.

Figura 4.16 – TableLayout

I widget ed i layout illustrati devono essere combinati in maniera coerente. I layout, in maniera particolare, possono e devono essere annidati l'uno dentro l'altro, finché non si ottiene il design desiderato.

4.1.4 Approcci

La progettazione delle interfacce utente in Android può avvenire ricorrendo a due approcci:

- ◇ *approccio programmatico, basato sul codice di programmazione in Java*. I componenti dell'interfaccia utente sono creati e gestiti dentro il codice dell'applicazione, tramite opportuni metodi di cui ogni widget dispone. Tali metodi servono per aggiungere, configurare, impostare le proprietà e disporre sullo schermo i principali widget messi a disposizione da Android.

- ◇ *approccio dichiarativo, basato sul linguaggio di markup XML*. La struttura predefinita di un progetto Android creato in Eclipse contiene sempre la directory res/layout che gestisce le differenti categorie di risorse possibili. All'interno della cartella layout possono essere disposti dei file XML che il sistema interpreterà come descrizioni dichiarative dei layout e dei widget che saranno poi usati in una o più attività dell'applicazione. Facendo doppio clic su uno di tali file, l'ambiente di sviluppo lo aprirà nel suo editor visuale. Qui è possibile aggiungere layout e widget semplicemente trascinandoli dal menu presente nell'editor visuale sulla schermata al centro, che rappresenta l'interfaccia grafica così come apparirà sullo schermo del dispositivo. Selezionando un componente è possibile accedere all'elenco delle sue proprietà, mostrate nella scheda 'Properties'. Da qui è possibile manipolare i parametri del componente e precisamente impostare gli attributi specifici di ogni widget e layout.

Nell'approccio programmatico, la logica che permea la creazione dei componenti di un'interfaccia utente è potente, ma anche lunga e noiosa. Ogni volta che si deve utilizzare un widget, lo si deve creare, personalizzare ed inserire in un contenitore predisposto in precedenza. Sin dalle origini delle interfacce basate sui widget, i creatori delle piattaforme di sviluppo hanno cercato di porre rimedio a questo difetto facendo ricorso ad editor visuali: il programmatore, anziché scrivere codice, trascina i componenti sull'editor, dimensionandoli ad occhio ed impostandone le caratteristiche salienti. Il resto del lavoro viene svolto dietro le quinte dall'editor, generando ed interpretando il codice di programmazione necessario. Questo approccio è valido, ma da solo non costituisce una vera e propria soluzione al problema: il codice prolisso e difficile da gestire, infatti, è ancora presente. Il codice generato automaticamente, infine, è spesso di difficile comprensione: l'ambiente, infatti, non ha l'intelligenza sufficiente per scrivere e mantenere un codice leggibile e performante.

Con l'avvento dei browser moderni, di AJAX e degli interpreti di nuova concezione, si sono portate sul Web molte applicazioni che, fino a ieri, erano appannaggio esclusivo degli ambienti desktop e dei linguaggi compilati. Il proliferare delle applicazioni Web sta facendo maturare velocemente gli strumenti di sviluppo propri di questo ambito. La programmazione Web ha dimostrato quanto sia più facile gestire un'interfaccia utente descrivendone i componenti con un linguaggio a marcatori, anziché con un linguaggio di programmazione. I linguaggi a marcatori come XML ben si prestano a questo genere di operazioni: sono più facili da leggere e da scrivere, sia per l'uomo sia per la macchina (cioè per gli editor visuali). Di conseguenza, oggi le piattaforme moderne applicano alla programmazione di applicazioni native il medesimo principio, fornendo agli sviluppatori framework ed editor basati perlopiù su XML. Gli attributi di XML, poi, sono molto più semplici ed intuitivi, rispetto ai metodi del tipo setProprietà() di Java. Con gli attributi è più semplice impostare le proprietà di ogni singolo componente, come il testo visualizzato, il padding, la gravità e così via. Creare un editor visuale in grado di leggere e scrivere questo XML, inoltre, è estremamente più facile che realizzare un editor in grado di fare lo stesso con del codice Java. Al contrario dell'approccio dichiarativo, l'approccio programmatico, ha inoltre come svantaggio la difficoltà di supportare linguaggi multipli e/o dimensioni dello schermo multiple e il fatto che, per modificare un'interfaccia utente, non è chiaro dove e cosa andare a modificare ed eventuali modifiche comportano la ricompilazione. A favore dell'approccio dichiarativo è anche il fatto che la rappresentazione dell'interfaccia utente è legata al codice Java che controlla il comportamento.

Per costruire un'interfaccia utente è possibile adottare anche un approccio misto che combina i due appena descritti, precisamente:

- ◇ definendo l'interfaccia utente in XML
- ◇ facendo riferimento all'interfaccia utente così definita e ai controlli in essa contenuti dentro il codice Java

4.1.5 Richiamare un layout XML

Nella directory di progetto res/layout si possono memorizzare quanti file si desidera: l'ambiente li compila e genera automaticamente un riferimento verso ciascuno di essi nella classe `R`, all'interno del gruppo layout. Ad esempio il file `res/layout/mioLayout.xml` avrà il suo riferimento in `R.layout.mioLayout`. Questo riferimento, passando al codice Java, può essere utilizzato per invocare e manipolare il layout realizzato. La classe Activity, ad esempio, dispone di una versione di `setContentView()` che accetta come argomento un riferimento ad un oggetto di tipo layout.

4.1.6 Gestire il touch: gli event listener

Per raccogliere l'input dell'utente si utilizza una tecnica che si basa sull'intercettare gli eventi scatenati dai widget.

4.1.7 Event listener

Per intercettare le azioni di tocco e digitazione eseguite dall'utente sui widget presenti sullo schermo, Android mette a disposizione un meccanismo semplice, basato sull'utilizzo dei cosiddetti event listener. Tutti i widget mettono a disposizione una serie di metodi, del tipo `setOn`*TipoEvento*`Listener();` ad esempio, il widget Button dispone del metodo `setOnClickListener()`. Attraverso questi metodi è possibile registrare al widget degli event listener, cioè delle istanze di speciali classi, realizzate appositamente per ricevere notifica ogni volta che lo specifico evento accade, ad esempio:

`button.setOnClickListener(new MyClickListener());` per ciascun tipo di event listener esiste un'interfaccia apposita, che lo sviluppatore deve implementare per creare il suo gestore dell'evento. Ad esempio, l'interfaccia da implementare per gli eventi di clic è android.view.View.OnClickListener (interfaccia innestata nella classe View). Ciascuna interfaccia, ovviamente, richiede l'implementazione di uno o più metodi. Nel caso di OnClickListener il metodo da ridefinire è:

`public void onClick(View v);` tale metodo dovrà contenere il codice necessario per gestire l'evento di clic. Il parametro ricevuto dal metodo, nel caso specifico, rappresenta l'oggetto View o derivato sul quale l'evento è stato riscontrato. Affinché la classe MyClickListener venga utilizzata come gestore dell'evento di clic su uno specifico widget, è necessario registrarne un'istanza sul widget stesso, servendosi del metodo `setOnClickListener()` citato in precedenza. Lo si può fare quando si allestisce o si richiama il layout dalla schermata. Dopo aver richiamato il layout definito nel file XML, non si deve far altro che recuperare il bottone al quale si vuole collegare l'evento e registrare su di esso il proprio listener personalizzato.

4.1.8 Come scrivere meno codice

Con gli event listener, è comunque necessario creare una classe distinta per ciascun gestore previsto, con il rischio di avere più codice dedicato alla cattura degli eventi che non alla loro gestione. Esistono diversi trucchi applicabili con gli event listener che aiutano ad evitare le situazioni di questo tipo. Per realizzare un event listener bisogna estendere un'interfaccia. Java non supporta l'ereditarietà multipla, e quindi una classe può avere una sola super-classe.

Questo limite però non vale nel caso delle interfacce: una classe ne può implementare un numero qualsiasi. Ecco allora che, nel caso di UI non troppo complesse, si può far sì che l'activity che controlla lo schermo sia essa stessa event listener di uno o più eventi, per uno o più widget: è la stessa activity a implementare l'interfaccia `OnClickListener`, definendo di conseguenza il metodo `onClick()`. Non è dunque necessario creare una classe apposita. Un'altra tecnica per risparmiare codice consiste nell'adoperare le classi innestate anonime di Java. Di fatto si crea e si registra allo stesso tempo il gestore dell'evento di clic. Ci pensa il compilatore a separare la classe anonima innestata su un file .class differente. Così facendo non c'è bisogno di far implementare alcuna interfaccia all'attività. La tecnica consente di scrivere davvero poco codice per intercettare gli eventi, lasciando il programmatore libero di concentrarsi sulla logica della loro gestione.

4.1.9 Panoramica sugli eventi

Ogni widget ha i suoi eventi e, di conseguenza, i suoi event listener. Gli eventi più diffusi, riconosciuti e che è possibile gestire su qualsiasi widget sono definiti dalla classe android.view.View; i più importanti sono:

- *Click*. Un evento che accade quando si clicca su un widget. Il metodo sul widget è `setOnClickListener()`, e l'interfaccia per il gestore da implementare è View.OnClickListener. Il metodo richiesto dall'interfaccia è `onClick(View view)`, che nel parametro riporta il widget che ha subito l'evento.

- *Click lungo*. Un evento che accade quando si clicca su un widget e si mantiene la pressione per qualche istante. Il metodo per registrare l'event listener è `setOnLongClickListener()`, e l'interfaccia per il gestore è View.OnLongClickListener. Il metodo da implementare è `onLongClick( View view)`. Il metodo, come nel caso precedente, riceve come parametro un riferimento al widget su cui si è prodotto l'evento. In più, il metodo deve restituire un booleano per segnalare se l'evento è stato completamente gestito (`true`) oppure no (`false`).

- *Tocco*. Un evento più generico dei due precedenti: serve per rilevare un tocco qualsiasi su un componente. Il metodo per registrare il listener sul widget è `setOnTouchListener()`, mentre l'interfaccia per implementarlo è View.OnTouch Listener. L'interfaccia richiede il metodo `onTouch(View view, MotionEvent event)`. Come nei casi precedenti, view è il widget che ha subito l'evento. Il parametro di tipo MotionEvent riporta invece i dettagli dell'evento (tipo di azione, coordina-

te, durata ecc.) Il metodo deve restituire un booleano per segnalare se l'evento è stato completamente gestito (`true`) oppure no (`false`).

◇ *Digitazione*. Un evento usato per segnalare la pressione o il rilascio di un tasto della tastiera. Il metodo per registrare il listener sul widget è `setOnKeyListener()`, mentre l'interfaccia per implementarlo è View. OnKeyListener. L'interfaccia richiede il metodo `onKey(View view, int keyCode, KeyEvent event)`. Come nei casi precedenti, view è il widget che ha subito l'evento. Il parametro keyCode riporta il codice associato al tasto premuto, mentre quello di tipo KeyEvent riporta ulteriori dettagli (tasto pigiato, tasto rilasciato, eventuali modificatori e così via). Il metodo deve restituire un booleano per segnalare se l'evento è stato completamente gestito (`true`) oppure no (`false`).

4.1.10 Menu

I menu sono una parte importante di qualsiasi applicazione. Gli utenti sono abituati ad avere a che fare con il concetto di menu, al quale si rivolgono ogni volta che vogliono cercare i comandi o modificare le opzioni delle loro applicazioni. In Android esistono tre differenti tipi di menu, che lo sviluppatore può collegare ad un'activity.

1. Options menu. Sono i menu concepiti per raggruppare le opzioni ed i comandi di un'applicazione. Essi costituiscono il menu principale di qualsiasi applicazione. Il menu delle opzioni è un concetto strettamente legato a quello di singola attività. Ogni Activity, infatti, può avere un solo options menu. La classe Activity dispone di un metodo definito come segue:

   ```
   public boolean onCreateOptionsMenu(Menu menu)
   ```

   Questo metodo, nel ciclo di vita dell'attività, viene richiamato automaticamente dal sistema la prima volta che l'utente preme il tasto "menu" del suo dispositivo. L'argomento passato, un oggetto di tipo android.view.Menu, costituisce l'options menu inizialmente vuoto. Ridefinendo il metodo è possibile intercettare queste chiamate e popolare così il menu fornito con le voci utili alla propria applicazione. Il metodo `onCreateOptionsMenu()`, al termine dei propri compiti, deve restituire un booleano: true per rendere attivo il menu realizzato, false per dichiarare che l'attività non dispone di un menu, e quindi alla pressione del tasto 'menu' del dispositivo non si deve mostrare nulla. Per aggiungere un elemento al menu sono disponibili alcuni metodi che si dividono in due sotto-gruppi, icon menu ed expanded menu, descritti di seguito.

- ◇ Icon menu. Sono i menu con le opzioni principali di un'applicazione. Vengono visualizzati nella parte bassa dello schermo quando si schiaccia il tasto 'menu' del dispositivo. Vengono chiamati icon menu perché gli elementi contenuti al loro interno, in genere, sono delle grosse icone che l'utente può selezionare con il dito. Costituiscono il menu principale di ogni attività e dovrebbero contenere sempre e solo le opzioni più importanti. Questi menu sono di rapido accesso, ma soffrono per questo di alcune limitazioni: possono contenere al massimo sei elementi, e non è possibile inserire negli icon menu elementi avanzati come le caselle di spunta (checkbox) e i bottoni radio.
- ◇ Expanded menu. Quando il primo tipo di menu non è sufficiente per esporre tutti i comandi e tutte le opzioni di un'applicazione, le attività fanno ricorso agli expanded menu (letteralmente menu espansi). Quando ciò avviene, il menu principale, come suo ultimo tasto, presenta il bottone 'altro'. Attivandolo si accede ad una lista aperta a tutto schermo, che permette la consultazione delle altre opzioni di menu.

2. Context menu. I menu contestuali (permettono di associare particolari opzioni o comandi ai singoli widget di un'attività) sono quelli che appaiono quando si mantiene il tocco per qualche istante su un widget che ne è dotato. Ad esempio nel browser è possibile eseguire un tocco di questo tipo sopra ad un'immagine. Dopo qualche istante verrà aperto un menu contestuale con alcune opzioni relative alla pagina corrente e all'immagine selezionata, come ad esempio i comandi per salvarla in locale e condividerla con gli amici. Come nel caso precedente, questo genere di menu si presenta come una lista a tutto schermo, che può contenere numerose opzioni.

3. Submenu. Le applicazioni che dispongono di molti comandi possono usufruire anche dei submenu. In pratica, in uno qualsiasi dei menu descritti in precedenza, è possibile inserire un elemento che, invece di compiere un'azione diretta, va ad aprire un sottomenu, nel quale si possono presentare ulteriori possibilità di scelta.

4.1.11 Notifiche

L'interattività delle applicazioni può essere ulteriormente incrementata dotandole della possibilità di emettere degli avvisi e di interrogare l'utente attraverso i cosiddetti toast e le finestre di dialogo. I primi servono per segnalare delle no-

tifiche, mentre le seconde possono essere usate sia per emettere un output sia per ricevere un input.

4.1.12 Un toast come avviso

Un toast è un avviso mostrato per qualche istante in sovraimpressione sullo schermo. Le notifiche toast sono usate per brevi messaggi testuali, e precisamente per informazioni del tipo 'impostazioni salvate', 'operazione eseguita' e simili. I messaggi toast rimangono sullo schermo per qualche istante e poi il sistema li rimuove automaticamente: non c'è alcuna interazione con l'utente. La classe di riferimento per la creazione e la gestione dei messaggi toast è `android.widget.Toast` che mette a disposizione dei metodi statici in cui i parametri da fornire sono, rispettivamente, il contesto applicativo (ad esempio l'attività stessa), il messaggio da mostrare (come stringa, nel primo caso, o come riferimento a risorsa esterna, nel secondo) e la durata del messaggio. Non è possibile specificare quanti secondi, esattamente, il messaggio dovrà restare visibile, ma si può soltanto dire se il messaggio deve durare poco o tanto tramite opportune costanti. Una volta creato, il toast può essere mostrato chiamandone il metodo `show()`.

4.1.13 Finestre di dialogo

Le finestre di dialogo sono dei riquadri che è possibile aprire sopra l'attività corrente. Quando una finestra di dialogo compare, l'attività da cui dipende viene bloccata, e l'utente deve necessariamente interagire con la finestra di dialogo per farvi ritorno. L'esempio tipico è la finestra di conferma, che domanda all'utente se vuole proseguire con una certa operazione. L'utente, quando tale finestra compare, non può far altro che scegliere tra l'opzione per proseguire e quella per annullare. Finché la scelta non viene espressa, l'attività sottostante rimane bloccata e non può essere ripresa. A differenza dei toast, quindi, le finestre di dialogo sono sia bloccanti sia interattive. Per questo motivo la loro gestione risulta leggermente più complessa. L'astrazione di base cui far riferimento è la classe `android.app.Dialog`, che definisce mediante i suoi metodi cosa una finestra di dialogo può fare e come può essere manipolata.

AlertDialog

Un tipo di finestra di dialogo è `android.app.AlertDialog`, utile per mostrare un avviso o per chiedere qualcosa all'utente, come una conferma o la selezione di un elemento da una lista. Per notificare un evento e per essere sicuri che l'utente ne prenda atto, un messaggio toast non andrebbe bene in

quanto potrebbe scomparire prima che l'utente lo noti. È invece appropriata una finestra di dialogo in grado di bloccare l'applicazione fin quando l'utente non noterà ed accetterà il messaggio.

ProgressDialog

Se è necessario svolgere delle operazioni non istantanee, che possono cioè durare qualche secondo o anche di più, si deve far capire all'utente che c'è un'operazione in corso. Se non lo si fa, l'utente potrebbe pensare che l'applicazione non gli sta rispondendo perché è lenta o bloccata. In questo caso ci si può servire di una ProgressDialog; una finestra di dialogo concepita appositamente per mettere in attesa l'utente. Lo scopo è duplice: da una parte blocca l'attività, in modo che non si possa far altro che attendere, mentre allo stesso tempo comunica all'utente che l'applicazione sta lavorando e che tutto procede come previsto.

4.2 Honeycomb

Honeycomb 3.0 è il nome della nuova versione del sistema operativo Android. Tale versione non è tanto un'evoluzione di un sistema operativo già esistente, quanto un prodotto nuovo, voluto appositamente per i tablet di nuova generazione. Questo approccio ha reso Honeycomb 3.0 più distante dalle precedenti versioni, ma con una serie di miglioramenti a partire dall'interfaccia utente, ideata con un occhio di riguardo per schermi da 7 a 10 pollici, rinnovata concettualmente. Infatti, una singola schermata può contenere molte più informazioni, in modo da dare un migliore colpo d'occhio all'utente. La schermata Home di Honeycomb 3.0 è personalizzabile, con differenti widget, collegamenti ad applicazioni installate e sfondi. La grafica è stata migliorata, con effetti tridimensionali per la gestione delle finestre e delle applicazioni multitasking.

4.2.1 L'interfaccia utente

L'interfaccia utente di Honeycomb è forse il più grande cambiamento di Android dalla prima versione 0.9, rilasciata prima che il primo smartphone fosse disponibile.

Nella parte superiore dello schermo è presente una barra orizzontale che mostra i menu delle applicazioni aperte. Questa è la Action Bar al cui interno si possono trovare anche gli strumenti per il controllo dei widget e per la navigazione Internet. Un'altra barra orizzontale che mostra le notifiche in tempo reale, installazioni effettuate, messaggi di posta elettronica ricevuti, avvisi, eventi, ecc, si trova nella parte inferiore dello schermo ed è chiamata barra di

sistema. Qui si trova anche l'elenco delle applicazioni utilizzate di recente; questa funzione è molto comoda per gli utenti che ricorrono a più applicazioni quotidianamente. In questo modo non si dovrà più andare a cercare quelle usate più spesso perché saranno li a completa disposizione. Precisamente, sul lato sinistro di tale barra ci sono i pulsanti virtuali Back, Home, Menu rispettivamente, mentre sul lato destro appaiono le icone di notifica insieme a un orologio e agli indicatori del segnale e della carica della batteria.

Figura 4.17 – L'interfaccia utente di Honeycomb

Sviluppo di un'applicazione per tablet

Per sviluppare un'applicazione appositamente per dispositivi come i tablet che montano Android 3.0 è necessario usare le API di Android 3.0. Quindi, la prima cosa da fare quando si crea un progetto per Android 3.0, è impostare la minima versione del sistema a 11 nel manifest dell'applicazione, come nel Codice 4.1

```
1 <manifest ... >
2   <uses-sdk android:minSdkVersion="11"
3   <application ... >
4     ...
5   <application>
6  </manifest>
```

Codice 4.1 – minSdkVersion

Impostando la piattaforma Android 3.0 il sistema applica automaticamente ad ogni activity il tema olografico, che è il tema standard nelle applicazioni progettate per Honeycomb. Esso fornisce un nuovo design per i widget come

bottoni, seekbar, caselle di testo e ridisegna altri elementi visuali. Inoltre, il tema olografico abilita l'Action Bar che è un widget che rimpiazza la tradizionale barra del titolo in cima alla finestra dell'activity e include sul lato sinistro il logo dell'applicazione, seguito dal titolo dell'activity e fornisce l'accesso all'utente ad alcuni items disponibili nel menu delle opzioni presenti sul lato destro.

Ottimizzare un'applicazione per Android 3.0

Le applicazioni Android sono forward-compatible e dunque un applicazione già sviluppata per una precedente versione di Android dovrebbe funzionare bene anche su dispositivi con Android 3.0. Comunque, allo scopo di fornire all'utente una miglior esperienza quando usa l'applicazione su tablet con Honeycomb, è meglio aggiornare l'applicazione affinché erediti il nuovo tema e fornire alcune ottimizzazioni per schermi grandi. Per applicare il tema olografico all'applicazione senza cambiare la versione, basta modificare l'elemento `uses-sdk` del manifest dell'applicazione come nel Codice 4.2.

```
1 <manifest ... >
2   <uses-sdk android:minSdkVersion="4"
3     android:targetSdkVersion="11" />
4   <application ... >
5     ...
6   <application>
7 </manifest>
```

Codice 4.2 – minSdkVersion

Nel caso in cui alle activity dell'applicazione siano applicati altri temi questi ultimi sovrascrivono il tema olografico.

Applicazioni solo per schermi extra-large

Android definisce i seguenti formati dello schermo.

- ◇ *smallScreens*. Indica se l'applicazione supporta il più piccolo dei fattori di forma dello schermo. Un piccolo schermo è definito come uno schermo con un rapporto di aspetto più piccolo del 'normale' (tradizionale HVGA). Un'applicazione che non supporta gli schermi di piccole dimensioni non sarà disponibile per i dispositivi di piccolo schermo da servizi esterni (come ad esempio Android Market), perché c'è poco che la piattaforma può fare per supportare una tale applicazione su uno schermo piccolo. Questo attributo è 'true' per default.

- ◇ *normalScreens*. Indica se un'applicazione supporta il fattore di forma 'normale' dello schermo. Tradizionalmente questo è uno schermo HVGA di

media densità, ma WQVGA a bassa densità e WVGA ad alta densità sono considerati nella norma. Questo attributo è 'true' per default.

- ◇ *largeScreens*. Indica se l'applicazione supporta fattori di forma dello schermo più grandi. Un grande schermo è definito come uno schermo che è significativamente più grande di uno schermo 'normale' del portatile, e quindi potrebbe aver bisogno di cure particolari da parte dell'applicazione per fare un buon uso di esso, anche se può contare sul ridimensionamento del sistema per riempire il schermo. Il valore predefinito per questo varia in realtà tra alcune versioni, quindi è meglio se si dichiara esplicitamente questo attributo in ogni momento. Attenzione al fatto che l'impostazione a 'false' abilita la modalità compatibile dello schermo.

- ◇ *xlargeScreens*. Indica se l'applicazione supporta fattori di forma dello schermo più larghi. Uno schermo xlarge è definito come uno schermo che è significativamente più grande di uno schermo 'grande', come un tablet (o qualcosa di più grande) e può richiedere cure particolari da parte dell'applicazione per fare un buon uso di esso, anche se può contare sul ridimensionamento dal sistema per riempire lo schermo. Il valore predefinito per questo varia in realtà tra alcune versioni, quindi è meglio se si dichiara esplicitamente questo attributo in ogni momento. Attenzione al fatto che l'impostazione a 'false' abilita la modalità compatibile dello schermo. Questo attributo è stato introdotto nel livello di API 9.

Quando si sviluppa un'applicazione si dovrebbe innanzitutto decidere se progettarla per dispositivi tipo i tablet, ovvero dispositivi con grandi dimensioni dello schermo, o per schermi di qualsiasi dimensione. Per rendere un'applicazione disponibile solo per dispositivi aventi schermi grandi si deve includere nel manifest dell'applicazione l'elemento `support-screens` nel quale si deve dichiarare che l'applicazione supporta schermi xlarge ponendo a `true`' l'attributo `xlargeScreens` e a `false` tutti gli altri, come nel Codice 4.3

```
1  <manifest ... >
2    ...
3   <supports-screens android:smallScreens="false"
4         android:normalScreens="false"
5         android:largeScreens="false"
6         android:xlargeScreens="true" />
7   <application ... >
8    ...
9   <application>
10 </manifest>
```

Codice 4.3 – extralarge

Servizi esterni, come Android Market, possono allora usare queste informazioni per filtrare le applicazioni dai dispositivi che non hanno schermi di grandi dimensioni. In modo analogo si può dichiarare che un'applicazione supporta schermi di piccole dimensioni. Per default, invece, applicazioni con `android:minSdkVersion` impostato a `4` e superiori, si ridimensioneranno per adattarsi a ogni tipo di schermo.

Le principali caratteristiche implementate nella versione 3 delle API di Android (indirizzate ai tablet per quanto riguarda le interfacce utente) sono:

- l'Action Bar, che è un componente grafico che sostituisce la tradizionale barra del titolo posta in alto nello schermo includendo molteplici item, come il logo dell'applicazione, il titolo dell'attività, e un option menu.
- i fragment, che rappresentano porzioni di interfaccia utente posizionati all'interno delle attività e che consentono, dunque, di organizzare, in un nuovo modo, la UI di un'applicazione

4.2.2 L'Action Bar

L'Action Bar permette di personalizzare la barra del titolo di un'attività che, nelle versioni precedenti ad Honeycomb, conteneva semplicemente il titolo dell'attività. L'Action Bar è l'intestazione di un'applicazione e il punto primario di interazione con l'utente ed è possibile customizzarla in diversi modi, ad esempio usando il framework 'style and theme' di Android per renderla maggiormente integrata con il design dell'applicazione stessa. L'implementazione dell'Action Bar si rifà al modello delle tipiche barre del titolo e del menu di un browser web presenti nelle applicazioni desktop. Essa è stata progettata in modo tale che sia possibile applicare alle applicazioni i modelli di navigazione tipo browser. Lo scopo principale dell'Action Bar è quello di rendere le azioni usate più frequentemente facilmente disponibili all'utente senza dover cercare tra option menu e context menu. Precisamente, essa fornisce varie caratteristiche di navigazione, tra le quali la capacità di:

- visualizzare gli item presenti nel menu delle opzioni direttamente nell'Action Bar, come 'item di azione', fornendo un accesso istantaneo alle azioni principali per l'utente. Gli item del menu che non appaiono come item di azione sono piazzati nel menu di overflow rivelato nell'Action Bar da una lista a discesa posta all'estremità destra.
- fornire dei 'tabs' per navigare tra i frammenti
- fornire una lista drop-down per la navigazione

◇ fornire una 'action view' interattiva al posto degli item di azione (come una casella di ricerca)

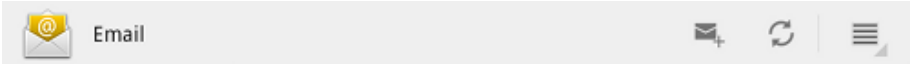

Figura 4.18 – Un'Action Bar nell'applicazione Email contenente item di azione per comporre nuove mail e aggiornare la lista messaggi

Gli item di azione

Un item di azione è semplicemente un item del menu delle opzioni che è stato dichiarato in modo tale da consentire la sua comparsa direttamente nell'Action Bar. Esso può includere un'icona e/o un testo.

Figura 4.19 – Action Bar con due item di azione e il menu di overflow

Per far apparire gli item del menu delle opzioni direttamente nell'Action Bar come item di azione basta aggiungere l'attributo `android:showAsAction='ifRoom'` ad ogni elemento item nel file XML di risorsa che descrive il menu delle opzioni. Il valore `ifRoom` specifica che l'item apparirà nell'Action Bar solo se c'è spazio disponibile per esso. Gli items per i quali non c'è spazio a disposizione vengono piazzati nel menu di overflow che si trova sul lato destro dell'Action Bar. Se gli item del menu delle opzioni forniscono sia un'icona che un testo, allora l'item di azione mostra solo l'icona per default. È possibile aggiungere all'attributo il valore `withText` (es. `ifRoom/withText`) per far apparire anche il titolo dell'item di azione adiacente all'icona (come in Codice 4.4).

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <item android:id="@+id/menu_save"
4          android:icon="@drawable/ic_menu_save"
5          android:title="@string/menu_save"
6          android:showAsAction="ifRoom|withText" />
7  </menu>
```

Codice 4.4 – XML usato per far apparire un item del menu delle opzioni nell'action bar

Un altro valore possibile per l'attributo `android:showAsAction` è `always` che specifica che l'item sarà sempre inserito nell'Action Bar; è però sconsiglia-

to l'uso di tale valore perché se ci sono troppi item di azione questi potrebbero collidere con altri elementi dell'Action Bar creando una UI troppo affollata. La presenza delle opzioni nell'Action Bar agevola l'interazione con l'utente per il quale sono più facili da trovare (essendo a portata di mano) e da usare tramite un semplice tocco quando è necessario.

L'action view

Un'action view è un componente grafico che appare nell'Action Bar come sostituto di un item di azione. Per esempio, se nel menu delle opzioni c'è un item per la 'ricerca', è possibile aggiungere per esso, qualora fosse abilitato a diventare un item di azione, un'action view che fornisce un widget SearchView nell'Action Bar.

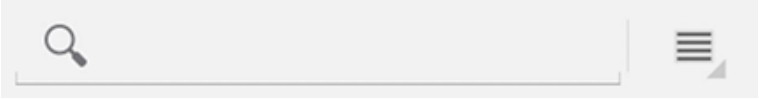

Figura 4.20 – Un'action view con un widget SearchView

Il modo migliore per dichiarare un'action view per un item è aggiungere all'elemento `item` o l'attributo `android:actionLayout` il cui valore deve essere un riferimento al file che descrive i layout, o l'attributo `android:actionViewClass` il cui valore deve essere il nome completamente qualificato della view che si desidera usare.

I tab

L'Action Bar può visualizzare 'tab' che permettono all'utente di navigare tra i vari frammenti presenti in un'activity dell'applicazione; essi possono includere un titolo e/o un'icona. Innanzitutto il layout deve includere una view che visualizzi ogni frammento associato ad un 'tab'.

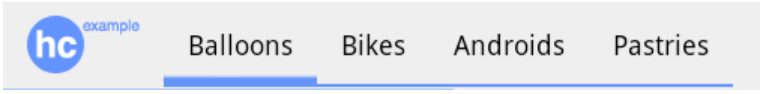

Figura 4.21 – Tabs presenti in un'Action Bar

Drop-down navigation

Un altro modo per raggruppare i comandi disponbili all'utente consiste nel fornire nell'ActionBar una lista drop-down. Tale lista può, ad esempio, fornire modalità alternative per ordinare il contenuto dell'activity o navigare tra gli account dell'utente.

4.3 I frammenti

Il più grande cambiamento in Android 3.0 è il sistema dei frammenti. Questo è uno strato opzionale che si può mettere tra le attività e i widget, progettato per aiutare il programmatore a riconfigurare le attività per supportare schermi grandi (come i tablet) e piccoli (come i cellulari).

L'interfaccia in Android 3.0 ha le sue fondamenta nella progettazione per tablet: però fortunatamente questo non significa che si debbano tralasciare le tecniche di programmazione usate prima di Honeycomb, anzi saranno di aiuto per lo sviluppo di applicazioni per tablet. Quello che Android 3.0 introduce è un insieme di concetti e caratteristiche aggiuntive che si devono saper padroneggiare per scrivere applicazioni che sfruttino il vantaggio della maggiore dimensione dello schermo. Le applicazioni scritte per le versioni pre-Honeycomb potranno sempre essere eseguite su tablet Android 3.0 o successivi, tuttavia non saranno ottimizzate per schermi più ampi.

Un frammento rappresenta un comportamento o una porzione di interfaccia utente in un'activity. All'interno di una singola activity si possono combinare più frammenti per costruire una UI multi-pannello e un frammento può essere utilizzato in più activity. Si può pensare a un frammento come a una sezione modulare di un'activity, la quale ha un suo proprio ciclo di vita, riceve i propri eventi in input e può essere aggiunta o rimossa mentre l'activity è in esecuzione. I frammenti non sono widget, come Button o EditText, né contenitori come LinearLayout o RelativeLayout, e nemmeno attività; una maniera di considerare un frammento è quella di intenderlo come una sottoattività, in quanto la sua semantica è simile a quella di un attività. Pensare ai frammenti come ad una maniera per visualizzare più attività sullo schermo nello stesso momento è un buon modo per assimilare questo concetto. Poiché potrebbe essere difficile gestire più di un'attività sullo schermo di un tablet, i frammenti sono stati creati per implementare questa filosofia; ciò significa che i frammenti sono contenuti all'interno di un'attività e hanno senso solo dentro il contesto di un'attività. I frammenti aggregano widget e contenitori e non possono esistere al di fuori di un'attività, rappresentando quindi un pezzo di interfaccia utente che può essere usata in diverse attività (dimensionate sulla taglia dello schermo).

Un'applicazione Android è formata da un insieme di attività raggruppate insieme per raggiungere uno scopo ben preciso, ad esempio gestire un'account e-mail, leggendo e spedendo messaggi. Su di un cellulare vi è spazio per gestire una singola attività alla volta, e ovviamente questo approccio è buono per dispositivi che hanno schermi appunto limitati, ma su tablet che dispongono anche di schermi da 10 pollici c'è spazio sullo schermo per gestire più di una

semplice attività alla volta. Per raggiungere lo scopo di visualizzare, ad esempio, su di uno schermo molto ampio, una lista di e-mail e un semplice editor di testo che erano stati progettati per uno schermo più piccolo, in modo da affiancare le due viste, in un contesto pre-Honeycomb è necessario fare ricorso alla duplicazione dei layout per gestire entrambe le logiche, oppure è necessario ricorrere alla tecnica del subclassing; tutto ciò fa aumentare la complessità del codice. Ciò che si vorrebbe avere infatti mediante uno schermo più grande è il beneficio di ottenere più informazioni con pochi click. Ogni porzione di interfaccia utente che può essere usata in più attività, e che non cambia la sua una logica di funzionamento anche su schermi di dimensione differente, risiede in un frammento. I frammenti inoltre permettono di gestire anche modelli di interfacce utente più complessi di quanto si possa ottenere con una semplice visualizzazione condizionata da differenti layout. I frammenti sono una tecnologia opzionale, infatti vanno utilizzati solo per le parti dell'interfaccia utente che dovrebbero apparire in attività diverse, in differenti scenari; quindi le attività che non cambiano mai (ad esempio un help screen o uno splash screen) non dovrebbero far ricorso ai frammenti.

I principali punti di forza di questa tecnologia sono i seguenti.

- La capacità di aggiungere frammenti dinamicamente sulla base dell'interazione con l'utente. Ad esempio, l'applicazione Gmail mostra una lista di cartelle di mail dell'utente: cliccando su di una cartella sullo schermo appaiono le conversazioni in essa presenti, cliccando ulteriormente su una di queste viene visualizzato il testo del messaggio.

- La capacità di animare dinamicamente i frammenti muovendoli sullo schermo; ad esempio sempre in Gmail quando l'utente clicca su di una conversazione, le cartelle delle conversazioni scivolano sulla sinistra, le conversazioni stesse scivolano verso sinistra e si riducono per lasciare più spazio ai messaggi che subentrano sulla destra.

- La gestione automatica del bottone BACK per frammenti dinamici: per esempio quando l'utente preme il tasto BACK mentre sta guardando i messaggi su Gmail, viene ripristinata la situazione precedente con la lista delle conversazioni presenti nella cartella senza che lo sviluppatore debba scrivere del codice ad hoc, neanche per gestire le animazioni inverse.

- La capacità di aggiungere opzioni al menu delle opzioni, quindi all'Action Bar: i menu contestuali vengono gestiti in maniera analoga a un'attività.

◇ La capacità di aggiungere tabs all'Action Bar: l'Action Bar può avere delle tabs e il contenuto di ogni tabs è un frammento.

Se i frammenti fossero disponibili solo per Android 3.0 e versioni successive, probabilmente non potrebbero essere adottati come modelli di programmazione su gran parte dei dispositivi esistenti. Google ha però rilasciato l'Android Compatibility Library (ACL) che è disponibile attraverso l'SDK di Android, la quale permette l'utilizzo dei frammenti nelle precedenti versioni di Android sino alla versione 1.6 inclusa. Poiché la maggior parte dei dispositivi Android montano la versione 1.6 o successiva è possibile usare i frammenti e mantenere la compatibilità all'indietro. In generale, l'uso di questa libreria è pressoché identico all'uso diretto delle classi di Android 3.0. Così, anche se non si è interessati a scrivere applicazioni per tablet, la tecnologia dei frammenti può semplificare lo sviluppo di applicazioni su dispositivi che non siano i tablet.

4.3.1 Quando usare i frammenti

Come anticipato, una delle ragioni primarie per l'utilizzo di un frammento è quella di riusare un pezzo di interfaccia utente e relative funzionalità su diversi dispositivi e schermi; questo è particolarmente vero per i tablet. Un'applicazione tablet è più simile ad un'applicazione desktop che ad una per smartphone, in quanto molte applicazioni desktop hanno un'interfaccia utente multipannello. Nell'esempio precedente della mail si aveva una lista di indirizzi su di una vista e un editor di testo su di un'altra nello stesso istante; questo è facile da gestire quando il dispositivo è in modalità landscape, dove la lista è sulla sinistra e i dettagli sulla destra, ma se l'utente ruota il dispositivo in modalità portrait lo schermo diviene immediatamente più alto che largo: l'ideale sarebbe avere la lista in alto e i dettagli in basso. Ma se l'applicazione è in esecuzione su di uno schermo limitato non c'è spazio per entrambe le attività e l'ideale sarebbe separarle, pur mantenendo la logica di funzionamento invariata. Questa è una circostanza nella quale è consigliabile l'uso dei frammenti. Altre situazioni in cui è preferibile ricorrere ai frammenti sono:

◇ quando è necessario semplificare la gestione (salvataggio e ripristino) dello stato dell'istanza (ad esempio durante il cambio dell'orientamento);

◇ quando l'utente vuole ripristinare stati precedenti della UI mediante pressione del tasto BACK;

◇ quando si effettuano grandi cambiamenti alla UI di un'activity.

4.3.2 Filosofia di progettazione

Android ha introdotto i frammenti nella versione 3.0 (API Level 'Honeycomb'), primariamente per supportare progetti più flessibili e dinamici di UI su schermi più ampi, come i tablet. Poiché lo schermo di un tablet è molto più grande di quello di un cellulare, c'è più spazio per combinare e intercambiare i componenti della UI. I frammenti permettono di realizzare tali progetti senza la necessità di gestire cambiamenti complessi alla gerarchia della view. Suddividendo il layout di un'activity in frammenti risulta possibile modificare l'aspetto dell'activity a runtime e preservare questi cambiamenti in un back stack che è gestito dall'activity. Per esempio, una nuova applicazione può utilizzare un frammento per mostrare una lista di articoli sul lato sinistro dello schermo e un altro frammento per visualizzare i dettagli di un articolo sul lato destro dello schermo: entrambi i frammenti appaiono in un'activity, affiancati, e ogni frammento ha il proprio insieme di metodi di callback del ciclo di vita e gestisce i propri input provenienti dall'utente. Quindi, invece di usare un'activity per selezionare un articolo e un'altra activity per leggere l'articolo, l'utente può selezionare un articolo e leggerlo tutto dentro la stessa activity, come illustrato in Figura 4.22

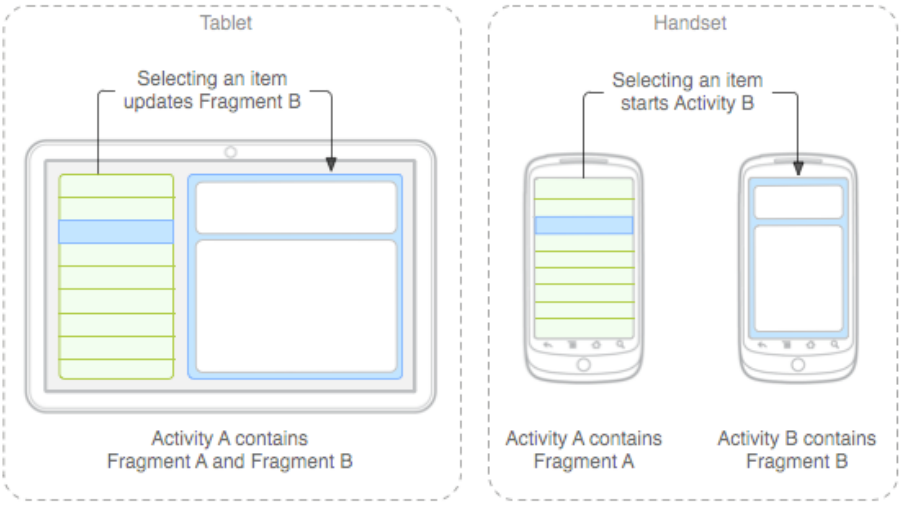

Figura 4.22 – Un esempio di come due moduli UI, che sono tipicamente separati dentro due activity, possano essere combinati in una sola activity usando i frammenti. Immagine tratta dalla documentazione ufficiale di Android http://developer.android.com/guide/topics/fundamentals/fragments.html

Un frammento è un componente modulare e riutilizzabile in un'applicazione. In altre parole, poiché il frammento definisce il suo proprio layout e il suo comportamento usando i propri metodi di callback, si può includere un fram-

mento in più activity. Questo è particolarmente importante perché permette al programmatore di adattare l'esperienza per l'utente della sua applicazione alle diverse dimensioni dello schermo. Per esempio, si possono includere più frammenti in un'activity solo quando la dimensione dello schermo è sufficientemente grande, e quando questo non lo è lanciare activity separate che usano proprio quei frammenti. Facendo sempre riferimento all'applicazione di news considerata, essa può inserire due frammenti in un'activity A, quando è in esecuzione su uno schermo extra largo (per esempio un tablet). Tuttavia, su uno schermo dalle dimensioni normali come un cellulare, non c'è abbastanza spazio per entrambi i frammenti, così l'activity A include solo il frammento per la lista di articoli, e quando l'utente seleziona un articolo, inizia l'activity B, che include il frammento per leggere l'articolo. Quindi, l'applicazione supporta entrambe le organizzazioni suggerite in Figura 4.22.

4.3.3 Il ciclo di vita di un frammento

Il ciclo di vita di un frammento è illustrato nella Figura 4.23 in cui si può notare la presenza dei metodi di base del ciclo di vita di un'activity, oltre ad ulteriori metodi importanti relativi alle interazioni con l'activity che lo contiene e alla generazione della UI.

Nello specifico, i metodi del ciclo di vita di un frammento sono i seguenti.

- `onAttach(Activity activity)`: chiamato dal sistema una volta che il frammento è stato associato alla sua activity

- `onCreate(Bundle savedInstanceState)`: chiamato dal sistema al momento della creazione del frammento. Nell'implementazione di tale metodo dovrebbero essere inizializzati i componenti essenziali del frammento che si vogliono conservare quando il frammento viene messo in pausa o stoppato e poi riesumato.

- `onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)`: chiamato dal sistema quando è tempo per il frammento di disegnare la sua interfaccia utente per la prima volta. Per disegnare una UI per il frammento, è necessario che tale metodo ritorni una View che è la radice del layout del frammento. Si può ritornare null se il frammento non fornisce una UI.

- `onActivityCreated(Bundle savedInstanceState)`: chiamato dal sistema quando l'activity del frammento è stata creata e la gerarchia di view del frammento istanziata.

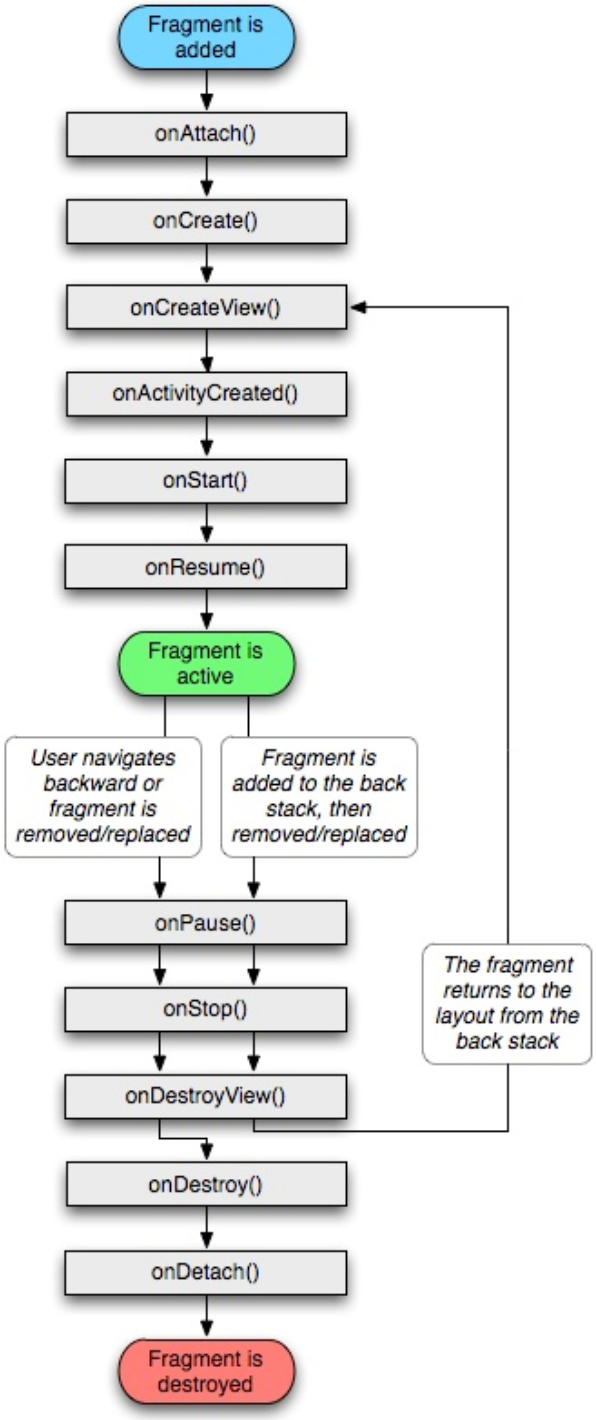

Figura 4.23 – Il ciclo di vita di un frammento (mentre un'activity è in esecuzione). Tratto da http://developer.android.com/guide/topics/fundamentals/fragments.html

- ◇ `onStart()`: chiamato dal sistema per rendere il frammento visibile all'utente con cui però non ha ancora iniziato a interagire.

- ◇ `onResume()`: chiamato dal sistema quando il frammento è visibile dall'utente e in esecuzione, ovvero interagente con l'utente.

- ◇ `onPause()`: chiamato dal sistema quando il frammento non sta più interagendo con l'utente o perché l'activity che lo contiene è stata messa in pausa o perché un operazione di un frammento sta modificando l'activity. È chiamato dal sistema come prima indicazione che l'utente sta lasciando il frammento (tuttavia ciò non sempre significa che il frammento sta per essere distrutto). Questo è di solito il punto in cui dovrebbe effettuare qualche cambiamento che dovrebbe essere mantenuto oltre la sessione utente corrente (perché l'utente non può tornare indietro).

- ◇ `onStop()`: chiamato dal sistema quando il frammento non è visibile all'utente o perché l'activity che lo contiene è stata stoppata o perché un'operazione di un frammento sta modificando l'activity.

- ◇ `onDestroyView()`: chiamato quando la view precedentemente creata con `onCreateView()` sta per essere rimossa. La prossima volta che il frammento dovrà essere visualizzato, sarà creata una nuova view.

- ◇ `onDestroy()`: chiamato dal sistema quando il frammento non è più in uso.

- ◇ `onDetach()`: chiamato quando il frammento non è più collegato alla sua activity.

Tra tutti questi, i principali metodi che dovrebbero essere implementati nelle applicazioni sono `onCreate()`, `onCreateView()` e `onPause()`. Un frammento deve sempre essere inserito in un'activity e il ciclo di vita del frammento è direttamente influenzato dal ciclo di vita dell'activity che lo ospita. Per esempio, quando l'activity passa nello stato di 'pausa' anche tutti i frammenti contenuti in essa passano nello stato di 'pausa', e quando l'activity viene distrutta pure i suoi frammenti vengono distrutti. Tuttavia, mentre un'activity è in esecuzione (si trova cioè nello stato resumed del ciclo di vita) si può manipolare ogni suo frammento in modo del tutto indipendentemente da essa, come aggiungerne altri o rimuoverne qualcuno. Queste transazioni di frammento possono essere salvate in un back stack gestito dall'activity che consente all'utente di navigare all'indietro attraverso i frammenti ripristinandone gli stati precedenti, premendo il tasto BACK.

4.3.4 Creare un frammento

Come già detto, un frammento è una sorta di sotto-attività che ha uno scopo ben preciso e quasi sempre visualizza una interfaccia utente, però non è una estensione di un'attività. Per creare un frammento, si deve estendere la classe `Fragment` (o una delle sue sottoclassi). Tal classe contiene metodi di callback simili a quelli di un'activity come `onCreate()`, `onStart()`, `onPause()`, e `onStop()`. Questo permette di convertire un'applicazione Android esistente in un'applicazione con i frammenti, semplicemente spostando il codice dai metodi di callback delle activity nei corrispondenti metodi di callback dei frammenti. Inoltre è possibile estendere le sottoclassi della classe Fragment che sono le seguenti.

- ◇ `DialogFragment`: un frammento che visualizza una finestra di dialogo flottante sopra la finestra dell'activity. Questa classe è una buona alternativa all'uso dei metodi di aiuto della classe Activity, perché si può incorporare un frammento di dialogo nel back stack dei frammenti gestiti dall'activity, permettendo all'utente di ritornare a un frammento precedente.

- ◇ `ListFragment`: un frammento che visualizza una lista di item che sono gestiti da un adattatore (come un SimpleCursorAdapter), in modo analogo a ListActivity. Tale classe fornisce diversi metodi per gestire una listview, tra i quali alcuni gestori di evento quando l'utente seleziona un item.

- ◇ `PreferenceFragment`: un frammento che visualizza una gerarchia di oggetti di tipo Preference come una lista, in modo analogo a PreferenceActivity. Questo è utile quando si crea un'activity tramite cui l'utente possa settare le proprie preferenze.

- ◇ `WebViewFragment`: un frammento che visualizza una WebView.

Aggiungere un'interfaccia utente

Un frammento fa parte dell'interfaccia utente di un'activity e contribuisce ad essa con un suo proprio layout. Per fornire un layout per un frammento si deve implementare il metodo di callback `onCreateView()` che il sistema Android invoca per visualizzare il layout stesso sullo schermo del dispositivo. Tale metodo deve ritornare una View che è il nodo radice del layout del frammento, il quale rappresenta il corpo del frammento. Il metodo `onCreateView()` ha tre parametri:

1. `LayoutInflater inflater`, usato per caricare un layout per il frammento. Infatti, per restituire un layout tramite `onCreateView()` lo si può caricare da una risorsa layout definita in un apposito file XML. Per fare questo, `onCreateView()` fornisce un oggetto `LayoutInflater`. Precisamente, prima si deve definire la UI del frammento in un file XML e poi fare riferimento ad essa tramite la relativa risorsa.

2. `ViewGroup container`: se non è nullo è il contenitore a cui deve essere associata la UI del frammento.

3. `Bundle savedInstanceState`: se non nullo il frammento sarà ricostruito da uno stato precedentemente salvato.

Per esempio, nel Codice 4.5 c'è una sottoclasse di `Fragment` che carica un layout dal file example_fragment.xml:

```
1 public static class ExampleFragment extends Fragment {
2  @Override
3  public View onCreateView(LayoutInflater inflater, ViewGroup container,
       Bundle savedInstanceState) {
4   return inflater.inflate(R.layout.example_fragment, container, false);
5  }
6 }
```

Codice 4.5 – Fragment che carica un layout dal file example_fragment.xml

Se il frammento è una sottoclasse di `ListFragment`, l'implementazione di default di `onCreateView()` ritorna una ListView, così non c'è necessità di implementare tale metodo.

Nel codice di esempio 4.5, R.layout.example_fragment è un riferimento alla risorsa costituita dal file example_fragment.xml che definisce il layout del frammento salvato nella cartella delle risorse dell'applicazione. Il parametro container passato a `onCreateView()` è il ViewGroup genitore nel quale il layout del frammento sarà inserito. Il parametro `savedInstanceState` è un Bundle che fornisce i dati dell'istanza precedente del frammento, quando il frammento stesso deve essere ripristinato.

Il metodo `inflate()` carica una nuova gerarchia di view da una specifica risorsa XML. Esso ha i seguenti argomenti:

- ◇ *resource*: l'ID della risorsa layout che si desidera caricare

- ◇ *container*: il ViewGroup che deve essere il genitore del layout caricato. È importante passare il container affinché il sistema possa applicare i parametri del layout alla view.

◇ *attachToRoot*: un booleano indicante se il layout debba essere collegato al Viewgroup (il secondo parametro) durante la creazione. (Nel caso in esame, è falso perché il sistema sta già inserendo il layout nel container; passando true si creerebbe un ViewGroup ridondante nel layout finale).

Quando si aggiunge un frammento come una parte del layout di un'activity, esso vive in un ViewGroup dentro la gerarchia delle view dell'activity.

Aggiungere un frammento ad un'activity

Di solito un frammento contribuisce a una porzione di UI dell'activity che la ospita, il quale è inserito come una parte della gerarchia delle view dell'activity. Un frammento può essere inserito nel layout di un'activity in due modi:

◇ Dichiarando il frammento nel file che definisce il layout dell'activity, come un elemento `fragment`. In questo caso si possono specificare le proprietà del layout per il frammento come se fosse una view. Per esempio, un file che descrive un layout per un'activity contenente due frammenti è :

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android
       "
3    android:orientation="horizontal"
4    android:layout_width="match_parent"
5    android:layout_height="match_parent">
6    <fragment android:name="com.example.news.ArticleListFragment"
7      android:id="@+id/list"
8      android:layout_weight="1"
9      android:layout_width="0dp"
10     android:layout_height="match_parent" />
11   <fragment android:name="com.example.news.ArticleReaderFragment"
12     android:id="@+id/viewer"
13     android:layout_weight="2"
14     android:layout_width="0dp"
15     android:layout_height="match_parent" />
16 </LinearLayout>
```

Codice 4.6 – XML usato per dichiarare un frammento

L'attributo `android:name` nel tag fragment specifica la classe Fragment da istanziare nel layout. Quando il sistema crea questo layout per l'activity, esso istanzia ogni frammento specificato e per ognuno invoca il metodo `onCreateView()` per recuperare il relativo layout. Ogni frammento richiede un unico identificatore che il sistema può usare per poi fare riferimento ad esso se l'activity è re-inizializzata oppure per eseguire una transazione, come ad esempio per rimuoverlo. Ci sono tre modi per attribuire un ID a un frammento:

  - fornire l'attributo `android:id` con un ID unico;
  - fornire l'attributo `android:tag` con una stringa unica;
  - usare l'ID della view contenitore, se non si utilizza nessuno dei due precedenti.

- ◇ Aggiungendo il frammento a un ViewGroup esistente dal codice dell'applicazione.

Mentre un'activity è in esecuzione, si possono aggiungere frammenti al suo layout in qualsiasi momento: è semplicemente necessario specificare un ViewGroup in cui piazzare il frammento. Per effettuare modifiche ai frammenti di un'activity (come aggiungere, rimuovere, o rimpiazzare un frammento), si devono utilizzare i metodi della classe `FragmentTransaction`. Si può ottenere un'istanza di `FragmentTransaction` come illustrato nel Codice 4.7:

```
1 FragmentManager fragmentManager = getFragmentManager()
2 FragmentTransaction fragmentTransaction = fragmentManager.
      beginTransaction();
```

Codice 4.7 – Come ottenere un'istanza di FragmentTransaction

Successivamente si può aggiungere un frammento usando il metodo `add()`, specificando il frammento da aggiungere e la view in cui inserirlo, per esempio come nel Codice 4.8:

```
1 ExampleFragment fragment = new ExampleFragment();
2 fragmentTransaction.add(R.id.fragment_container, fragment);
3 fragmentTransaction.commit();
```

Codice 4.8 – Come effettuare un commit attraverso il FragmentManager

Il primo argomento passato al metodo `add()` è il ViewGroup, specificato dall'identificatore di risorsa, nel quale il frammento dovrebbe essere posizionato, e il secondo parametro è il frammento da aggiungere. Successivamente è necessario invocare il metodo `commit()` per far sì che i cambiamenti vengano effettuati.

Aggiungere un frammento senza UI

L'esempio appena discusso mostra come aggiungere un frammento a un'activity allo scopo di fornire una UI; tuttavia non è richiesto che un frammento faccia parte del layout di un'activity: si può anche usare un frammento come un lavoratore invisibile per l'activity, senza presentare una UI addizionale. A prima vista questa funzionalità può sembrare un po' strana, ma di fatto permette

di riutilizzare del codice (quindi un comportamento) che è stato implementato altrove, senza però avere il contributo alla UI; si ricordi che un frammento gode per l'appunto di un proprio ciclo di vita autonomo rispetto all'activity e ciò potrebbe venir utile. Quindi è possibile associare un frammento a un'activity senza renderlo visibile all'utente; per farlo basta invocare il metodo `public FragmentTransaction add(Fragment fragment, String tag)` in cui viene fornita, al posto dell'identificatore della view, una stringa `tag` per poterlo recuperare successivamente. Fornire una stringa è il solo modo per identificare univocamente un frammento che non esibisce una UI. Poiché il frammento aggiunto in questo modo non è associato ad alcuna view del layout dell'activity, non riceve nessuna chiamata a `onCreateView()`, dunque non è necessario implementare tale metodo.

4.3.5 Gestire i frammenti

I frammenti appartenenti a un'attività sono gestiti da un componente descritto dalla classe FragmentManager. Per ottenere un gestore di frammenti basta invocare il metodo `getFragmentManager()` da un'attività o da un frammento ad essa associato. Esso consente di:

- ottenere i frammenti contenuti nell'activity tramite i metodi:
  - `findFragmentById()` per i frammenti che forniscono una UI nel layout dell'activity,
  - `findFragmentByTag()` per i frammenti che non forniscono una UI;
- estrarre i frammenti dal back stack tramite il metodo `popBackStack()`, che in pratica simula la pressione del tasto BACK da parte dell'utente;
- registrare un listener per i cambiamenti avvenuti nel back stack tramite il metodo `addOnBackStackChangedListener()`

Come dimostrato nella precedente sezione, la classe FragmentManager può essere inoltre usata per aprire una FragmentTransaction, che permette di eseguire transazioni come aggiungere o rimuovere frammenti.

4.3.6 Eseguire transazioni di frammento

Un'importante caratteristica che riguarda l'uso dei frammenti in un'activity è la capacità di aggiungere, rimuovere, rimpiazzare ed eseguire altre azioni con essi, in risposta all'interazione con l'utente. Ogni insieme di cambiamenti che vengono effettuati dall'attività è chiamato transazione e la si può eseguire usando le API presenti nella classe FragmentTransaction. Inoltre, ogni

transazione può essere salvata in un back stack che viene gestito dall'attività, il quale permette all'utente di navigare all'indietro attraverso i cambiamenti apportati nei frammenti (in modo analogo alla navigazione all'indietro tra le attività). Per un esempio di come acquisire un'istanza di FragmentTransaction da FragmentManager si faccia riferimento al Codice 4.7

Ogni transazione è un insieme di cambiamenti che si desiderano effettuare nello stesso momento, che possono essere avviati usando metodi come `add()`, `remove()`, e `replace()`. Infine, per eseguire la transazione è necessario chiamare il metodo `commit()`. Però, prima di chiamare quest'ultimo metodo si può invocare il metodo `addToBackStack()` per aggiungere la transazione al back stack delle transazioni di frammento, il quale è gestito dall'attività e permette all'utente di ritornare allo stato precedente del frammento, premendo il tasto BACK. Un esempio di come rimpiazzare un frammento con un altro e preservare lo stato nel back stack è illustrato nel Codice 4.9.

```
1 Fragment newFragment = new ExampleFragment();
2 FragmentTransaction transaction = getFragmentManager().beginTransaction
      ();
3 transaction.replace(R.id.fragment_container, newFragment);
4 transaction.addToBackStack(null);
5 transaction.commit();
```

Codice 4.9 – Rimpiazzare un frammento con un altro

In tale esempio, `newFragment` rimpiazza qualsiasi altro frammento si trovi correntemente (se ce n'è qualcuno) nel contenitore di layout identificato dall'ID R.id.fragment_container. Chiamando `addToBackStack()`, la transazione di rimpiazzo è salvata nel back stack così l'utente può ritornare indietro al precedente frammento premendo il tasto BACK. Se si aggiungono multipli cambiamenti (come `add()` o `remove()`) alla transazione e si chiama il metodo `addToBackStack()`, allora i cambiamenti applicati prima della chiamata a `commit()` sono aggiunti al back stack come una singola transazione e il tasto BACK li annullerà tutti insieme. L'ordine in cui vengono aggiunti i cambiamenti a FragmentTransaction non è importante, eccetto quando si stanno aggiungendo frammenti multipli allo stesso contenitore; in questo caso l'ordine nel quale vengono aggiunti determina l'ordine in cui appaiono nella gerarchia delle view.

Se non si chiama il metodo `addToBackStack()`, quando si esegue una transazione che rimuove un frammento, esso è distrutto e l'utente non può ripristinarlo. Mentre, se si chiama il metodo `addToBackStack()` quando si sta rimuovendo un frammento, allora il frammento è stoppato e sarà riavviato se l'utente preme il tasto BACK. È inoltre possibile, ad ogni transazione, applicare

un'animazione di transizione chiamando il metodo `setTransition()` prima di invocare `commit()`.

Chiamando `commit()` la transazione non viene eseguita immediatamente: essa viene schedulata per essere eseguita nel thread della UI dell'attività (il thread principale) non appena il thread è in grado di fare questo. Comunque, se necessario, è possibile chiamare il metodo `executePendingTransactions()` dal thread della UI per forzare l'immediata esecuzione delle transazioni pendenti, anche se normalmente non è necessario farlo.

Si presti attenzione al fatto che si può effettuare una transazione usando `commit()` solo prima che l'attività abbia salvato il suo stato (quando l'utente lascia l'attività). Se si cerca di effettuarla dopo questo punto verrà sollevata un'eccezione: questo perché lo stato dopo commit può essere perso se l'attività necessita di essere ripristinata. Nelle situazioni in cui è accettabile perdere lo stato, si può usare il metodo `commitAllowingStateLoss()`.

4.3.7 Comunicare con l'attività

Anche se un frammento è implementato come un oggetto che è indipendente da un'attività e che può essere usato dentro attività multiple, una data istanza di un frammento è legata direttamente all'attività che lo contiene. Precisamente, il frammento può accedere all'istanza dell'attività tramite il metodo `getActivity()` ed eseguire facilmente compiti quali trovare una view nel layout dell'attività (es. Codice 4.10).

```
1 View listView = getActivity().findViewById(R.id.list);
```

Codice 4.10 – Recupero della listview dall'XML

Analogamente, l'attività può chiamare metodi nel frammento acquisendo un riferimento a Fragment da FragmentManager, usando il metodo `findFragmentById()` oppure il metodo `findFragmentByTag()`. Per un esempio si veda il Codice 4.11:

```
1 ExampleFragment fragment = (ExampleFragment)getFragmentManager().
      findFragmentById(R.id.example_fragment);
```

Codice 4.11 – Recupero del fragment

Creare callback di evento per l'attività

In alcuni casi, può essere necessario che un frammento condivida eventi con l'attività. Un buon modo per fare questo è definire un interfaccia di callback

dentro il frammento e richiedere che l'attività che lo ospita la implementi. Quando l'attività riceve una callback attraverso l'interfaccia, essa può condividere l'informazione con altri frammenti presenti nel layout se necessario. Per esempio, se una nuova applicazione ha due frammenti in un'attività, uno che mostra una lista di articoli (frammento A) e un altro che visualizza un articolo (frammento B), il frammento A deve dire all'attività quando un item della lista è selezionato così che essa possa comandare al frammento B di visualizzare l'articolo. In questo caso, l'interfaccia OnArticleSelectedListener è dichiarata nel frammento A mediante il Codice 4.12.

```
1 public static class FragmentA extends ListFragment {
2  ...
3  // Container Activity must implement this interface
4  public interface OnArticleSelectedListener {
5   public void onArticleSelected(Uri articleUri);
6  }
7  ...
8 }
```

Codice 4.12 – Interfaccia OnArticleSelectedListener

L'attività che ospita il frammento implementa l'interfaccia OnArticleSelectedListener e il metodo `onArticleSelected()` per notificare il frammento B dell'evento dal frammento A. Per essere sicuri che l'attività che contiene i frammenti implementi l'interfaccia il metodo di callback `onAttach()` del frammento A (che il sistema invoca quando sta aggiungendo il frammento all'attività) istanzia un'oggetto da OnArticleSelectedListener, come da Codice 4.13:

```
1  public static class FragmentA extends ListFragment {
2   OnArticleSelectedListener mListener;
3   ...
4   @Override
5   public void onAttach(Activity activity) {
6    super.onAttach(activity);
7    try {
8     mListener = (OnArticleSelectedListener) activity;
9    } catch (ClassCastException e) {
10    throw new ClassCastException(activity.toString() + " must implement
          OnArticleSelectedListener");
11   }
12  }
13  ...
14 }
```

Codice 4.13 – Metodo `onAttach()`

Se l'attività non ha implementato l'interfaccia, allora il frammento solleva l'eccezione ClassCastException. In caso di successo, il membro mListener man-

tiene un riferimento all'implementazione dell'attività di OnArticleSelectedListener, così che il frammento A può condividere eventi con l'attività invocando i metodi definiti dall'interfaccia OnArticleSelectedListener. Per esempio, se il frammento A è un'estensione di ListFragment, ogni volta che l'utente clicca su un item della lista, il sistema chiama il metodo `onListItemClick()` nel frammento, che poi chiama il metodo `onArticleSelected()` per condividere l'evento con l'attività,come da Codice 4.14.

```
1  public static class FragmentA extends ListFragment {
2   OnArticleSelectedListener mListener;
3   ...
4   @Override
5   public void onListItemClick(ListView l, View v, int position, long id)
        {
6    // Append the clicked item's row ID with the content provider Uri
7    Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT_URI,
        id);
8    // Send the event and Uri to the host activity
9    mListener.onArticleSelected(noteUri);
10  }
11  ...
12 }
```

Codice 4.14 – Codice per `onListItemClick()`

Il parametro id passato a `onListItemClick()` è l'ID della riga dell'item cliccato, che l'attività (o altri frammenti) usano per il fetch dell'articolo dal content provider dell'applicazione.

Gestione delle opzioni

I frammenti possono fornire opzione all'OptionMenu dell'attività (e, conseguentemente, all'Action Bar) implementando il metodo `onCreateOptionsMenu()`. Affinché questo metodo venga invocato, si deve chiamare il metodo `setHasOptionsMenu()` durante `onCreate()`, per indicare che il frammento intende aggiungere item all'OptionsMenu (altrimenti, il frammento non riceverà una chiamata a `onCreateOptionsMenu()`). Tutti gli item che poi verranno aggiunti all'OptionMenu dal frammento saranno posti negli item del menu esistente. Il frammento inoltre riceve callback al metodo `onOptionsItemSelected()` quando un item del menu è selezionato.

Inoltre è possibile registrare una view nel layout del frammento per fornire un context menu invocando il metodo `registerForContextMenu()`. Quando l'utente apre il context menu, il frammento riceve una chiamata a `onCreateContextmenu()`. Quando l'utente seleziona un item, il frammento riceve una chiamata a `onContextItemSelected()`.

In realtà l'attività è la prima a ricevere la callback quando l'utente seleziona un elemento del menu. Se la callback dell'elemento selezionato all'interno dell'activity non gestisce tale elemento, allora l'evento è passato alla callback del frammento; questo è vero per l'OptionMenu e ContextMenu.

4.3.8 Gestire il ciclo di vita di un frammento

Come un'attività, un frammento può esistere in 3 stati:

- ◇ *Resumed*: il frammento è visibile nell'attività in esecuzione.
- ◇ *Paused*: un'altra activity è in primo piano, ma l'attività in cui il frammento vive è ancora visibile (l'attività in primo piano è parzialmente trasparente o non copre l'intero schermo).
- ◇ *Stopped*: il frammento non è visibile; o l'attività che contiene il frammento è stata stoppata, o il frammento è stato rimosso dall'attività ma aggiunto al back stack. Un frammento stoppato è ancora vivo (lo stato è conservato dal sistema). Comunque, non è visibile all'utente e sarà ucciso se l'activity viene uccisa.

Inoltre, come per un'attività, è possibile conservare lo stato di un frammento usando un Bundle, nel caso che il processo dell'attività sia ucciso e si abbia la necessità di memorizzare lo stato quando l'attività viene ri-creata. Si può salvare lo stato durante la callback del frammento `onSaveInstanceState()` e memorizzarlo durante `onCreate()`, `onCreateView()`, o `onActivityCreated()`.

La differenza più significativa nel ciclo di vita tra un'attività e un frammento è il modo in cui sono memorizzati nel rispettivo back stack. Un'attività è piazzata dentro un back stack di attività che è gestito dal sistema quando è stoppata, per default (così che l'utente può navigare indietro tramite il tasto BACK). Un frammento, invece, è piazzato dentro un back stack gestito dall'attività che lo ospita, e solo quando è richiesto esplicitamente che l'istanza sia salvata invocando il metodo `addToBackStack()` durante una transazione che rimuove il frammento. A parte questo, gestire il ciclo di vita di un frammento è veramente simile a gestire il ciclo di vita dell'attività. Così, le stesse 'pratiche' per gestire il ciclo di vita di un'attività possono essere applicate ai frammenti.

Coordinazione con il ciclo di vita di un'attività

Il ciclo di vita dell'attività in cui vive un frammento influenza direttamente il ciclo di vita del frammento stesso, al punto che ogni callback del ciclo di vita

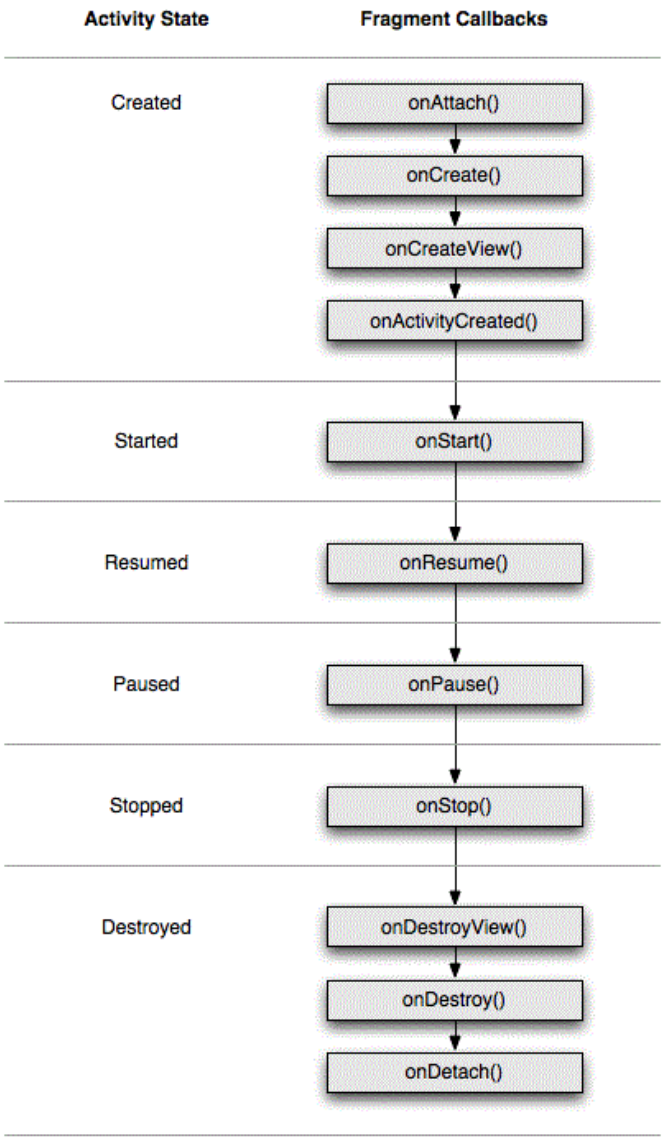

Figura 4.24 – Influenza del ciclo di vita dell'attività sul ciclo di vita del frammento. Immagine tratta dalla documentazione uffciale di Android http://developer.android.com/guide/topics/fundamentals/fragments.html

per un'attività risulta in una callback simile per ogni frammento. Per esempio, quando l'attività riceve `onPause()`, ogni frammento nell'attività riceve `onPause()`.

Comunque, i frammenti hanno pochi metodi di callback extra che manipolano un'unica interazione con l'attività allo scopo di eseguire azioni come costruire e distruggere la UI di un frammento. Questi metodi di callback addizionali sono: `onAttach()`, `onCreateView()`, `onActivityCreated()`, `onDestroyView()` e `onDetach()`.

Il flusso del ciclo di vita di un frammento, che è influenzato dal ciclo di vita dell'attività che lo ospita, è illustrato in Figura 4.24, nella quale si può vedere come ogni stato successivo dell'attività determina quali metodi di callback un frammento può ricevere. Per esempio, quando l'attività ha ricevuto il suo metodo di callback `onCreate()`, un frammento contenuto nell'attività riceve non più di un metodo di callback `onActivityCreate()`.

Una volta che l'attività raggiunge lo stato Resumed, si possono aggiungere e rimuovere liberamente frammenti all'attività. Quindi, solo mentre l'attività è nello stato Resumed il ciclo di vita di un frammento può cambiare indipendentemente.

Capitolo 5

Applicazione di esempio per tablet

5.1 Funzionamento

PER illustrare il funzionamento dei frammenti è stata realizzata una piccola applicazione che permette di effettuare la ricerca di un articolo sul sito 'www.ciao.it'. Dopo aver inserito il nome dell'articolo desiderato nell'apposita casella di testo editabile ed aver premuto il pulsante 'Cerca', come da Figura 5.1, appare una nuova attività che gestisce due frammenti. Il primo frammento appare sulla parte sinistra dello schermo e riporta una lista di siti; il secondo frammento appare invece sulla parte destra, affiancato al primo, e visualizza alcuni dettagli relativi al sito selezionato sulla sinistra: in particolare il prezzo, un'immagine e un link. Cliccando sul prezzo appare poi a pieno schermo il sito in cui è possibile acquistare l'articolo. Si noti che il comportamento sopra descritto è relativo ad un tablet tenuto in modalità 'landscape'; qualora il tablet fosse tenuto in 'portrait', per ragioni di spazio non sarebbe possibile affiancare i due frammenti, motivo per cui, con poco sforzo a livello di programmazione, è stato possibile fare in modo che i due frammenti siano visualizzati uno dopo l'altro.

5.2 Realizzazione

L'applicazione è strutturata principalmente in due parti: il recupero delle informazioni dal sito web menzionato sopra e la gestione dei frammenti.

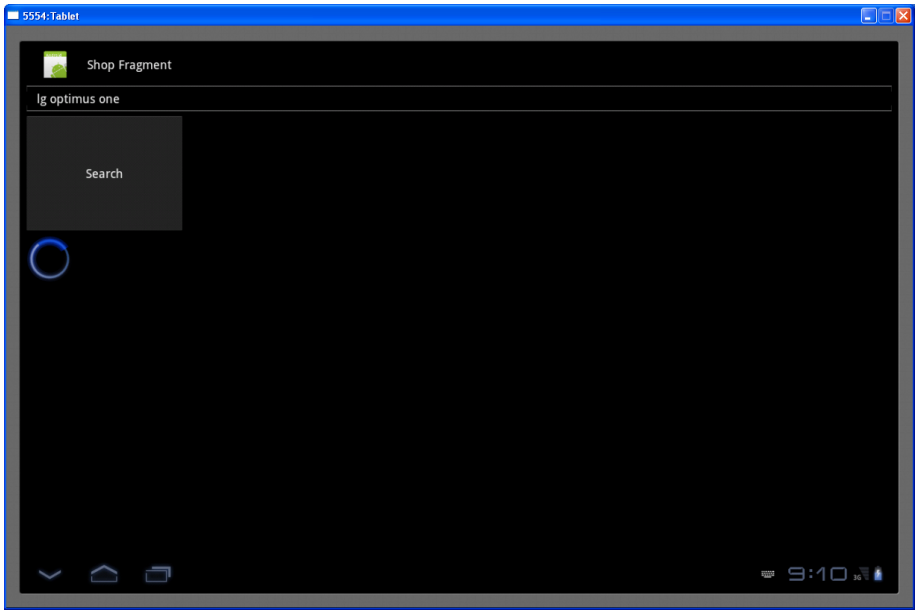

Figura 5.1 – Ricerca di un prodotto, nel caso in esame un cellulare

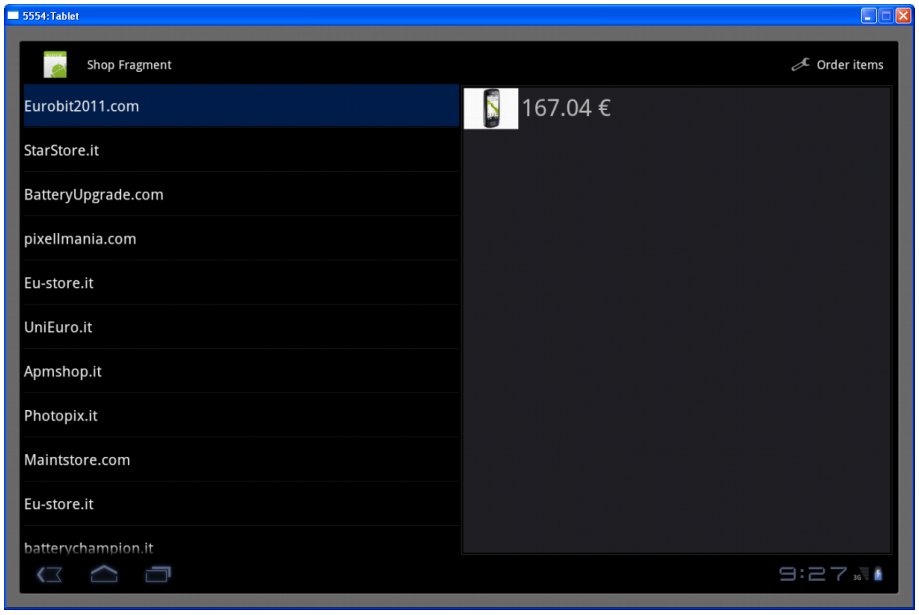

Figura 5.2 – Risultato di una ricerca di un prodotto

Figura 5.3 – Dopo aver cliccato su un prezzo appare a pieno schermo il sito in cui è possibile acquistare l'articolo

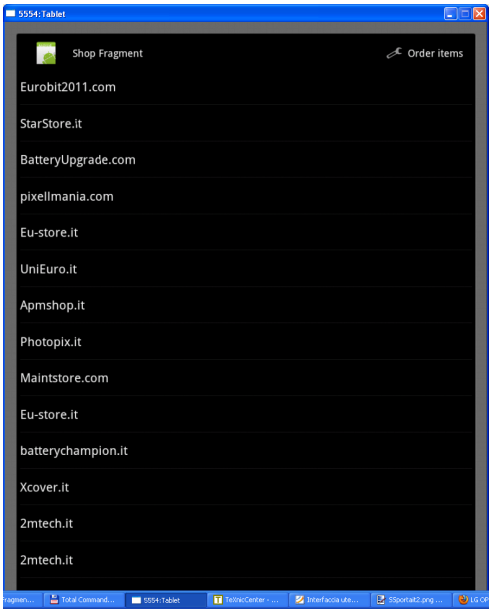

Figura 5.4 – Lista risultati con l'applicazione in portrait

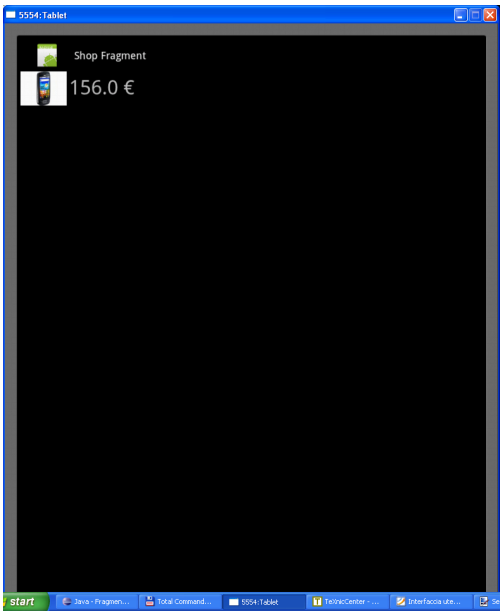

Figura 5.5 – Articolo selezionato con l'applicazione in portrait

5.2.1 Recupero delle informazioni

Una volta che l'utente ha digitato il nome dell'oggetto da ricercare, è necessario eseguire una richiesta al web server di 'ciao'; nel frattempo viene visualizzata una progress bar per segnalare all'utente di attendere. Innanzitutto è necessario completare l'URL di partenza per le ricerche 'http://www.ciao.it/sr/q' con quanto scritto dall'utente. Si è visto (sperimentando con un browser) che gli spazi bianchi, nella ricerca, vanno sostituiti con l'operatore di concatenazione tra stringhe (cioè con il carattere '+'). Una volta ottenuto l'URL finale, è necessario avviare una richiesta HTTP (nel protocollo chiamata GET); fortunatamente Android dispone della classe `HttpClient` che svolge proprio questa funzione, per cui non è necessario farsi carico dell'apertura di un socket e del recupero del testo HTML. È però necessario aggiungere un permesso di accesso ad Internet all'interno del manifest dell'applicazione in modo che non venga sollevata un'eccezione a runtime. Siccome una richiesta HttpClient può essere un'operazione lenta, Android non permette la sua esecuzione all'interno di un thread che gestisce un'interfaccia utente, generando un'apposita eccezione se il programmatore prova a farlo: per esempio lanciando direttamente la richiesta HTTP dal metodo `OnCreate()` di un attività viene immediatamente sollevata tale eccezione. Per questa ragione è necessario che la richiesta HTTP venga effettuata da un thread separato da quello della UI; per semplificare la sincronizzazione fra thread Android mette a disposizione la classe `AsyncTask` che aiuta

il programmatore nella gestione dell'operazione asincrona su un thread distinto. In pratica, sovrascrivendo il metodo `doInBackground()`, invocabile mediante una chiamata al metodo `execute()` dalla UI, si possono realizzare operazioni asincrone su un thread distinto, mentre gli aggiornamenti, anche parziali, all'interfaccia utente vengono gestiti dai metodi `onProgressUpdate()` e `onPostExecute()` (dato che gli oggetti dell'interfaccia utente non possono assolutamente essere manipolati da thread diversi da quello della UI). Per agevolare il passaggio di informazioni fra i differenti thread (UI e Task asincrono) si usano tre distinti parametri:

- uno per gli argomenti iniziali
- uno per gli argomenti finali
- uno per gli aggiornamenti (ad esempio per aggiornare una progress bar)

Al fine di ottenere la massima flessibilità, Android definisce i tipi dei tre parametri ricorrendo all'uso delle 'Generics' di Java. In pratica, sarà quindi necessario definire esplicitamente il tipo (la classe) di ognuno di questi tre parametri (o ricorrere alla classe 'Void' se un parametro non è usato).

Dato che le richieste HTTP coinvolgono anche altre risorse (le immagini) si è dapprima creato un metodo di base che restituisce un InputStream, utilizzato per il recupero del testo e delle immagini (Codice 5.1).

```
1
2  InputStream in = null;
3  HttpClient client = new DefaultHttpClient();
4  HttpGet request = new HttpGet();
5  try {
6          request.setURI(new URI(url));
7  }
8  catch (URISyntaxException e) {
9          return null;
10 }
11
12 HttpResponse response = null;
13 try {
14         response = client.execute(request);
15 }
```

Codice 5.1 – Esecuzione di una richiesta Http

Una volta recuperato l'InputStream, è necessario estrarre le singole linee di testo dell'HTML. Inizialmente si era pensato di utilizzare un parser XML per il recupero veloce delle informazioni dall'HTML, essendo l'HTML un particolare tipo di XML utilizzato per descrivere contenuti statici. Purtroppo l'HTML utilizzato dal sito non è un XML sintatticamente valido in quanto molti tag di chiusura sono stati omessi e quindi non è stato possibile far ricorso al parser

XML standard fornito da Android ma è stato necessario effettuare una scansione 'a mano', leggendo linea per linea dall'InputStream. Dopo un rapido esame dell'HTML scaricato dal sito mediante web client, sono stati individuati i tag utili per il recupero delle informazioni da visualizzare successivamente; solamente le linee di testo HTML di interesse vengono estratte dall'InputStream e sono memorizzate in un array di stringhe. Successivamente questo array viene ulteriormente elaborato per estrarre le informazioni vere e proprie; anche questa operazione viene eseguita all'interno dell'AsyncTask dato che rischierebbe di rallentare troppo la UI. Dall'analisi delle stringhe si ottengono il prezzo, il link al sito e quello all'immagine. Per recuperare l'immagine stessa è necessario, come detto, effettuare un'ulteriore richiesta HTTP, motivo per cui il parsing deve necessariamente risiedere all'interno dell'AsyncTask. Una volta recuperate tutte le informazioni di interesse mediante una serie di metodi ausiliari, queste vengono inserite in un'istanza della classe `RetrievedInfo` che contiene una lista di istanze della classe `Offer`, la quale al suo interno effettivamente memorizza il link, il prezzo e l'immagine stessa. Prima di descrivere il comportamento dei frammenti, si noti che i link esposti sul sito 'ciao' verso i prodotti in offerta non sono dei link diretti; in altre parole il link a un prodotto è in realtà un link al sito 'ciao' stesso che poi rimanda alla pagina del prodotto. Probabilmente questo comportamento serve al sito 'ciao' per monitorare il numero di click verso una singola offerta; ad ogni modo l'applicazione realizzata evita questa forma di controllo recuperando il link diretto.

5.2.2 Gestione dei frammenti

La classe `FragmentLayout` si occupa di gestire i due frammenti veri e propri. Quando il dispositivo è in modalità 'landscape', l'XML che descrive il layout contiene un tag di tipo `fragment` e un tag di tipo `FrameLayout`.

Il tag `fragment`, mediante l'attributo `class`, fa riferimento alla classe interna `TitlesFragment` (contenuta nella classe `FragmentLayout`) che si occupa di gestire la lista di offerte che appare sulla sinistra dello schermo: infatti `TitlesFragment` estende la classe `ListFragment`. Il tag `FrameLayout` definisce un generico contenitore di oggetti grafici, infatti è stato progettato per disporre un'area sullo schermo per visualizzare un singolo oggetto; il suo contenuto verrà gestito al momento della selezione di un elemento della lista di sinistra. L'associazione tra il FrameLayout e il suo effettivo contenuto (il frammento) è effettuata alla linea 18 del Codice 5.3. Il frammento `TitlesFragment` ha principalmente due compiti: il primo è quello di recuperare la lista di offerte e visualizzarle; ciò viene semplicemente implementato mediante un adattatore richiamato dal metodo `setListAdapater()`.

```
1
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android
      "
3   android:orientation="horizontal"
4   android:layout_width="match_parent" android:layout_height="
        match_parent">
5
6   <fragment class="dei.unipd.fragments.FragmentLayout$TitlesFragment"
7       android:id="@+id/titles" android:layout_weight="1"
8       android:layout_width="0px" android:layout_height="match_parent" /
            >
9
10  <FrameLayout android:id="@+id/details" android:layout_weight="1"
11      android:layout_width="0px" android:layout_height="match_parent"
12      android:background="?android:attr/detailsElementBackground" />
13
14 </LinearLayout>
```

Codice 5.2 – XML dell'attività FragmentLayout

Il secondo compito, quello più importante, consiste nell'intercettare l'evento di click su di un elemento della lista, aggiornando il frammento di destra. Per fare questo il metodo dapprima recupera l'istanza corrente della classe `DetailsFragment` mediante il FragmentManager, poi, qualora il frammento di destra visualizzi i dettagli di un altro prodotto (o nessun prodotto) crea una nuova istanza di `DetailsFragment` passandogli l'indice dell'offerta da visualizzare.

Per cambiare lo stato del frammento di destra non è sufficiente crearne una nuova istanza ma è necessario invocare il FragmentManager, impostare la transizione al nuovo frammento e infine eseguire il cambiamento. Qualora il dispositivo si trovi in posizione 'portrait' il flag mDualPanel è falso: in tal caso viene lanciata un'istanza della classe interna `DetailsActivity`; quest'attività a sua volta rilancia il frammento `DetailsFragment` facendo così in modo che occupi l'intero schermo. Attraverso i frammenti si riescono quindi a gestire i dettagli dell'offerta sia occupando metà schermo (quando il dispositivo è orizzontale) sia occupando l'intero schermo (quando il dispositivo è verticale); il vantaggio di questo approccio sta nel fatto che la classe `DetailsFragment` non dev'essere adattata o modificata per supportare entrambe le modalità. Ovviamente è necessario fornire un XML apposito per la visualizzazione in verticale, in cui vi è solo riferimento al frammento di sinistra.

Il tag `FrameLayout` dell'XML in modalità landscape fa invece riferimento alla classe `DetailsFragment` che appunto può occupare metà schermo in modalità landscape o lo schermo intero in portrait. Il metodo più importante di tale classe è sicuramente `onCreateView()` che restituisce una vista con i dettagli dell'offerta.

```
1
2  public void onListItemClick(ListView l, View v, int position, long id)
       {
3  showDetails(position);
4  }
5
6  void showDetails(int index) {
7   mCurCheckPosition = index;
8
9   if (mDualPane) {
10   getListView().setItemChecked(index, true);
11   DetailsFragment details = (DetailsFragment)
12    getFragmentManager().findFragmentById(R.id.details);
13   if (details == null || details.getShownIndex() != index) {
14    details = DetailsFragment.newInstance(index);
15    FragmentTransaction ft = getFragmentManager().beginTransaction();
16    ft.replace(R.id.details, details);
17    ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
18    ft.commit();
19   }
20  }
21  else {
22   Intent intent = new Intent();
23   intent.setClass(getActivity(), DetailsActivity.class);
24   intent.putExtra("index", index);
25   startActivity(intent);
26  }
27 }
```

Codice 5.3 – Gestione del frammento di destra

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:layout_width="match_parent" android:layout_height="
      match_parent">
3         <fragment class="dei.unipd.fragments.FragmentLayout$
              TitlesFragment "android:id="@+id/titles"
              android:layout_width="match_parent" android:layout_height="
              match_parent" />
4 </FrameLayout>
```

Codice 5.4 – XML dell'attività FragmentLayout in modalità portrait

```
1
2  public View onCreateView(LayoutInflater inflater, ViewGroup container,
3   Bundle savedInstanceState) {
4  if (container == null) {
5   return null;
6  }
7
8  ScrollView scroller = new ScrollView(getActivity());
9  TextView text = new TextView(getActivity());
10 int padding = (int)TypedValue.applyDimension(TypedValue.
       COMPLEX_UNIT_DIP, 4, getActivity().getResources().getDisplayMetrics
       ());
11 text.setPadding(padding, padding, padding, padding);
12 text.setText(MainActivity.ShopInfo.getPrice(getShownIndex())+"€");
13 text.setOnClickListener(new OnClickListener() {
14
15  @Override
16  public void onClick(View v) {
17   Uri uri = Uri.parse(MainActivity.ShopInfo.getUrl(getShownIndex()));
18   Intent intent = new Intent(Intent.ACTION_VIEW,uri);
19   startActivity(intent);
20  }
21 });
22 ImageView imgView = new ImageView(this.getActivity());
23 imgView.setImageDrawable(MainActivity.ShopInfo.getImage(getShownIndex()
       ));
24
25 LinearLayout ll = new LinearLayout(getActivity());
26 ll.addView(imgView);
27 ll.addView(text);
28 scroller.addView(ll);
29 return scroller;
30 }
```

Codice 5.5 – Creazione della View all'interno del frammento di destra

Dato che l'offerta è costituita da un'immagine e da un prezzo (cioè un testo), e visto che non si conoscono a priori le lunghezze e le dimensioni di tali oggetti, si fa ricorso ad una ScrollView per adattarle alla grandezza dello schermo. Immagine e prezzo devono essere visualizzati affiancati orizzontalmente, perciò si è fatto ricorso ad un LinearLayout che le contiene. Infine, per permettere di aprire in una WebView il sito dell'offerta, è necessario fare ricorso ad un ClickListener sul testo.

Action Bar

Nell'Action Bar è visibile un'opzione che permette di ordinare i risultati trovati per ordine crescente di prezzo. Per realizzare ciò si è seguito quanto visto con il Codice 4.4.

Bibliografia

[AptIf] C. Pelliccia PuntoInformatico, *Android Programming*, Edizioni Master, 2009

[Hello] E. Brunette, *Hello, Android: Introducing Google's Mobile Development Platform*, Third Edition, 2010

[LoMa] E. Lopez, D. Magnani *Cos'è Android? La storia del sistema operativo mobile di Google*
http://www.androiditaly.com/articoli/speciali/189-cose-android-la-storia-del-sistema-operativo-mobile-di-google.html

[ApGe] sito appgenius *Android Honeycomb 3.0: tutte le novità introdotte*
http://www.appgenius.it/2011/02/03/android-honeycomb-3-0-tutte-le-novita-introdotte/

[NeNw] sito netbooknews *Android 3.0 Honeycomb: anteprima delle nuove caratteristiche lato utente*
http://www.netbooknews.it/android-3-0-honeycomb-anteprima-delle-nuove-caratteristiche-lato-utente/

[Wik1] sito wikipedia *Android*
http://it.wikipedia.org/wiki/Android

[Wik2] sito wikipedia *POST WIMP*
http://it.wikipedia.org/wiki/Post-WIMP

[Wik3] sito wikipedia *Graphical User Interface*
http://it.wikipedia.org/wiki/Graphical_user_interface

[Wik4] sito wikipedia *Interfaccia grafica utente*
http://it.wikipedia.org/wiki/Interfaccia_grafica_utente

[Fant] Slide Corso Sistemi Embedded A.A. 2010 2011 *Embedded Systems*

[Andr] Android developer's guide
http://developer.android.com/guide/index.html

[Pro] S. Komatineni, D. MacLean , S. Hashimi, *Pro Android 3*, Apress, 2011

[Beg] M. Murphy, *Beginning Android 3*, Apress, 2011

[Dam] A. vanDam, *POST-WIMP User Interfaces*, Communication of the ACM, Febrauary 1997

[Deb] Debian wiki: User Interface, http://wiki.debian.org/it/User_interface

[Urp] Comunicazione Pubblica Usabile: Storia delle Interfaccie, http://www.urp.it/cpusabile/index8d7c.html

Appendice A

Codice

A.1 MainActivity.java

```
package dei.unipd.fragments;

import android.app.Activity;
import android.content.Intent;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ProgressBar;
import android.widget.Toast;

public class MainActivity extends Activity {

 // reference to the editbox
 private EditText mEditText;
 // reference to the progressbar
 private ProgressBar mProgressBar;

 // Asynchronous task for retrieve and parse of results
 private HttpGetTask mHttpGetTask;

 // static reference to the result of HTTP get and HTML parsing
```

```
// (for other activities)
public static RetrievedInfo ShopInfo;

@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 // set main layout
 setContentView(R.layout.mainlayout);
 // retrieve some view
 mEditText = (EditText) this.findViewById(R.id.editText1);
 mProgressBar = (ProgressBar) this.findViewById(R.id.progressBar1);
 mProgressBar.setVisibility(ProgressBar.INVISIBLE);
 Button bt = (Button) this.findViewById(R.id.button1);

 // listener for click
 bt.setOnClickListener(new OnClickListener() {

  @Override
  public void onClick(View v) {
   // retrieve text
   String sText = mEditText.getText().toString();
   // replace white space with '+' since space are not allowed
   // inside a URL
   sText = sText.replace(' ', '+');
   mHttpGetTask = new HttpGetTask();
   // this will cause onPostExecute() to run
   // since we cannot start operations involving the net from UI
   // thread (this thread)
   mHttpGetTask.execute("http://www.ciao.it/sr/q-" + sText);
   // display a progressbar
   mProgressBar.setVisibility(ProgressBar.VISIBLE);

  }
 });

 //
}

@Override
protected void onDestroy() {
```

```
 super.onDestroy();
}

@Override
protected void onPause() {
 super.onPause();
}

private class HttpGetTask extends AsyncTask<String, Void,
RetrievedInfo> {

 // this methods do a lot of things in background
 // (it takes a very long time):
 // execute HttpGet request
 // parse HTML
 protected RetrievedInfo doInBackground(String... url) {
  // executes HTTP get retrieving HTML
  String[] sHTTP = Util.executeHtmlHttpGet(url[0]);
  if (sHTTP == null)
   return null;
  // Ok, parse the HTML and save result inside an instance of
  // RetrievedInfo
  RetrievedInfo info = new RetrievedInfo();
  Util.parse(sHTTP, info);
  // return info to onPostExecute()
  return info;
 }

 protected void onProgressUpdate(Void... progress) {
  // do nothing... (back to UI thread: here we could
  // update a progress bar)
 }

 protected void onPostExecute(RetrievedInfo result) {
  // we come back to UI thread
  // ok we can update the UI accoriding to the result
  if (result == null) {
   // something goes wrong... just show a simple toast
   Toast t = Toast.makeText(MainActivity.this, "NO CONNECTION!",
     Toast.LENGTH_LONG);
```

```
    t.show();
    // and remove the progressbar
    mProgressBar.setVisibility(ProgressBar.INVISIBLE);
    return;
   }
   // ok now its time to start the fragments
   Intent i = new Intent(MainActivity.this, FragmentLayout.class);
   // make static reference to the result (for other activities)
   ShopInfo = result;
   // remove the progressbar
   mProgressBar.setVisibility(ProgressBar.INVISIBLE);
   // start fragments
   startActivity(i);
  }
 }

}
```

A.2 FragmentLayout.java

```
package dei.unipd.fragments;

import android.app.Activity;
import android.app.Fragment;
import android.app.FragmentTransaction;
import android.app.ListFragment;
import android.content.Intent;
import android.content.res.Configuration;
import android.net.Uri;
import android.os.Bundle;
import android.util.TypedValue;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
```

```
import android.widget.LinearLayout;
import android.widget.ListView;
import android.widget.ScrollView;
import android.widget.TextView;

public class FragmentLayout extends Activity {

 final static String CUR_CHOICE = "curChoice";

 /** Called when the activity is first created. */
 @Override
 public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.fragment_layout);
 }

 /** To reference */
 private static TitlesFragment TitlesFragment = null;

 public static class TitlesFragment extends ListFragment {
  boolean mDualPane;
  int mCurCheckPosition = 0;

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
   super.onActivityCreated(savedInstanceState);

   // Populate list with the shops
   setListAdapter(new ArrayAdapter<String>(getActivity(),
     android.R.layout.simple_list_item_activated_1,
     MainActivity.ShopInfo.getAllShops()));

   TitlesFragment = this;

   // Check to see if we have a frame in which to embed the details
   // fragment directly in the containing UI.
   View detailsFrame = getActivity().findViewById(R.id.details);
   mDualPane = detailsFrame != null
     && detailsFrame.getVisibility() == View.VISIBLE;
```

```
 if (savedInstanceState != null) {
  // Restore last state for checked position.
  mCurCheckPosition = savedInstanceState.getInt(CUR_CHOICE, 0);
 }

 if (mDualPane) {
  // In dual-pane mode, the list view highlights the selected
  // item.
  getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
  // Make sure our UI is in the correct state.
  showDetails(mCurCheckPosition, false);
 }
}

@Override
public void onSaveInstanceState(Bundle outState) {
 super.onSaveInstanceState(outState);
 // save selected item
 outState.putInt(CUR_CHOICE, mCurCheckPosition);
}

@Override
public void onListItemClick(ListView l, View v, int position,
 long id) {
 showDetails(position, false);
}

/**
 * Helper function to show the details of a selected item,
 * either by displaying a fragment in-place in the current UI,
 * or starting a wholenew activity in which it is displayed.
 *
 * @param index
 *            : the index of the item to show
 * @param ForceUpdate
 *            : to force the update (useful when you have to
 *            update the list since it has been sorted)
 */
public void showDetails(int index, boolean forceUpdate) {
 mCurCheckPosition = index;
```

```
  if (mDualPane) {
   // We can display everything in-place with fragments, so update
   // the list to highlight the selected item and show the data.
   getListView().setItemChecked(index, true);

   // Check what fragment is currently shown, replace if needed.
   DetailsFragment details = (DetailsFragment) getFragmentManager()
     .findFragmentById(R.id.details);
   // right fragment is created if
   // 1) it does NOT exists
   // 2) it exists but show another item
   // 3) forced by @param forceUpdate
   if (details == null || details.getShownIndex() != index
     || forceUpdate) {
    // Make new fragment to show this selection.
    details = DetailsFragment.newInstance(index);
    // Execute a transaction, replacing any existing fragment
    // with this one inside the frame.
    FragmentTransaction ft = getFragmentManager()
      .beginTransaction();
    ft.replace(R.id.details, details);
    ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
    ft.commit();
   }

  } else {
   // Otherwise we need to launch a new activity to display
   // the dialog fragment with selected text.
   Intent intent = new Intent();
   intent.setClass(getActivity(), DetailsActivity.class);
   intent.putExtra(DetailsFragment.INDEX, index);
   startActivity(intent);
  }
 }
}

public static class DetailsFragment extends Fragment {

 public static final String INDEX = "index";
```

```
/**
 * Create a new instance of DetailsFragment, initialized to
 * show the text at 'index'.
 */
public static DetailsFragment newInstance(int index) {
 DetailsFragment f = new DetailsFragment();

 // Supply index input as an argument.
 Bundle args = new Bundle();
 args.putInt(INDEX, index);
 f.setArguments(args);

 return f;
}

public int getShownIndex() {
 return getArguments().getInt(INDEX, 0);
}

@Override
public View onCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
 if (container == null) {
  // We have different layouts, and in one of them this
  // fragment's containing frame doesn't exist. The fragment
  // may still be created from its saved state, but there is
  // no reason to try to create its view hierarchy because it
  // won't be displayed. Note this is not needed -- we could
  // just run the code below, where we would create and return
  // the view hierarchy; it would just never be used.
  return null;
 }

 // scroll view to hold all the elements
 ScrollView scroller = new ScrollView(getActivity());
 // A text view for the price
 TextView text = new TextView(getActivity());
 int padding = (int) TypedValue.applyDimension(
   TypedValue.COMPLEX_UNIT_DIP, 4, getActivity()
```

```
        .getResources().getDisplayMetrics());
    text.setPadding(padding, padding, padding, padding);
    text.setText(MainActivity.ShopInfo.getPrice(getShownIndex())
    + " euro");
    text.setTextSize(text.getTextSize() * 2.5f);

    // click listener to open a webview when clicking on the price
    text.setOnClickListener(new OnClickListener() {

     @Override
     public void onClick(View v) {
      // retrieve the URI
      Uri uri = Uri.parse(MainActivity.ShopInfo
        .getUrl(getShownIndex()));
      // Start a web view
      Intent intent = new Intent(Intent.ACTION_VIEW, uri);
      startActivity(intent);

     }
    });

    // load the image
    ImageView imgView = new ImageView(this.getActivity());
    imgView.setImageDrawable(MainActivity.ShopInfo
      .getImage(getShownIndex()));
    // we need a linear layout to manage an image and a price
    LinearLayout ll = new LinearLayout(getActivity());
    ll.addView(imgView);
    ll.addView(text);
    scroller.addView(ll);
    return scroller;
   }
}

public static class DetailsActivity extends Activity {

 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
```

```
   if (getResources().getConfiguration().orientation ==
   Configuration.ORIENTATION_LANDSCAPE) {
    // If the screen is now in landscape mode, we can show the
    // dialog in-line with the list so we don't need this activity.
    finish();
    return;
   }

   if (savedInstanceState == null) {
    // During initial setup, plug in the details fragment.
    DetailsFragment details = new DetailsFragment();
    details.setArguments(getIntent().getExtras());
    getFragmentManager().beginTransaction()
      .add(android.R.id.content, details).commit();
   }
  }
}

/** Set layout for menu */
@Override
public boolean onCreateOptionsMenu(Menu menu) {
 super.onCreateOptionsMenu(menu);
 // to create an option menu
 MenuInflater inflater = getMenuInflater();
 inflater.inflate(R.layout.optionmenu, menu);
 return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
 switch (item.getItemId()) {
 case R.id.OrderItem: {
  // sort offers
  MainActivity.ShopInfo.OrderOffer();
  // set a new adapter for sorted shops
  FragmentLayout.TitlesFragment.getListView().setAdapter(
    new ArrayAdapter<String>(this.getBaseContext(),
      android.R.layout.simple_list_item_activated_1,
      MainActivity.ShopInfo.getAllShops()));
  // force a refresh of right fragment
```

```
   FragmentLayout.TitlesFragment.getListView().refreshDrawableState();
   // select the first item (force also the update)
   FragmentLayout.TitlesFragment.showDetails(0, true);
   // yes we processed the Option
   return true;
  }

  default: {
   return false;
  }
  }
 }

}
```

A.3 Offer.java

```
package dei.unipd.fragments;

import android.graphics.drawable.Drawable;

public class Offer implements Comparable<Object> {

 private String mShopName;
 private float mPrice;
 private Drawable mImage;
 private String mUrl;

 Offer(String name, float price, Drawable image, String url) {
  mShopName = name;
  mPrice = price;
  mImage = image;
  mUrl = url;
 }

 public String getShopName() {
  return mShopName;
 }

 public float getPrice() {
```

```
  return mPrice;
 }

 public Drawable getImage() {
  return mImage;
 }

 public String getUrl() {
  return mUrl;
 }

 // we need to implement compareTo to order a list of offer
 @Override
 public int compareTo(Object arg0) {
  Offer off = (Offer) arg0;
  if (this.mPrice < off.mPrice)
   return -1;
  else if (this.mPrice > off.mPrice)
   return 1;

  return 0;
 }

}
```

A.4 RetrievedInfo.java

```
package dei.unipd.fragments;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import android.graphics.drawable.Drawable;

public class RetrievedInfo {

 // info is a list of offer
 private List<Offer> mOfferList;
```

```
 RetrievedInfo() {
  mOfferList = new ArrayList<Offer>();
 }

 public void AddOffer(String shopName, float price, Drawable image,
   String url) {
  Offer off = new Offer(shopName, price, image, url);
  mOfferList.add(off);
 }

 public String[] getAllShops() {
  String[] sShops = new String[mOfferList.size()];
  for (int i = 0; i < sShops.length; i++)
   sShops[i] = mOfferList.get(i).getShopName();

  return sShops;
 }

 public Drawable getImage(int index) {
  return mOfferList.get(index).getImage();
 }

 public float getPrice(int index) {
  return mOfferList.get(index).getPrice();
 }

 public String getUrl(int index) {
  return mOfferList.get(index).getUrl();
 }

 public void OrderOffer() {
  // since offer implements compareTo
  Collections.sort(mOfferList);
 }

}
```

A.5 Util.java

```
package dei.unipd.fragments;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.apache.http.HttpResponse;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;
import android.graphics.drawable.Drawable;

public class Util {

 //strings to search inside HTML
 private final static String SHOP_NAME = "shopname";
 private final static String PRICE_OFFERS = "priceOffers";
 private final static String PROD_PHOTO = "prodPhoto";
 private final static String PROD_INFO = "prodInfo";

 /***
  * Check if a string contains some useful information
  *
  * @return the number of line after @s which are interesting
  * */
 private static int CheckString(String s) {
  if (s.contains(SHOP_NAME))
   return 1;
  else if (s.contains(PRICE_OFFERS))
   return 2;
  else if (s.contains(PROD_PHOTO))
```

```
   return 3;
  else if (s.contains(PROD_INFO))
   return 3;

  return 0;
 }

 /***
  * Execute a generic HttpGet
  *
  * @url reference url
  * @return an InputStream useful to extract the contents
  * (null in case of error)
  * */
 private static InputStream executeHttpGet(String url) {
  InputStream in = null;
  HttpClient client = new DefaultHttpClient();
  HttpGet request = new HttpGet();
  try {
   request.setURI(new URI(url));
  } catch (URISyntaxException e) {
   return null;
  }

  HttpResponse response = null;
  try {
   response = client.execute(request);
  } catch (ClientProtocolException e) {
   return null;
  } catch (IOException e) {
   return null;
  }

  try {
   in = (response.getEntity().getContent());
  } catch (IllegalStateException e) {
   return null;
  } catch (IOException e) {
   return null;
  }
```

```
 return in;
}

/**
 * Execute an HttpGet for an image
 *
 * @param reference
 *            url
 * @return image as a Drawable
 */
public static Drawable executeImageHttpGet(String url) {

 InputStream in = executeHttpGet(url);
 if (in == null)
  return null;
 Drawable d = Drawable.createFromStream(in, "src");
 try {
  in.close();
 } catch (IOException e) {

 }
 return d;
}

/**
 * Executes an HtmlHttpGet, and extracts the text. For
 * performance and memory reasons, only useful lines are returned
 *
 * @param reference url
 * @return an array of String extracted form HTML
 * **/
public static String[] executeHtmlHttpGet(String url) {

 String[] sRes = null;
 InputStream is = executeHttpGet(url);
 if (is == null)
  return null;

 //
 BufferedReader in = new BufferedReader(new InputStreamReader(is));
```

```
 // use a list to store the lines
 List<String> ls = new ArrayList<String>();
 String line = "";
 try {
  int LineAcc = 0;
  while ((line = in.readLine()) != null) {
   // check if a line contains some useful information
   LineAcc += CheckString(line);
   if (LineAcc > 0) {
    ls.add(line);
    LineAcc--;
   }
  }
  in.close();
 } catch (IOException e) {
  return null;
 }

 sRes = new String[ls.size()];
 ls.toArray(sRes);

 return sRes;
}

/**
 * Parse the HTML, and store the informations inside info
 *
 * @param sHttp
 *            : is the string to parse
 * @param info
 *            : where to store infos
 * */
public static void parse(String[] sHttp, RetrievedInfo info) {
 String shopName = "";
 float price = 0;
 Drawable image = null;
 String url = null;

 // scan the array
```

```
  for (int i = 0; i < sHttp.length; i++) {
   String s = sHttp[i];

   // retrieve shop name
   int pos = s.indexOf(SHOP_NAME);
   if (pos > 0) {
    shopName = getHtmlInnerValue(pos, s);
    // beware: this is the last entry in the HTML
    // so we have to call here AddOffer()
    info.AddOffer(shopName, price, image, url);
   }

   // retrieve price
   if (s.contains(PRICE_OFFERS)) {
    s = sHttp[i + 1];
    String sRawPrice = getHtmlInnerValue(0, s);
    String sPrice = getPrice(sRawPrice);
    sPrice = sPrice.replace(',', '.');
    price = Float.parseFloat(sPrice);
   }

   // retrieve photo
   if (s.contains(PROD_PHOTO)) {
    s = sHttp[i + 2];
    s = getDoubleQuoteText(s, 0);
    image = executeImageHttpGet(s);
   }

   // retrieve url from internal link to 'ciao'
   if (s.contains(PROD_INFO)) {
    s = sHttp[i + 2];
    s = getDoubleQuoteText(s, 0);
    int urlPos = s.indexOf("url=");
    s = s.substring(urlPos + 4);
    url = toURL(s);
   }
  }
 }

/**
```

```
 * Return inner text (within '<' and '>')
 * @param pos: the position to start inside the string
 * */
private static String getHtmlInnerValue(int pos, String s) {
 return getTextDelim(s, pos, '>', '<');
}

/**
 * Extracts the price from the string (using Regular Expression)
 * */
private static String getPrice(String s) {
 Pattern p = Pattern.compile("[0-9]+,[0-9]+");
 Matcher m = p.matcher(s);
 if (!m.find())
  return null;
 return m.group();
}

/**
 * Return inner text (within '"' and '"')
 *
 * @param pos
 *            : the position to start inside the string
 * */

private static String getDoubleQuoteText(String s, int pos) {
 return getTextDelim(s, pos, '"', '"');
}

/**
 * get the text embedded between startDelim and endDelim,
 * starting from position pos of the string s
 * */
private static String getTextDelim(String s, int pos,
char startDelim, char endDelim) {
 s = s.substring(pos);
 int startPos = s.indexOf(startDelim);
 if (startPos < 0)
  return null;
```

```
  int endPos = s.indexOf(endDelim, startPos + 1);
  if (endPos < 0)
   return null;
  return s.substring(startPos + 1, endPos);
 }

 /**
  * remove URL invalid chars from String s
  * */
 private static String toURL(String s) {
  s = s.replace("%2F", "/");
  s = s.replace("%3A", ":");
  s = s.replace("%26", "&");
  s = s.replace("%3D", "=");
  s = s.replace("%3F", "?");
  return s;
 }

}
```

Appendice B

Elenco delle Figure

Appendice C

Elenco dei Codici