Università Politecnica delle Marche

Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Progettazione e implementazione di un'app Android per supportare l'organizzazione delle attività da parte di uno studente

Design and implementation of an Android app for supporting the organization of the activities of a student

Relatore	Candidato
Prof. Domenico Ursino	Kevin Cela

Anno Accademico 2018-2019

Indice

Elenco delle figure

Elenco dei listati

Introduzione

Grazie agli importanti progressi tecnologici avvenuti nell'ultimo decennio, sempre più persone hanno ora accesso ad uno smartphone. Lo *smartphone*, anche conosciuto come "telefono intelligente", ha radicalmente cambiato il modo con cui una persona si interfaccia alla realtà, essendo questo in grado di effettuare delle operazioni, normalmente complesse, che aiutano l'utente nelle sue attività giornaliere. Grazie a questi dispositivi è possibile, infatti, oltre alla chiamata e alla messaggistica, effettuare operazioni che in precedenza erano possibili soltanto mediante l'uso di un PC, o altri dispositivi appositi, come, ad esempio, ascoltare dei file audio, scrivere delle e-mail, acquisire delle immagini o guardare dei video.

Questa diffusione degli smartphone, avvenuta in seguito all'avvento di dispositivi quali l'iPhone, che ha rivoluzionato il form factor di questi dispositivi, rendendoli più accessibili al pubblico, ha portato alla creazione di un enorme ecosistema, permettendo a chiunque di sviluppare, a fine commerciale o personale, delle applicazioni per tali dispositivi. È proprio grazie a queste app che lo smartphone continua ad acquisire funzionalità sempre più interessanti e complesse, che vanno a vantaggio dell'utente. La crescente popolarità di questi dispositivi è accompagnata da una crescita nelle opportunità lavorative nell'ambito dello sviluppo di applicazioni mobile, evidenziando un mercato che sta diventando sempre più competitivo.

Essendo l'avvento degli smartphone un evento relativamente recente, la maggior parte dei loro utilizzatori è giovane, nonostante sempre più persone stiano ormai adottando questi dispositivi. Lo smartphone è ormai entrato pienamente nella routine di questa categoria di persone; molti di loro controllano il proprio cellulare appena svegliati, oppure lo utilizzano ogniqualvolta ci sia un momento di pausa, essendo gli smartphone particolarmente agevoli da utilizzare in queste occasioni. Ed è proprio per questo che, ultimamente, sempre più giovani tendono ad organizzare le proprie attività mediante il proprio smartphone, utilizzando delle applicazioni specifiche. In questo scenario, una vera e propria rivoluzione si è avuta nella gestione del calendario personale; grazie alla natura intrinseca dello smartphone e del suo ecosistema è, infatti, possibile consultare in ogni momento i propri impegni, essendo tutte le informazioni raccolte in un unico posto e sempre accessibili in ogni istante.

Un'altra categoria di applicazioni che ha recentemente visto un enorme sviluppo nel mondo degli smartphone è costituita dalle app usate a scopo educativo, com'è possibile notare dalle molte applicazioni utilizzate per apprendere altre lingue, che

permettono a chiunque di imparare nuovi termini tramite la loro ripetizione. Queste applicazioni sono sempre più popolari grazie alla loro facilità d'uso, assieme all'accessibilità e al ruolo centrale che ha ormai assunto lo smartphone.

Considerato il fatto che molti degli utilizzatori di smartphone sono giovani, e visto il loro utilizzo frequente di tali dispositivi, si potrebbe, allora, pensare di unire l'aspetto organizzativo, presente in applicazioni quali calendari o note, a quello educativo, presente, per esempio, nelle applicazioni sopra descritte per l'apprendimento di una nuova lingua, realizzando una nuova applicazione che aiuta uno studente nelle sue attività. In questo modo, qualsiasi persona che abbia intenzione di organizzarsi sullo studio di una determinata materia può farlo mediante tale app, che sarà progettata appositamente per permettere una gestione centralizzata delle attività di studio.

Ed ecco, quindi, che nasce l'idea di un'applicazione che permette, ad uno studente intenzionato ad apprendere uno o più argomenti sia di organizzare le proprie attività di studio che di apprendere delle nuove nozioni o ripassarne alcune, il tutto utilizzando un unico dispositivo e, in particolare, un'unica applicazione.

Questa app, la cui progettazione verrà discussa nella presente tesi, permetterà allo studente di organizzarsi innanzitutto grazie ad un *calendario*, che gli consentirà di registrare gli eventi relativi a una determinata materia, così da poter programmare in modo appropriato le sue attività di studio. Questo calendario sarà, anche, corredato da un *sistema di notifiche*, usato per ricordare allo studente tutti gli eventi a breve termine. Questo sistema potrà essere liberamente attivato o disattivato dall'utente stesso, nel caso in cui voglia usufruire o meno di tale funzionalità.

Un altro strumento, utilizzato dall'applicazione per permettere all'utente una migliore organizzazione delle proprie attività consiste in un sistema che permette la gestione delle *note* relative ad una materia. Grazie ad esso lo studente potrà appuntarsi dei dettagli di rilievo, relativamente ad un argomento, per poi poterli consultare liberamente all'interno dell'applicazione, ogniqualvolta lo desidera.

Per quanto riguarda l'attività di studio, invece, l'utente potrà apprendere in qualsiasi momento dei nuovi concetti, facendo uso di un sistema basato su flashcard. Una *flashcard* può essere vista come una carta che rappresenta un concetto, utilizzata per la memorizzazione dello stesso. Quando lo studente ha intenzione di apprendere dei nuovi concetti, o di ripassarne alcuni, potrà, infatti, utilizzare l'applicazione, che mostrerà uno ad uno i concetti rappresentati su ogni carta, ordinati in modo appropriato in base alla difficoltà degli stessi. In questo modo, visualizzando più volte tali nozioni e ripetendole, lo studente potrà ricordare meglio, mediante un esercizio di stimolazione attiva della memoria.

Per catalogare le flashcard l'applicazione farà, inoltre, uso di alcuni *mazzi*, che possono essere definiti dall'utente e vengono utilizzati per raggruppare tra di loro più carte che mostrano nozioni relative allo stesso argomento. Questi mazzi saranno, a loro volta, raggruppati in base alla materia a cui faranno riferimento, per permettere allo studente di identificare il mazzo di interesse su cui effettuare l'attività di studio più facilmente.

Lo studente, grazie all'applicazione, potrà, infine, definire una o più *materie*, utilizzate per catalogare le informazioni dell'applicazione per garantire una organizzazione più efficiente. Per facilitare il riconoscimento di una materia verrà, inoltre,

associato un colore a ciascuna di esse, in modo da essere facilmente individuabili all'interno dell'app.

Nella presente tesi, dopo una breve panoramica riguardante il sistema operativo Android e gli strumenti utilizzati per sviluppare su tale piattaforma, si introdurrà nello specifico l'applicazione in oggetto, illustrandone la sua progettazione e implementazione.

La tesi, nel dettaglio, sarà strutturata nei seguenti capitoli:

- Nel primo capitolo verrà proposta una breve descrizione sul sistema operativo Android, mostrando, oltre alle sue funzionalità, il suo funzionamento dal punto di vista di uno sviluppatore, presentando, tra l'altro, anche gli strumenti principalmente utilizzati per lo sviluppo di un'applicazione su tale piattaforma.
- Nel secondo capitolo si comincerà a parlare dell'applicazione a partire dall'analisi dei requisiti, in cui verranno meglio delineate le funzionalità che questa dovrà presentare, assieme ai requisiti che dovranno, in seguito, essere soddisfatti in fase di progettazione.
- Nel terzo capitolo verrà trattata la progettazione dell'applicazione, in cui si decideranno, oltre alla struttura che questa dovrà utilizzare, anche l'interfaccia utente che dovrà presentare, illustrando, inoltre, come alcune delle sue funzionalità più complesse dovranno essere implementate e come i dati di interesse dovranno essere rappresentati nel modello relazionale.
- Nel quarto capitolo verrà descritta l'implementazione dell'applicazione; in essa verranno descritte le sue funzionalità principali illustrando le componenti utilizzate per la loro implementazione, presentando, inoltre, dei listati per mostrare, nel codice dell'applicazione stessa, l'implementazione di tali funzionalità.
- Nel quinto capitolo verranno tratte le conclusioni, e si discuterà su come l'applicazione può essere ulteriormente migliorata nel caso di eventuali sviluppi futuri.

1

Uno sguardo ad Android

In questo capitolo verrà proposta una breve introduzione al sistema operativo Android, preso in considerazione per lo sviluppo dell'applicazione oggetto della presente tesi, descrivendone le caratteristiche principali e gli strumenti adottati per sviluppare su tale piattaforma.

1.1 Android come Sistema Operativo

1.1.1 Storia di Android

Android è un sistema operativo per dispositivi mobili che negli ultimi anni ha conosciuto una diffusione straordinaria, al punto da raggiungere l'88% della quota nel mercato dei sistemi operativi mobile nel secondo trimestre del 2018. Prima di arrivare a tal punto, tuttavia, c'è stata una serie di eventi che hanno portato questo sistema operativo al punto in cui si trova adesso. Seppur Android sia ora conosciuto come il sistema operativo di Google, infatti, all'inizio non era di sua proprietà.

Android vide la luce in seguito alla nascita della società Android Inc., fondata nel 2003 da Andy Rubin, Rich Miner, Nick Sears e Chris White. L'acquisizione dell'azienda da parte di Google avvenne nel 17 agosto 2005, dal momento che che la società di Mountain View desiderava entrare nel mercato della telefonia mobile; è in questo periodo che l'azienda comincia a sviluppare un sistema operativo per dispositivi mobili, basandosi sul kernel di Linux. Tale sistema operativo, che prese il nome di Android, venne quindi presentato nel 2007 alla neonata *Open Handset Alliance (OHA)*, un consorzio di aziende del settore Hi Tech che include Google, HTC e Samsung, assieme a diversi operatori di telefonia mobile, e cominciò ad essere distribuito al pubblico e agli sviluppatori come un software open source.

Dopo il rilascio della prima versione nel 2008 Android conobbe una rapida crescita, sia in popolarità che dal punto di vista tecnologico: nel corso di pochi anni vennero infatti rilasciati molti aggiornamenti, fino ad arrivare alla Versione 3.0 (Gingerbread) nel 2011, che introdusse il supporto del sistema operativo ai tablet, dispositivi che al tempo ebbero una grande crescita in popolarità. Nei successivi anni Android si diffuse anche su dispositivi non inerenti a smartphone, quali televisioni ed orologi, estendendo ulteriormente la sua popolarità. In questo momento

l'ultima versione rilasciata del sistema operativo è la Versione 9.0, che è attualmente presente in buona parte degli smartphone di fascia media/alta in vendita.

1.1.2 Struttura del sistema operativo Android

L'*architettura* del sistema operativo Android può essere schematizzata come mostrato nella Figura 1.1.

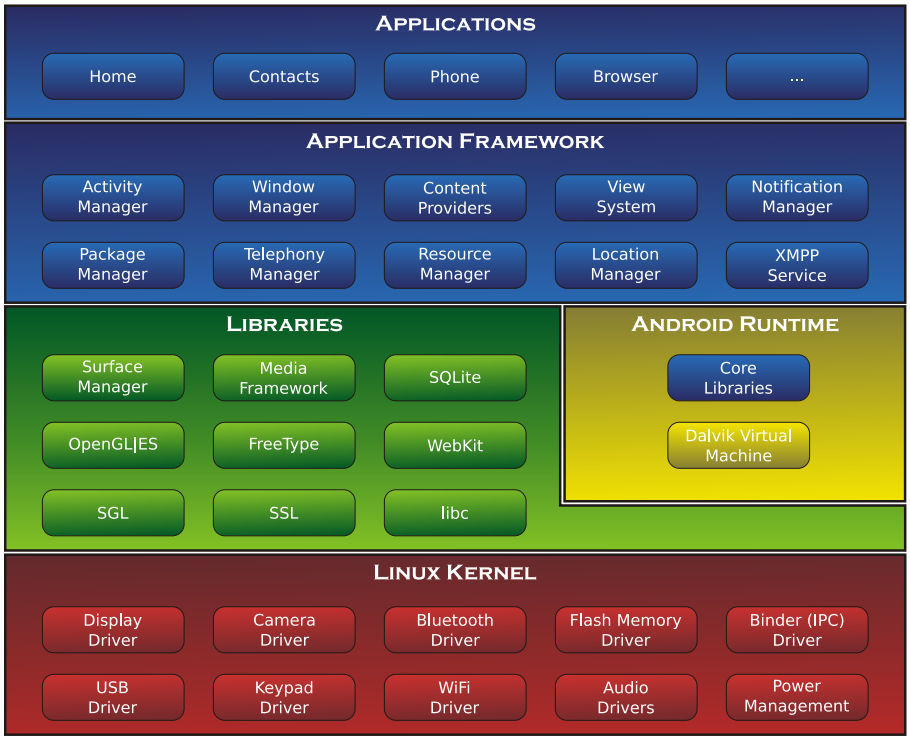

Figura 1.1. Architettura del sistema operativo Android

Al livello più basso Android fa uso di un *kernel Linux*, opportunamente modificato in base al dispositivo su cui è installato. Nonostante questo, Android ha poco a che fare con i sistemi operativi basati su Linux comunemente usati nell'ambito desktop, essendo progettato appositamente per dispositivi mobili, che hanno un'architettura diversa da quella desktop, così come dei requisiti differenti da soddisfare.

Sopra il kernel, che è la componente principale del sistema operativo, è presente un *middleware* formato da librerie e API scritte in C quali, ad esempio, SQLite e OpenGL, non inclusi nel kernel di Android, così come un *framework* utilizzato dalle applicazioni Android, contenente anche librerie compatibili con Java.

Le applicazioni che vengono eseguite su Android sono normalmente scritte in Java; tuttavia, al posto della Java Virtual Machine viene fatto uso del *Runtime di Android (ART)*. ART, inizialmente, utilizzava la macchina virtuale *Dalvik* per eseguire il bytecode ottenuto in seguito alla compilazione del codice Java. Tale codice,

infatti, dopo essere stato opportunamente compilato come un insieme di file `.class`, veniva ricompilato mediante il compilatore dex per ottenere il *"dex-code"*, un tipo di codice che può essere eseguito direttamente dalla macchina virtuale Dalvik. Dopo il rilascio di Android 5.0, tuttavia, è stata introdotta una nuova versione del Runtime di Android, il quale, al posto di utilizzare una macchina virtuale, compila direttamente il dex-code ottenuto da Java traducendolo in codice macchina utilizzando la *compilazione Ahead-Of-Time (AOT)*, garantendo delle ottimizzazioni sulle prestazioni dell'applicazione che non era possibile ottenere in precedenza.

Le componenti sopra descritte permettono allo sviluppatore una realizzazione più rapida ed efficiente di un'applicazione, ottenendo delle buone prestazioni anche senza lavorare a basso livello.

1.2 Android come Piattaforma di Sviluppo

1.2.1 Cos'è un'app nativa

Dopo aver visto la struttura di Android come sistema operativo è, ora, possibile descrivere il processo di sviluppo di un'app su questa piattaforma, introducendo gli strumenti utilizzati a tale scopo. Prima di fare ciò, tuttavia, è necessario fare una considerazione sui tipi di applicazione che possono essere sviluppati nell'ambito della programmazione mobile. Si possono, infatti, distinguere tre categorie di applicazioni (Figura 1.2):

- *App Native*: sono sviluppate in maniera specifica per un sistema operativo, facendo pieno uso delle sue funzionalità.
- *Web App*: sono delle semplici applicazioni web che mostrano un'interfaccia specifica per i dispositivi mobili. Non richiedono di essere installate; però possono usare soltanto le funzionalità del browser, e non quelle del sistema operativo. Una web app viene spesso utilizzata per far vedere all'utente dei contenuti di un servizio, senza implementare funzionalità specifiche del sistema operativo, quali GPS o notifiche push.
- *App Ibride*: sono una via di mezzo tra le applicazioni native e le web app, essendo delle applicazioni multipiattaforma che mettono a disposizione alcune delle funzionalità del sistema operativo, seppur con prestazioni inferiori alle app native.

La tipologia di applicazioni che verrà considerata nell'ambito della presente tesi è quella delle *app native*, che costituiscono la maggioranza delle applicazioni presenti su Android. Queste app, come già detto, sono progettate in maniera specifica per il sistema operativo, ed è per questo motivo che le loro prestazioni sono generalmente eccellenti, offrendo una *buona reattività e affidabilità* e assicurando all'utente un'*ottima esperienza d'uso*. Altri vantaggi delle app native sono i seguenti:

- *Accesso semplice a tutte le funzionalità del sistema operativo*, incluse quelle dello smartphone (fotocamera, GPS, accelerometro, etc.)
- *Possibilità di funzionare anche senza una connessione Internet*, al contrario delle applicazioni web che invece ne fanno uso.

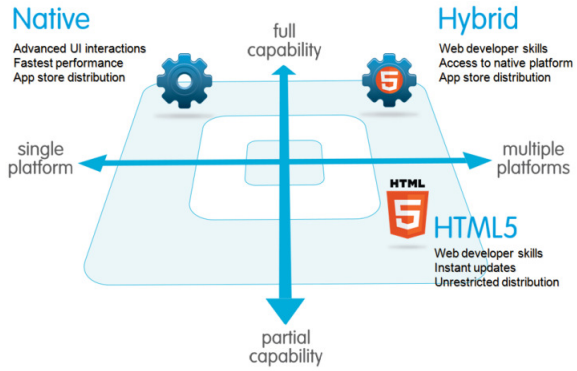

Figura 1.2. Confronto tra le tre categorie di applicazioni

- *Maggiore sicurezza*, non avendo le applicazioni native le molteplici falle che caratterizzano invece il mondo delle applicazioni web.

Nonostante questi vantaggi, tuttavia, sono anche presenti degli svantaggi, quali:

- *Maggiore costo per lo sviluppo dell'applicazione*, visto che ad ogni sistema operativo corrispondono un insieme di strumenti e tecnologie specifiche da utilizzare, rendendo necessario lo sviluppo dell'applicazione più volte, e in diversi linguaggi, a seconda delle piattaforme in cui si vuole lavorare.
- *Maggiore difficoltà nella manutenzione dell'applicazione*, in quanto, per ogni piattaforma, possono essere presenti problemi di diversa natura, per cui è necessario effettuare la manutenzione di più applicazioni invece che di una sola.

Per quanto riguarda Android nello specifico, le principali tecnologie utilizzate per lo sviluppo di app native sono il linguaggio di programmazione *Java* (oppure *Kotlin*, recentemente introdotto come alternativa), e il linguaggio di markup *XML*, usato nello specifico per la progettazione dell'interfaccia utente e del layout dell'applicazione.

1.2.2 Introduzione ad Android Studio

Il principale strumento usato per sviluppare applicazioni su Android è *Android Studio* (Figura 1.3), un *IDE* (ambiente di sviluppo) utilizzato per la realizzazione di app su Android, basato sull'IDE IntelliJ di Jetbrains.

Essendo Android Studio creato appositamente per facilitare lo sviluppo di applicazioni sulla piattaforma Android, viene offerta al programmatore la possibilità di creare un progetto con una struttura standardizzata e con tutti gli strumenti di cui ha bisogno per sviluppare e testare un'app. Alcuni esempi di strumenti che vengono offerti dall'IDE possono essere lo strumento di build automation *Gradle*, così come gli *emulatori* (Figura 1.4), che possono essere utilizzati per testare l'applicazione senza dover usare altri dispositivi. Ci sono, anche, altre funzionalità messe a disposizione dall'ambiente di sviluppo, quali, ad esempio:

Figura 1.3. Logo di Android Studio

- Un editor visuale per la progettazione dell'interfaccia utente senza l'utilizzo di XML.
- La possibilità di distribuire l'applicazione creata in formato APK, facilmente installabile su un dispositivo Android.
- Strumenti per monitorare le prestazioni dell'applicazione, per rilevare eventuali problemi di compatibilità tra le varie API di Android o per descrivere altri tipi di problemi non inerenti alla logica dell'app.
- Supporto a diversi servizi offerti da Google e, più in generale, a una vasta gamma di dispositivi, usati nella fase di debug dell'applicazione.

Figura 1.4. Emulatore usato per testare le applicazioni offerto dall'SDK di Android

A tutte queste funzionalità si aggiungono, anche, quelle che caratterizzano un IDE generale, come, ad esempio, la possibilità di vedere in tempo reale gli errori di

compilazione, la possibilità di utilizzare dei break point per analizzare il comportamento dell'applicazione in punti precisi (*debugging*), oppure la presenza di short-cut, per introdurre parti di codice senza doverli scrivere a mano, ottimizzando il workflow del programmatore.

1.2.3 Struttura di un'applicazione Android

In seguito all'apertura di Android Studio e alla creazione di un progetto al suo interno è già possibile notare come quest'ultimo abbia una *struttura ben consolidata*, con diverse cartelle associate a componenti differenti dell'applicazione (Figura 1.5).

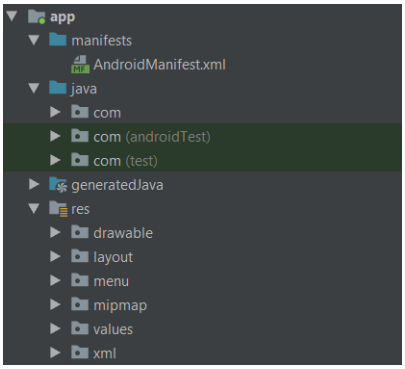

Figura 1.5. Struttura di un progetto Android

In particolare:

- La cartella `manifests` del progetto contiene l'*Android Manifest*, un file XML utilizzato per comunicare al sistema operativo, agli strumenti di compilazione di Android e a Google Play delle informazioni riguardanti l'applicazione. Alcuni dei dati riportati all'interno del Manifest possono, per esempio, essere:
  1. Il *nome del package* dell'applicazione, usato per identificare dove il codice è situato durante la compilazione del progetto.
  2. Le *componenti* dell'applicazione, che includono activity, service, broadcast receiver e content provider. Per ogni componente vengono indicate delle informazioni specifiche ad esso, come, ad esempio, il nome della sua classe Java ed, eventualmente, indicazioni su come esso può essere avviato.
  3. I *permessi* che l'applicazione deve richiedere per fare uso di funzionalità specifiche del dispositivo.
  4. Le *funzionalità hardware e software* richieste per un corretto funzionamento del programma.
- La cartella `java` contiene il codice sorgente dell'applicazione, il quale può essere opportunamente organizzato in package per avere un progetto ancora più strutturato. All'interno di questa cartella vengono, inoltre, già create le classi da utilizzare per effettuare operazioni di unit testing sull'applicazione.

- La cartella `res` contiene tutte le risorse dell'applicazione. Le *risorse* sono dei file aggiuntivi di cui l'applicazione fa uso, come, ad esempio, delle immagini, dei file che definiscono il layout dell'applicazione, o stringhe, contenute in una cartella a parte per separare la logica dell'applicazione dalla sua interfaccia. Nella Figura 1.5 è possibile notare un'ulteriore suddivisione delle risorse:
  1. La cartella `drawable` contiene delle immagini o dei file XML che rappresentano forme, immagini, animazioni ed altri elementi grafici. Gli elementi al suo interno vengono utilizzati per includere elementi grafici all'interno dell'applicazione.
  2. La cartella `layout` contiene i file XML che descrivono l'interfaccia utente di una componente dell'applicazione. Tali interfacce verranno, successivamente, iniettate all'interno dell'applicazione mediante l'uso di un inflater, che associa ad una componente dell'applicazione l'interfaccia definita nel file XML indicato.
  3. La cartella `menu` contiene dei file XML che definiscono le opzioni di un menù. Anche questi file verranno, poi, opportunamente iniettati all'interno di un menù attraverso l'uso di un inflater.
  4. La cartella `mipmap` contiene le icone dell'applicazione visibili dal launcher utilizzato dall'utente. Sono normalmente presenti diverse icone che vengono utilizzate per risoluzioni differenti del dispositivo.
  5. La cartella `values` contiene dei file XML che descrivono dei semplici valori, come dei colori, delle stringhe o dei numeri interi, che possono essere acceduti da qualsiasi parte dell'applicazione, evitando eventuali ripetizioni nel codice.
  6. La cartella `xml` contiene dei file XML di qualsiasi natura, che possono essere letti dall'applicazione.

  Oltre alle cartelle sopra menzionate, il programmatore può, anche, definire altre cartelle in base alle proprie esigenze tramite l'IDE, potendo personalizzare il proprio progetto a piacimento.

1.3 Componenti essenziali di un'applicazione Android

Dopo aver parlato della struttura di un'applicazione Android verranno, ora, introdotti alcuni degli elementi di base di cui essa è composta. Tali componenti rappresentano il fondamento di un'app Android, e la loro conoscenza è di importanza fondamentale per la sua realizzazione.

1.3.1 Activity

Tra tutte le componenti di un'app Android, l'Activity è probabilmente la più importante. Un'*Activity* in Android rappresenta a tutti gli effetti una cosa focalizzata che un'utente può fare. L'equivalente di un'Activity nell'ambito desktop è per esempio dato dalla finestra, anche questa rappresentante un'attività con cui l'utente può interagire. Ad ogni Activity viene associata un'interfaccia grafica mediante la funzione `setContentView()`, definita nella classe `Activity`, che verrà opportunamente ereditata dalle Activity che il programmatore intende implementare.

Ogni Activity presente in un'applicazione Android ha un suo *ciclo di vita*, definito da una serie di stati raggiungibili in momenti precisi dell'applicazione. Nella Figura 1.6 viene mostrato un diagramma che illustra nel dettaglio il ciclo di vita di un'Activity.

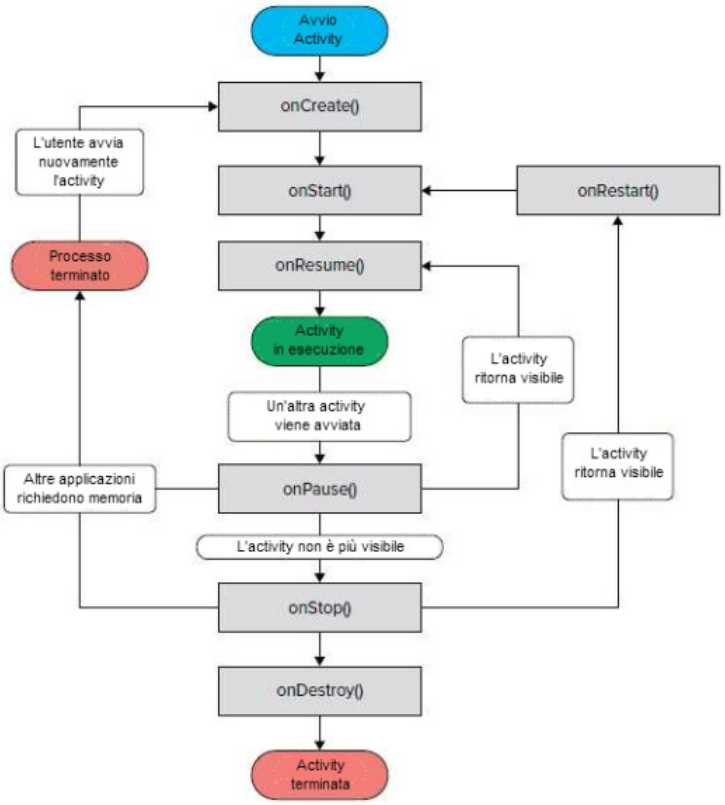

Figura 1.6. Il ciclo di vita di un'activity

I metodi riportati nel diagramma vengono eseguiti durante le varie fasi del ciclo di vita dell'Activity, e possono essere sovrascritti da classi che ereditano dalla classe `Activity` per effettuare delle operazioni in punti specifici del ciclo di vita del componente. Esaminiamo più da micino i metodi riportati nel diagramma:

- `onCreate()` viene chiamato quando un'Activity viene creata. Il metodo viene solitamente sovrascritto per effettuare operazioni di inizializzazione che dovrebbero avvenire soltanto una volta nel corso del ciclo di vita dell'Activity.
- `onStart()` viene chiamato quando l'Activity passa in primo piano (*foreground*). Ciò può avvenire dopo che un'Activity viene creata, (dopo `onCreate()`), o quando l'Activity ritorna ad essere visibile dopo essere stata messa in pausa (Activity non visibile), a seguito dell'attivazione del metodo `onResume()`.

- `onResume()` viene chiamato quando l'Activity si trova nello stato in cui può interagire con l'utente. L'app rimarrà in questo stato fino a quando non avviene un evento che rimuove il focus dall'applicazione (ricezione di una chiamata, utente che naviga in un'altra Activity, schermo bloccato, etc.).
- `onPause()` viene chiamato quando l'Activity è invisibile, o parzialmente visibile (ad esempio, quando un dialog compare sull'Activity). Viene solitamente sovrascritto per rilasciare delle risorse che non vengono utilizzate quando l'applicazione va in pausa.
- `onStop()` viene chiamato quando l'Activity è totalmente invisibile (*background*), oppure quando sta per terminare. Viene, solitamente, sovrascritto quando c'è la necessità di sospendere delle operazioni in corso nel momento in cui l'Activity va in background, o per rilasciare risorse di sistema.
- `onRestart()` viene chiamato quando l'Activity comincia a ritornare ad essere visibile per poi permettere delle nuove interazioni con l'utente, prima di richiamare il metodo `onStart()`.
- `onDestroy()` viene chiamato prima che l'Activity venga distrutta. Ciò avviene, ad esempio, quando l'Activity ha terminato il suo lavoro o quando, in seguito a un cambiamento nella configurazione del sistema (come un cambiamento di orientamento), l'Activity deve essere temporaneamente distrutta per poi essere ricreata. Viene, solitamente, sovrascritto per liberare tutte le risorse che non sono già state liberate durante gli stati precedenti.

Comprendere il ciclo di vita di un'Activity è di vitale importanza per lo sviluppo di un'app Android, in quanto molti dei problemi che possono essere riscontrati durate la sua realizzazione derivano proprio da tale meccanismo. Capita spesso, infatti, di dover includere delle operazioni aggiuntive durante la distruzione e la creazione dell'Activity per far fronte a problemi come il cambio dell'orientamento, in cui l'Activity deve essere ricostruita prima di poter essere utilizzata.

Tra le Activity di un'applicazione è presente un sistema di navigazione, implementato mediante una struttura chiamata back stack. Il *back stack* (Figura 1.7) è una pila in cui vengono inserite le Activity dell'app che sono attive o in background; in cima è presente l'Activity più recente, che è in esecuzione. Quando l'Activity in esecuzione viene distrutta (ad esempio, dopo aver premuto il tasto "back"), viene anche rimossa dal back stack, e viene ripresa l'Activity che stava precedentemente in cima alla coda.

1.3.2 Intent

Per poter mettere in comunicazione due Activity tra di loro, Android fa uso di un oggetto che prende il nome di intent. Un *intent* rappresenta, a tutti gli effetti, una descrizione astratta di un'operazione che deve essere eseguita. Il vantaggio principale nell'uso degli intent sta nel fatto che il collegamento tra Activity, e quindi la scelta dell'Activity da invocare, non deve essere necessariamente stretto, ma può essere lasco. Così facendo, si ha che che l'utente che deve eseguire un'azione (come, ad esempio, aprire una pagina web) può utilizzare il software che ritiene più adatto allo scopo (ad esempio, il browser scelto come predefinito). Gli intent, proprio a causa di questo, possono essere *impliciti*, quando non viene indicata esplicitamente

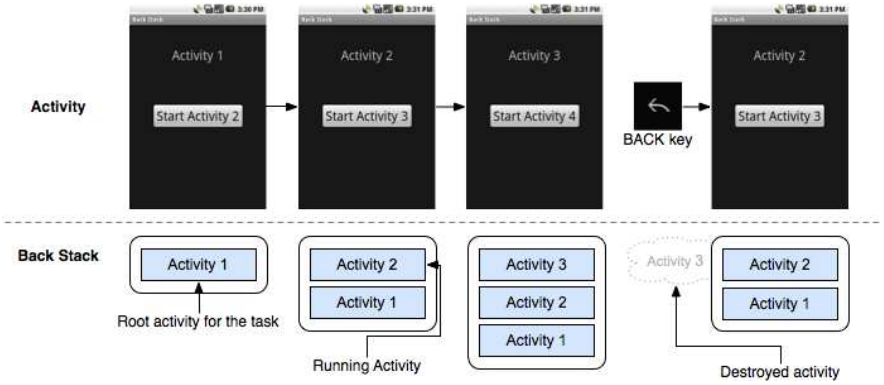

Figura 1.7. Esempio di back stack di un'applicazione

l'Activity da aprire, lasciando al sistema operativo la scelta dell'applicazione per l'azione, oppure *espliciti*, quando, invece, viene chiamata una componente specifica dell'applicazione. Gli intent permettono, anche, lo scambio di informazioni tra due componenti della stessa applicazione (Figura 1.8) mediante l'uso di *bundle*, ovvero degli oggetti in cui è possibile inserire dei valori di tipo primitivo (stringhe, interi, etc.) che possono essere inviati attraverso l'intent all'Activity che è stata chiamata; grazie a ciò, tale Activity può fare uso dei valori ricevuti.

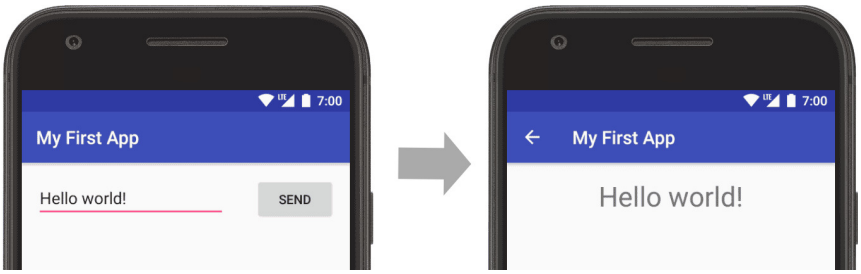

Figura 1.8. Un esempio di un intent, usato per il passaggio di dati da un'Activity a un'altra

1.3.3 View

Una *view* rappresenta l'elemento basilare che può essere utilizzato durante lo sviluppo di un'applicazione per la realizzazione di un'interfaccia grafica. Una view, in particolare, occupa uno spazio rettangolare all'interno della schermata ed è rappresentata dalla classe `View`, che contiene i metodi usati sia per eseguire il render dell'elemento che per gestire gli eventi associati a quest'ultimo (come ad esempio un click). Dalla classe `View` discendono sia i *widget*, ovvero le componenti usate per creare effettivamente le interfacce utente interattive, che i *ViewGroup*, utilizzati per l'implementazione dei layout. È possibile, nel caso in cui fosse necessario,

anche implementare degli elementi personalizzati, creando delle classi che derivano da `View`.

Alcuni dei widget previsti da Android che vengono comunemente utilizzati nella creazione di interfacce grafiche sono i seguenti:

- *Button:* rappresenta un pulsante che, una volta cliccato, può far partire un'azione specificata dal programmatore.
- *TextView:* utilizzato per mostrare sullo schermo un testo che può essere visualizzato dall'utente.
- *EditText:* elemento utilizzato per inserire e modificare testo. L'input potrà, poi, essere prelevato dall'app ed opportunamente elaborato.
- *ListView:* view usata per rappresentare un insieme di elementi come una lista. Richiede l'uso di un *adapter*, elemento che fa da ponte tra la sorgente di dati, che può essere di vario tipo, e la view, permettendo la visualizzazione degli elementi sullo schermo.
- *ImageView:* permette di mostrare delle immagini sullo schermo.
- *RadioButton:* pulsante che può essere selezionato o meno, esso viene, solitamente, usato in gruppo (usando i *RadioGroup*) per permettere la scelta di una sola opzione tra diverse alternative.
- *DatePicker:* widget che permette la selezione di una data, utilizzando un menù offerto dal sistema operativo.

Per progettare un'interfaccia grafica vengono utilizzati appositi file XML che contengono dei tag rappresentanti i widget, permettendo al programmatore di separare la parte logica dell'applicazione da quella grafica. Android Studio mette, anche, a disposizione un editor grafico per inserire le view senza dover necessariamente modificare il codice XML.

1.3.4 Layout

Prima di inserire dei widget all'interno della schermata è necessario definire una struttura ben precisa per la pagina, per garantire una disposizione degli elementi appropriata arricchendo l'esperienza d'uso. A tal fine è necessario utilizzare i *layout*, ovvero degli elementi che definiscono una struttura per l'interfaccia utente dell'applicazione. Normalmente in un'interfaccia grafica è presente una struttura gerarchica che ha come radice una *ViewGroup*, ovvero un contenitore per le view che implementa un layout (Figura 1.9).

Android fornisce diversi tipi di layout predefiniti. Quelli al momento più diffusi sono i seguenti:

- *LinearLayout* (Figura 1.10): gli elementi vengono posizionati sequenzialmente, dall'alto al basso, oppure da sinistra a destra, nel caso in cui l'orientamento impostato sia quello orizzontale.
- *RelativeLayout* (Figura 1.11): permette di visualizzare gli elementi figli in posizioni relative. La posizione di ogni elemento può essere specificata relativamente agli elementi vicini o all'elemento genitore (specificando, ad esempio, un allineamento in basso, al centro o a sinistra).

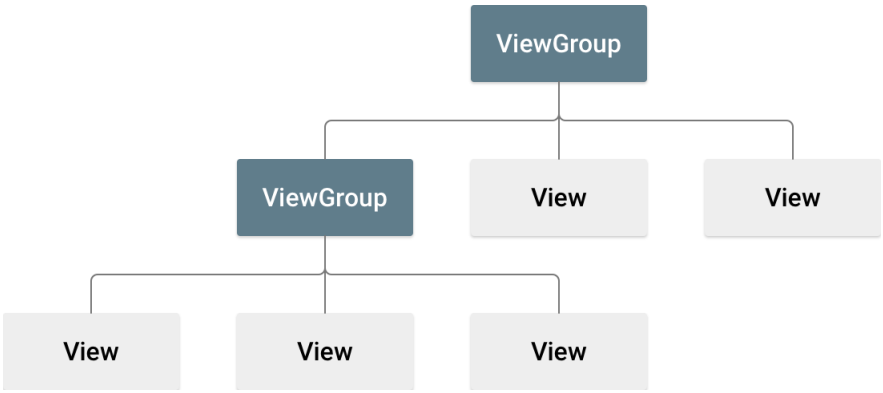

Figura 1.9. Struttura generalizzata di una interfaccia utente in Android

Figura 1.10. Un esempio di Linear Layout orizzontale

Figura 1.11. Un esempio di RelativeLayout

- *FrameLayout* (Figura 1.12): utilizzato per riservare un'area dello schermo a un elemento specifico, mostrandolo in quella zona. È possibile, inoltre, in-

serire diversi elementi figli in modo da mostrarli l'uno sopra l'altro, con un funzionamento simile a quello di una pila.

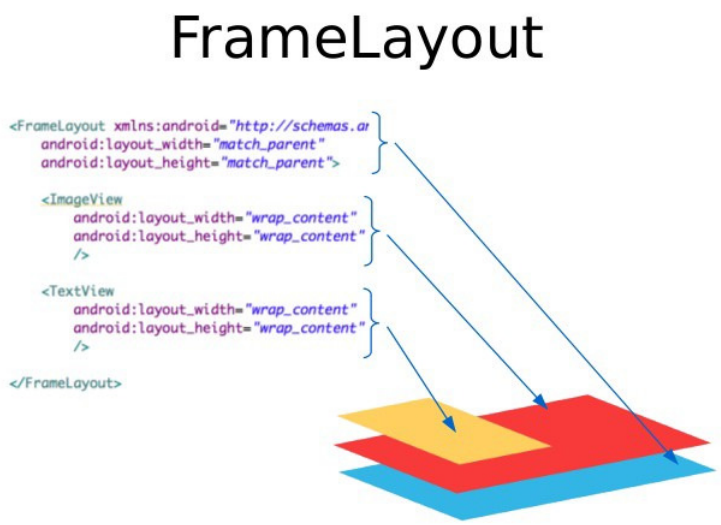

Figura 1.12. Un esempio di FrameLayout; i figli vengono mostrati su più livelli

- *ConstraintLayout* (Figura 1.13): permette il posizionamento degli elementi in base a dei vincoli (constraint), senza essere dipendente dalla risoluzione dello schermo e garantendo delle ottime prestazioni. L'uso dei constraint permette, inoltre, di realizzare interfacce più complesse che, normalmente, richiederebbero più tipi di layout innestati.

Questi sono soltanto alcuni dei molti layout messi a disposizione da Android; è anche possibile utilizzare più layout diversi innestati uno dentro l'altro per creare delle interfacce grafiche personalizzate e più complesse.

Figura 1.13. Un esempio di ConstraintLayout

2

Analisi dei requisiti

In questo capitolo verrà trattata l'analisi dei requisiti, di fondamentale importanza per la progettazione e l'implementazione di un'applicazione. Si partirà da una descrizione generale del progetto, per poi ottenere, da questa, i requisiti da soddisfare, realizzando, infine, un diagramma che descriverà i casi d'uso dell'applicazione. In questo modo sarà possibile individuare chiaramente i requisiti dell'applicazione, che dovranno essere successivamente soddisfatti durante la sua progettazione ed implementazione.

2.1 Descrizione del progetto

Il progetto, oggetto della presente tesi, riguarda lo sviluppo di un'applicazione Android. Tale applicazione ha lo scopo di offrire ad uno studente degli strumenti che lo assistano nelle sue attività di studio, permettendogli di organizzarsi al meglio e, allo stesso tempo, di apprendere più velocemente alcuni dei concetti su cui può riscontrare difficoltà. L'applicazione sarà, fondamentalmente, divisa in tre parti:

1. Una parte dedicata alla *gestione degli eventi del calendario*. In questo modo lo studente potrà avere un proprio calendario, in cui può tenere traccia di tutti gli eventi legati ad un corso (lezioni, esercitazioni, attività di studio, etc.), eventualmente con la possibilità di abilitare delle notifiche per ricordare all'utente l'evento prima dell'orario prestabilito.
2. Una parte dedicata all'*apprendimento tramite le flashcard*. Le *flashcard* rappresentano delle carte, raggruppate in mazzi divisi tematicamente, che contengono dei concetti che lo studente deve imparare. Durante la fase di studio delle flashcard lo studente vedrà prima una faccia della carta, contenente, per esempio, un titolo associato al concetto, dopodiché vedrà l'altra faccia, contenente il contenuto del concetto stesso. Se lo studente si è ricordato del concetto, la priorità della carta associata ad esso verrà diminuita, in caso contrario verrà aumentata. Così facendo si ha che, durante la fase di studio, verranno presentate prima le carte che hanno una priorità maggiore, così lo studente potrà esercitarsi prima sui concetti che trova più difficili.

   Questo strumento di apprendimento ha conosciuto una grande diffusione in aree quali, ad esempio, lo studio delle lingue straniere. Lo studio tramite

flashcard, infatti, favorisce l'apprendimento di concetti tramite la loro ripetizione, per cui il loro uso risulta essere spesso adatto per imparare parole di un'altra lingua (Figura 2.1).

Per realizzare questa sezione sarà necessario, inoltre, aggiungere delle funzionalità volte a gestire sia i mazzi che le carte associate ad ogni mazzo, offrendo all'utente l'opzione di creare, modificare ed eliminare tali elementi.

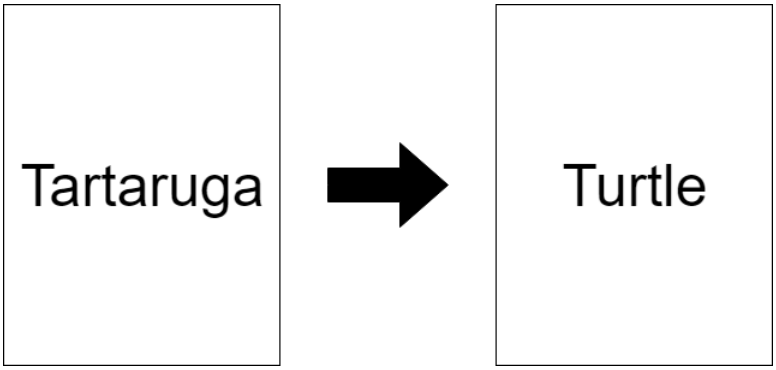

Figura 2.1. Esempio di una flashcard usata per apprendere una parola inglese

3. Una parte dedicata alla *gestione delle note*, offrendo l'opzione di creare, eliminare o modificare delle note per ogni materia. In questo modo l'utente potrà prendere velocemente nota di alcune informazioni di rilievo relative ad una materia, per poi poterle, in seguito, liberamente consultare all'interno dell'applicazione.

Per fare in modo che l'applicazione sia intuitiva da usare, inoltre, sarà necessaria l'implementazione di un menù per gestire le materie, in modo da poter catalogare in maniera appropriata ogni elemento appartenente alle tre sezioni precedentemente menzionate. Potrebbe essere importante anche l'implementazione di una pagina di aiuto, per guidare l'utente all'interno dell'applicazione, chiarendo eventuali dubbi, così come una pagina che gli permetta di gestire le impostazioni dell'app.

2.2 Requisiti funzionali e non funzionali

In questa sezione verrà proposta la *specifica dei requisiti* dell'applicazione, in modo da avere una descrizione completa del comportamento che questa dovrà assumere. I requisiti possono essere fondamentalmente divisi in due categorie: da un lato ci sono i *requisiti funzionali*, che descrivono le funzionalità che il sistema deve offrire all'utente, specificando, eventualmente, anche il suo comportamento rispetto a degli input particolari; dall'altro ci sono i *requisiti non funzionali*, che descrivono i vincoli sui servizi offerti dal sistema e sullo stesso processo di sviluppo. Generalmente i requisiti non funzionali non derivano da ciò che deve fare l'applicazione quanto, piuttosto, da fattori esterni ad essa.

2.2.1 Requisiti funzionali

L'applicazione da realizzare dovrà offrire le seguenti funzionalità all'utente:

1. *Creazione, lettura, modifica ed eliminazione (CRUD) di una materia*, utilizzata per catalogare gli eventi del calendario, le note e i mazzi.
2. *Creazione, lettura, modifica ed eliminazione (CRUD) di un evento del calendario*, specificando opportunamente la data, l'ora e la materia relative all'evento.
3. *Creazione, lettura, modifica ed eliminazione (CRUD) di un mazzo*, usato per contenere delle flashcard, specificandone il nome, una descrizione sintetica dei suoi contenuti e la materia a cui appartiene.
4. *Creazione, lettura, modifica ed eliminazione (CRUD) di una flashcard*, indicando sia il titolo della carta che il suo contenuto e specificando, inoltre, il mazzo a cui deve appartenere.
5. *Creazione, lettura, modifica ed eliminazione (CRUD) di una nota*, indicando sia il titolo della nota che la sua descrizione e specificando, inoltre, la materia a cui si riferisce.
6. *Attività di studio sulle flashcard appartenenti ad un mazzo*, con un sistema di priorità usato per presentare le carte in un determinato ordine.
7. *Implementazione di un sistema di notifiche*, usato per ricordare all'utente i prossimi eventi del calendario.
8. *Presentazione di una schermata di aiuto*, per guidare l'utente all'interno dell'applicazione.
9. *Presentazione di una schermata delle impostazioni*, che permette all'utente di configurare l'app (ad esempio, consentendogli di abilitare e disabilitare le notifiche sugli eventi del calendario).

2.2.2 Requisiti non funzionali

I requisiti non funzionali relativi all'applicazione sono i seguenti:

1. *L'applicazione dovrà essere sviluppata sulla piattaforma Android*, utilizzando le funzionalità offerte nativamente dal sistema operativo.
2. *L'applicazione dovrà fare uso di un database locale*, utilizzato per memorizzare tutte le informazioni rilevanti e per permettere l'implementazione di tutte le funzionalità CRUD.

2.3 Diagramma dei casi d'uso

Per concludere l'analisi dei requisiti, verrà ora introdotto il diagramma dei casi d'uso relativo all'applicazione. Il diagramma dei casi d'uso descriverà le funzionalità che vengono offerte dall'app, visualizzando in modo conciso le interazioni che si possono verificare tra un utente e le sue componenti. Gli elementi di base che sono utilizzati per costruire il diagramma sono:

- Gli *attori*, rappresentanti le entità esterne, che interagiscono con l'app tramite le funzionalità che essa offre. Gli attori possono essere di varia natura, come, per esempio, una persona che utilizza l'applicazione, oppure una componente dell'applicazione stessa.
- I *casi d'uso*, che rappresentano le funzionalità che il sistema offre, modellando i suoi requisiti funzionali.
- Le *associazioni*, che indicano una possibile interazione tra un attore e uno o più casi d'uso.
- Le *generalizzazioni*, usate per indicare delle relazioni di tipo gerarchico che possono esistere tra attori oppure tra casi d'uso, utili per considerare sia i casi generali che quelli specifici.
- L'*inclusione*, un collegamento tra due casi d'uso utilizzato quando un caso d'uso incorpora completamente il comportamento di un altro caso d'uso. Si ha, quindi, una dipendenza che è presente per ogni interazione.
- L'*estensione*, simile all'inclusione, ma utilizzato per dipendenze che si presentano contestualmente: in questo caso, infatti, il caso d'uso subordinato estende il comportamento del caso d'uso principale, aggiungendo funzionalità che possono essere richiamate contestualmente, rispettando determinate condizioni.

Nel caso dell'applicazione considerata, il diagramma dei casi d'uso è presentato nella Figura 2.2.

Dal diagramma si può notare, innanzitutto, come ci sia un unico attore; questo attore è il target dell'applicazione, ovvero lo *studente*, che fa uso delle funzionalità offerte da essa. Tra i casi d'uso, invece, rientrano tutte le funzionalità presenti nei requisiti funzionali dell'applicazione. Si può, inoltre, notare l'utilizzo dell'estensione in diversi punti del diagramma:

- I casi d'uso *Gestione Mazzi*, *Gestione Eventi* e *Gestione Note* estendono il caso d'uso *Gestione Materia*, in quanto, prima di poter aggiungere un mazzo, un evento o una nota, è necessario verificare che sia presente almeno una materia per catalogare l'elemento; in caso contrario, è necessario dapprima creare una materia, che verrà in seguito utilizzata per catalogare il nuovo elemento.
- Il caso d'uso *Gestione Eventi* estende il caso d'uso *Notifica Evento* in quanto, se l'utente acconsente, verrà utilizzato un sistema per gestire le notifiche di ogni nuovo elemento.
- Il caso d'uso *Gestione Carte* estende il caso d'uso *Gestione Mazzo* in quanto, prima di poter aggiungere una carta, potrebbe essere necessario aggiungere il mazzo a cui tale carta dovrà essere associata.

È possibile notare infine come sia presente un'associazione tra attore e caso d'uso per tutti i casi d'uso presenti nel diagramma fatta eccezione per il caso d'uso *Notifica Evento*, in quanto l'utente non interagisce direttamente con questo componente del sistema, ma è il sistema stesso che gestisce questa funzionalità, presentando il risultato all'utente.

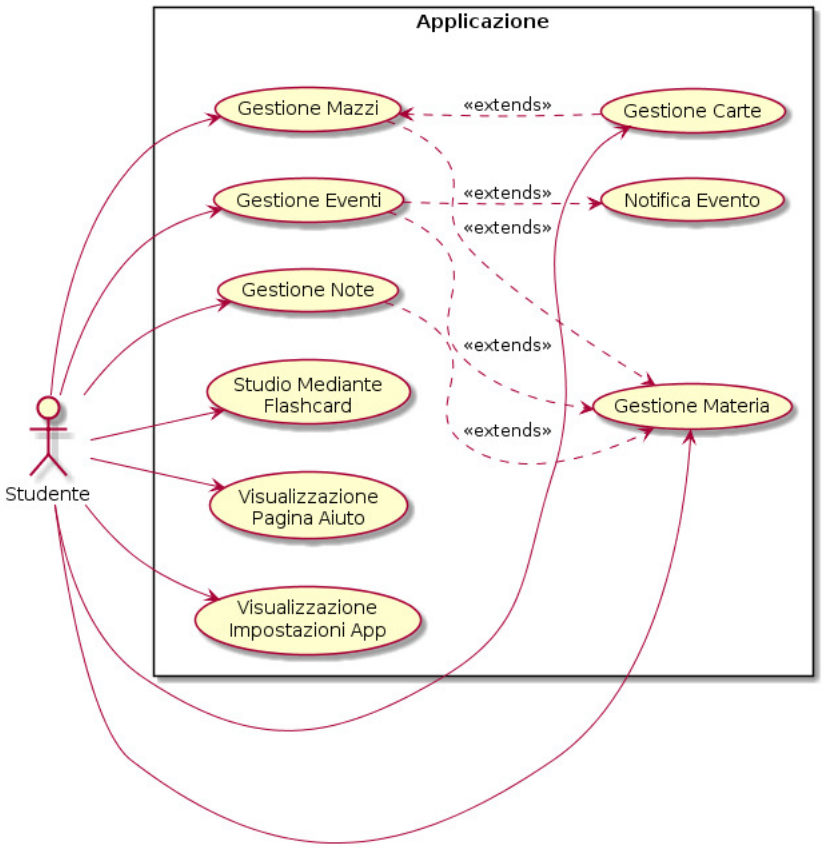

Figura 2.2. Diagramma dei casi d'uso dell'applicazione

3

Progettazione

Questo capitolo verrà dedicato alla progettazione dell'applicazione oggetto della presente tesi. In particolare verrà, innanzitutto, illustrata la struttura dell'applicazione; successivamente, verranno presentati dei wireframe che descriveranno sinteticamente la sua interfaccia grafica, così come degli activity diagram, che illustreranno alcuni dei processi che la caratterizzano. Si parlerà, infine, della progettazione della componente dati dell'app, che mira ad ottenere un modello di tipo relazionale per poter gestire i dati di interesse dell'applicazione.

3.1 Struttura dell'applicazione

Nella Figura 3.1 viene riportata la *mappa del sito* relativa all'applicazione, illustra, in modo intuitivo, la struttura che con cui si intendono suddividere le informazioni e le funzionalità in essa contenute.

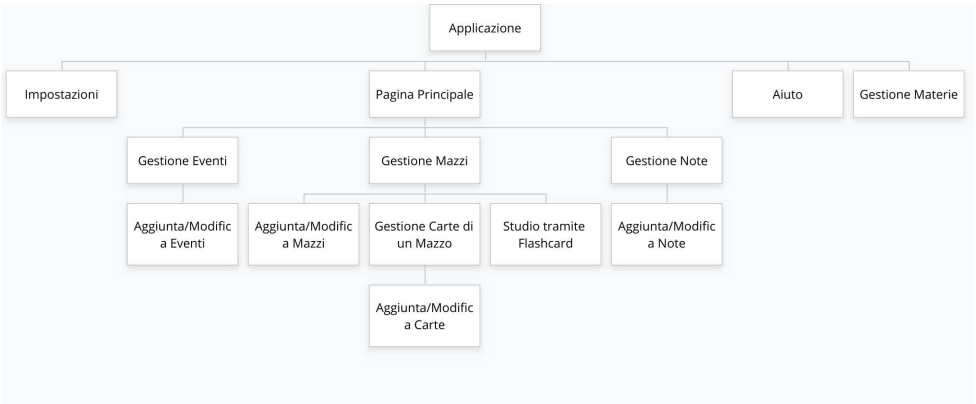

Figura 3.1. Mappa del sito dell'applicazione

Come è possibile notare dalla mappa, l'applicazione, una volta avviata, presenterà una *pagina principale*. Questa sarà composta da diverse sezioni (*Gestione*

Eventi, Gestione Mazzi, Gestione Note), a cui è possibile accedere mediante un sistema di navigazione, situato al suo interno. Da ciascuna di queste sezioni si potrà, poi, accedere a delle pagine che realizzano le loro funzionalità specifiche. In particolare:

- Dalla sezione dedicata alla gestione degli eventi è possibile accedere alla pagina per *aggiungere o modificare un evento*.
- Dalla sezione dedicata alla gestione delle note è possibile accedere alla pagina per *aggiungere o modificare una nota.*
- Dalla sezione dedicata alla gestione dei mazzi è possibile accedere alla pagina per *aggiungere o modificare un mazzo*, alla pagina che permette la *gestione delle carte* e alla pagina che viene utilizzata per effettuare l'attività di *studio mediante flashcard*. La pagina dedicata alla gestione delle carte, a sua volta, permetterà l'accesso ad una pagina per *aggiungere o modificare una carta* relativa al mazzo.

Oltre alla pagina principale sono presenti, allo stesso livello, altre pagine accessibili dalla toolbar dell'applicazione. Tali pagine sono, rispettivamente:

- La pagina delle *impostazioni*, che consente all'utente la configurazione dell'applicazione.
- La pagina di *aiuto*, che fornisce all'utente delle indicazioni circa il funzionamento dell'applicazione.
- La pagina di *gestione delle materie*, che fornisce all'utente le funzionalità per aggiungere, modificare o rimuovere una materia. Da questa pagina si può accedere alla schermata che permette l'aggiunta e la modifica di una materia.

3.2 Mockup dell'interfaccia utente

Una volta definita la struttura dell'applicazione, sarà necessario stabilire come le informazioni provenienti da essa devono essere presentate all'utente. Si dovrà, a tal fine, delineare l'*interfaccia utente* (UI) dell'app. Per definire un'interfaccia che sia intuitiva e gradevole, solitamente, nella fase di progettazione dell'applicazione, vengono realizzati dei mockup relativi ad essa. I *mockup* sono delle rappresentazioni statiche del progetto finale, che permettono di delinearne l'aspetto grafico prima della sua realizzazione. Grazie all'uso dei mockup è possibile, in fase di progettazione, evidenziare in modo efficace le caratteristiche che l'interfaccia dovrà presentare, e l'aspetto che questa dovrà avere. Generalmente la realizzazione dei mockup è divisa in tre fasi:

- Creazione dei *wireframe*, ovvero delle bozze che rappresentano, a basso livello, la struttura dell'interfaccia grafica, mettendo in risalto le caratteristiche essenziali che essa deve presentare.
- Creazione di un *prototipo*, in cui vengono implementate le funzionalità illustrate nel wireframe, presentando una prima interfaccia dinamica che può essere testata.
- Creazione dei *mockup* definitivi, che partono dai wireframe precedentemente definiti e li espandono, con il più alto livello di dettaglio e fedeltà possibile. In

questa fase si delineerà quello che sarà l'aspetto definitivo dell'applicazione, che verrà mostrato all'utente in seguito al suo avvio.

Per quanto riguarda l'applicazione presa in considerazione, per la progettazione sono stati realizzati dei wireframe che hanno lo scopo illustrare la struttura dell'interfaccia utente, che verrà in seguito realizzata. Il software utilizzato per la loro creazione è *Balsamiq Mockups*, uno degli strumenti attualmente più diffusi per la creazione di wireframe e mockup.

I wireframe che sono stati realizzati per l'applicazione sono i seguenti:

- *Gestione degli eventi (senza calendario)* (Figura 3.2);
- *Gestione degli eventi (con calendario)* (Figura 3.3);
- *Gestione dei mazzi* (Figura 3.4);
- *Gestione delle note* (Figura 3.5);
- *Gestione delle carte* (Figura 3.6);
- *Studio tramite flashcard* (Figura 3.7);
- *Impostazioni dell'app* (Figura 3.8);
- *Schermata di aiuto* (Figura 3.9);
- *Gestione delle materie* (Figura 3.10).

Dai wireframe è possibile, inoltre, intravedere alcune delle scelte, fatte in fase di progettazione, riguardanti l'interfaccia dell'applicazione.

La sezione relativa alla gestione degli eventi, ad esempio, presenta due tipi di layout: il primo elenca semplicemente gli eventi in una lista, raggruppandoli in base al giorno dell'anno (Figura 3.2), mentre il secondo mostra esplicitamente un *calendario* (Figura 3.3), usato per selezionare una data e, conseguentemente, mostrare tutti gli eventi programmati per quel giorno. L'utente potrà decidere quale dei due layout utilizzare tramite le impostazioni dell'applicazione.

Si è scelto, inoltre, di associare ad ogni materia un colore definito dall'utente (Figura 3.10), che verrà utilizzato in corrispondenza degli elementi appartenenti alla pagina principale, per identificare in maniera più efficace il corso a cui appartengono.

3.3 Diagrammi di attività

Per presentare chiaramente il funzionamento di alcuni dei casi d'uso relativi all'applicazione è possibile utilizzare i diagrammi di attività.

Un *diagramma di attività* permette di descrivere un processo attraverso dei grafi, in cui i nodi rappresentano le attività e gli archi l'ordine con cui vengono eseguite. Il loro funzionamento è simile a quello di una flowchart, la quale infatti serve allo stesso scopo, ovvero quello di descrivere, in maniera semi-formale, degli algoritmi o procedure di una certa complessità.

Nelle Figure 3.11 - 3.14 vengono illustrati i diagrammi di attività relativi ad alcuni dei caso d'uso più complessi dell'applicazione.

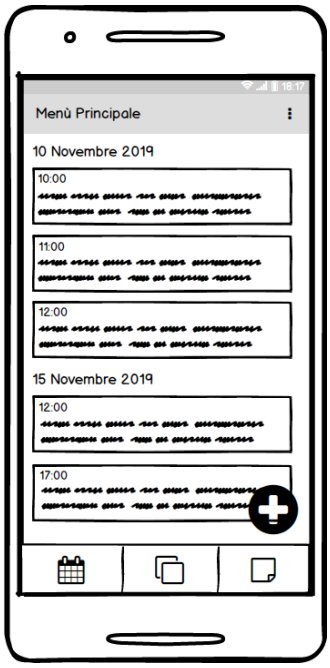

Figura 3.2. Mockup della sezione per la gestione degli eventi (senza calendario)

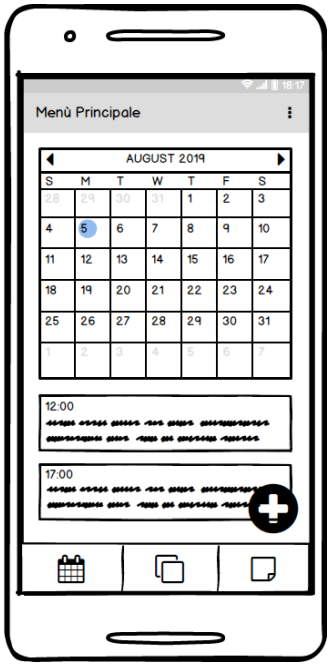

Figura 3.3. Mockup della sezione per la gestione degli eventi (con calendario)

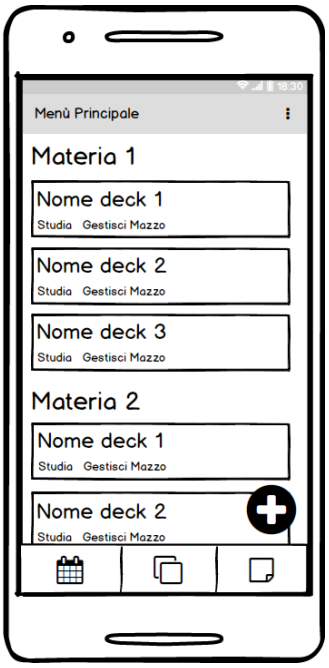

Figura 3.4. Mockup della sezione per la gestione dei mazzi

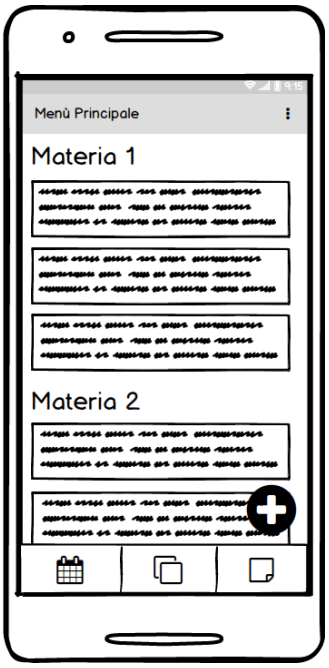

Figura 3.5. Mockup della sezione per la gestione delle note

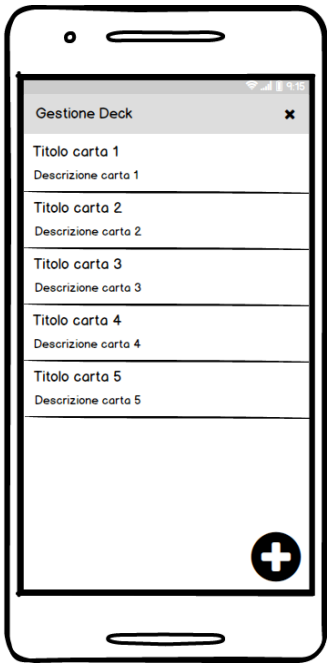

Figura 3.6. Schermata usata per la gestione delle carte

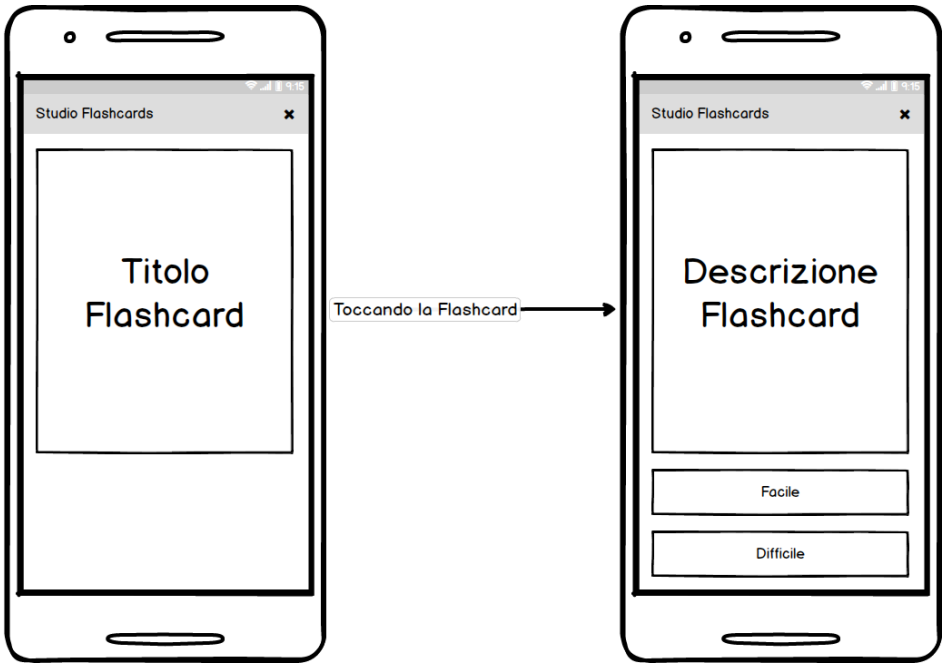

Figura 3.7. Schermata che permette lo studio tramite flashcard

Figura 3.8. Schermata delle impostazioni dell'app

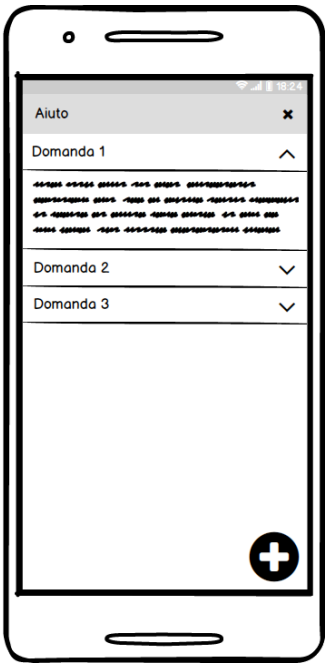

Figura 3.9. Schermata di aiuto dell'app

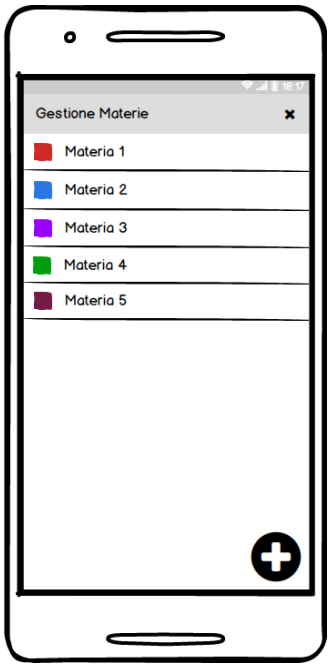

Figura 3.10. Schermata che permette la gestione delle materie

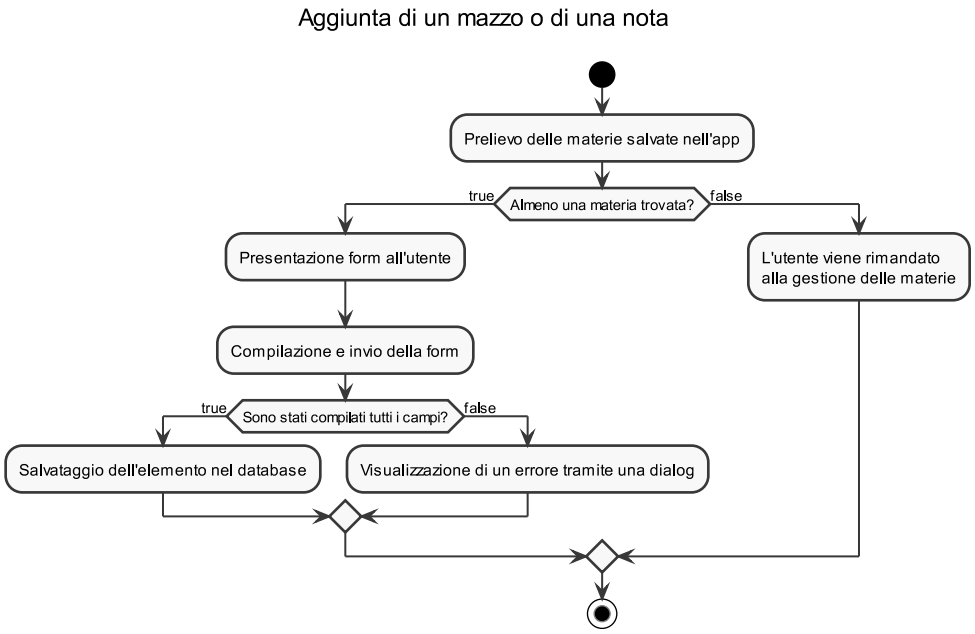

Figura 3.11. Diagramma di attività riguardante l'aggiunta di un mazzo o di una nota

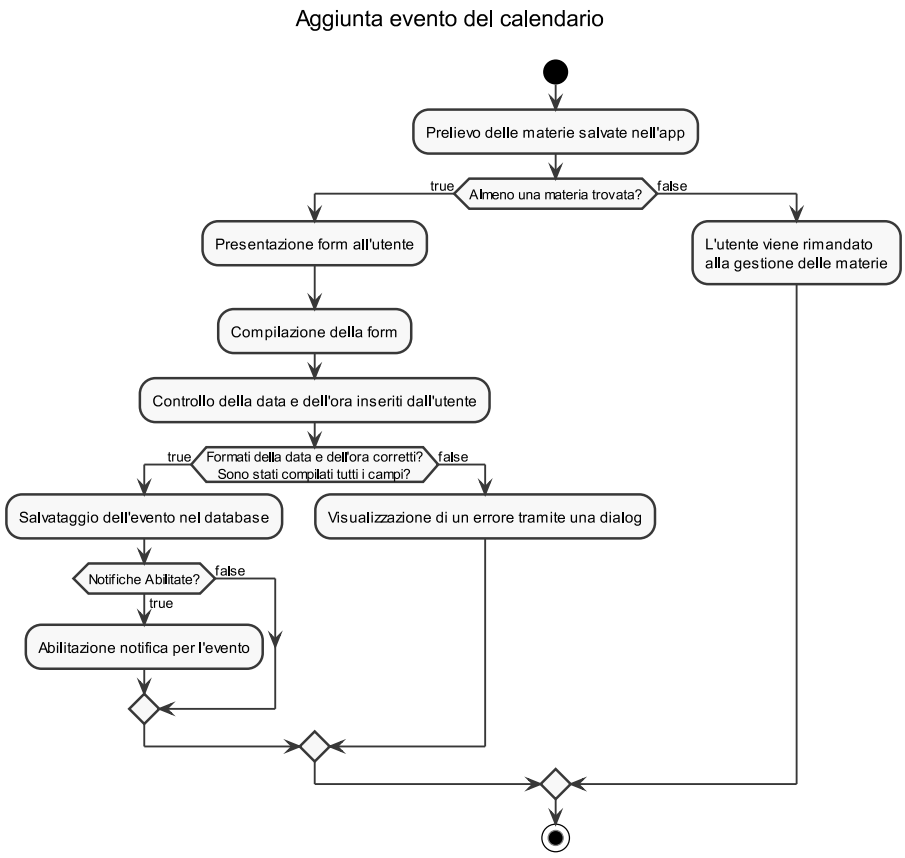

Figura 3.12. Diagramma di attività riguardante l'aggiunta di un evento

3.4 Progettazione dei dati

Affinché il funzionamento dell'applicazione sia garantito, sarà necessario memorizzare i dati che sono d'interesse, in modo da avere la persistenza dei dati una volta che l'applicazione viene chiusa oppure non è in uso. Per questa applicazione si farà uso, in particolare, di un *database SQLite*, che permetterà sia la conservazione dei dati che il loro prelievo.

Per poter realizzare tale database, tuttavia, sarà necessario, innanzitutto, procedere con la sua progettazione, volta ad ottenere la rappresentazione dei dati d'interesse nel modello relazionale, rispettando determinati requisiti funzionali e prestazionali.

Verranno, ora, illustrate le fasi salienti della progettazione della componente dati dell'applicazione. La prima fase che verrà descritta è la *progettazione concettuale*, in cui le informazioni richieste dall'applicazione verranno rappresentate in un modello concettuale, che permette l'astrazione di dati relativi alla realtà di interesse. La seconda fase è la *progettazione logica*, in cui si tradurrà il modello ottenuto dalla

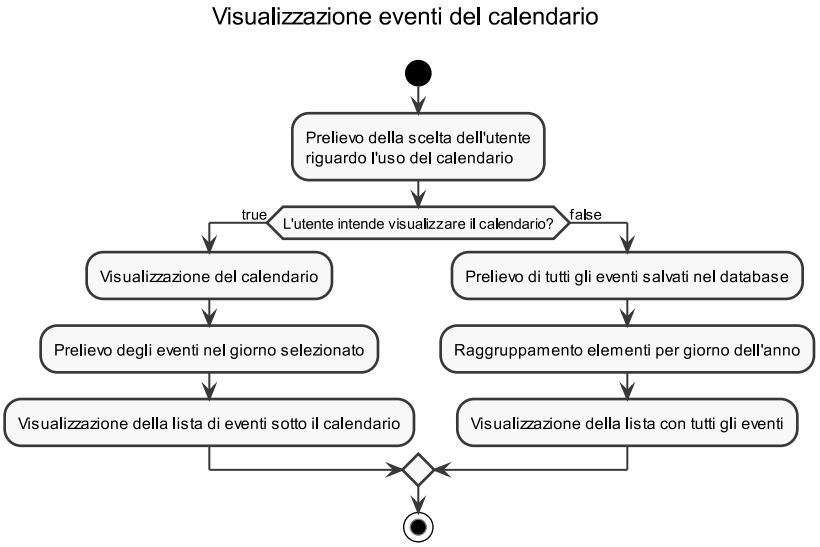

Figura 3.13. Diagramma di attività riguardante la visualizzazione degli eventi

progettazione concettuale in un modello relazionale, che potrà essere implementato direttamente nel database SQLite.

3.4.1 Progettazione concettuale

Durante la fase di *progettazione concettuale*, come precedentemente accennato, vengono rappresentati i dati d'interesse mediante un *modello concettuale*, usato per astrarre i concetti fondamentali da gestire, senza preoccuparsi della loro rappresentazione in un modello logico.

Lo schema utilizzato per la rappresentare le informazioni di interesse in questo caso è lo *Schema E/R* (Entity/Relationship). Con tale modello, mediante dei costrutti basilari, quali l'*entità* (che rappresenta una classe di oggetti aventi stesse proprietà) o la *relazione* (che astrae il concetto di associazione, che collega più istanze di entità diverse), è possibile avere una prima visione d'insieme dei dati d'interesse.

Nella Figura 3.15 viene riportato lo schema E/R relativo alla realtà di interesse dell'applicazione.

Dallo schema è possibile dedurre le entità principali per l'applicazione, ovvero:

- l'entità *Materia*, che rappresenta una materia qualsiasi, definita dall'utente;
- l'entità *Evento*, che rappresenta un evento del calendario;
- l'entità *Mazzo*, che rappresenta un mazzo, in cui verranno inserite delle flashcard;
- l'entità *Carta*, che rappresenta una flashcard;
- l'entità *Nota*, che rappresenta una nota, riguardante una delle materie.

Per ciascuna di queste entità sono illustrati gli attributi di rilievo, che serviranno all'applicazione per funzionare correttamente. In particolare:

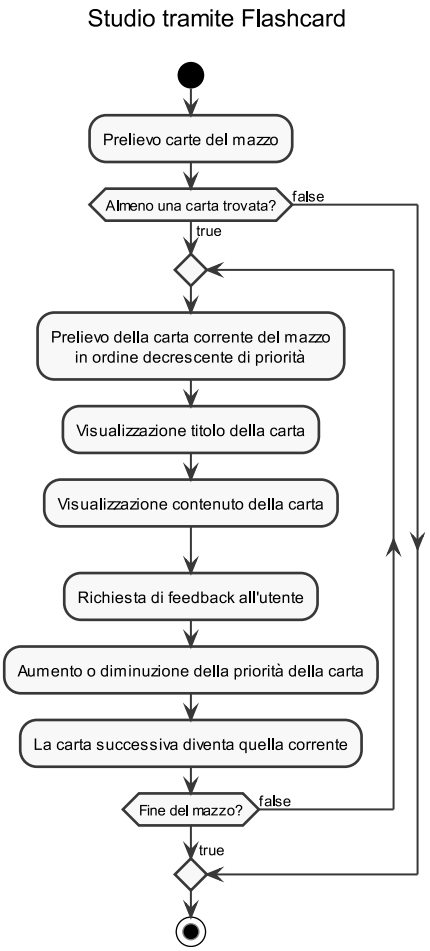

Figura 3.14. Diagramma di attività riguardante lo studio tramite flashcard

- Per quanto riguarda l'entità *Materia* si è deciso di considerare principalmente il *nome* della materia, che ne costituisce anche la chiave primaria, e il *colore* con cui questa materia verrà rappresentata nell'applicazione, memorizzato in formato esadecimale.
- Per quanto riguarda l'entità *Evento* si è deciso di considerare la *data* e l'*ora* in cui l'evento avviene, insieme ad una *descrizione* che specifica il tipo di evento, e ad un campo *notificato*, usato per verificare se l'evento è già stato notificato o meno. L'identificatore dell'entità è, infine, dato da un *id*, che, in questo caso, si è dovuto includere per garantire un corretto funzionamento del sistema di notifiche.
- Per quanto riguarda l'entità *Mazzo* sono stati considerati il suo *nome* e la sua *descrizione*, in modo da offrire all'utente un meccanismo per effettuare una distinzione accurata tra i vari mazzi. La chiave primaria di questa entità è rappresentata dal nome del mazzo e dalla materia a cui è associata.

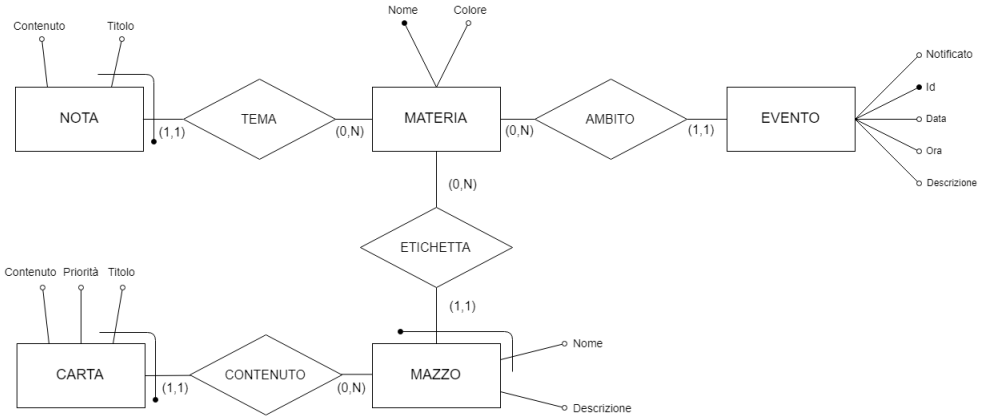

Figura 3.15. Schema E/R che modella i dati dell'applicazione

- Per quanto riguarda l'entità *Carta* si è deciso di considerare il *titolo* e il *contenuto* della flashcard, usati durante la fase di studio, assieme a un attributo *priorità*, che servirà all'applicazione per mostrare le carte usate durante la fase di studio in un ordine preciso, a seconda della priorità che la carta stessa ha. L'identificatore della carta è dato dal suo titolo e dal mazzo a cui si riferisce.
- Per quanto riguarda l'entità *Nota* si è deciso di considerare il *titolo* e il *contenuto* della nota. Anche in questo caso l'identificatore dell'entità è costituito dal titolo, assieme alla materia a cui è associata.

Si sono, infine, rappresentati, oltre agli attributi di ogni entità, i collegamenti presenti tra le entità stesse sotto forma di relazioni, ciascuna con delle cardinalità che descrivono il numero di associazioni che possono essere presenti tra istanze di entità diverse tra di loro.

Vincoli di integrità e dizionario dei dati

Uno schema E/R viene solitamente corredato di una documentazione, che descrive in modo conciso i dati di interesse rappresentati in essa, al fine di consentire una sua comprensione immediata. Solitamente la documentazione è costituita da due componenti:

1. Il *dizionario dei dati*, usato per descrivere le entità ed, eventualmente, le relazioni che le caratterizzano. Nel caso dello schema E/R di nostro interesse, il dizionario dei dati è riportato nella Tabella 3.1
2. I *vincoli di integrità*, che descrivono i vincoli che il database dovrà rispettare per il suo corretto funzionamento.

Per quanto riguarda l'applicazione di nostro interesse, vi è un solo vincolo di integrità da considerare, ovvero:

1. L'attributo *Priorità* dell'entità *Carta* deve essere maggiore o uguale a 0.

Nome entità	*Descrizione*	*Attributi*	*Identificatore*
Materia	Disciplina scolastica o di studio utilizzata dall'app per catalogare i suoi elementi.	Nome (stringa), Colore (stringa)	Nome
Evento	Fatto, o avvenimento, che si verificherà in un determinato istante, definito dall'utente tramite una data e un'ora.	Id (numerico), Data (stringa), Ora (stringa), Descrizione (stringa), Notificato (numerico)	Id
Mazzo	Insieme di flashcard, raggruppate tematicamente, usato per suddividere le carte in maniera appropriata.	Nome (stringa), Descrizione (stringa)	Nome della materia, Nome del mazzo
Carta	Entità che rappresenta un concetto da imparare; è caratterizzata da un titolo relativo al concetto e da un contenuto che lo descrive.	Titolo (stringa), Contenuto (stringa)	Nome della materia, Nome del mazzo, Titolo
Nota	Un insieme di informazioni relative ad una materia, che possono essere di vario tipo (organizzative, informative, etc.).	Titolo (stringa), Contenuto (stringa)	Nome della materia, Titolo

Tabella 3.1. Dizionario dei dati che descrive le entità dello schema E-R dell'applicazione

3.4.2 Progettazione logica

Nella fase di *progettazione logica* lo schema E/R, prodotto durante la progettazione concettuale, viene tradotto in un *modello relazionale*, che consente di rappresentare i dati di interesse in modo tale da poter essere facilmente gestiti tramite un DBMS relazionale. Solitamente, prima di procedere alla traduzione nel modello relazionale, è necessario eseguire i seguenti passi preliminari:

- *Analisi delle ridondanze*, in cui si rilevano gli attributi ridondanti e li si analizzano, per decidere se mantenerli oppure no, tenendo conto del possibile peggioramento delle prestazioni che si avrebbe durante le varie operazioni.
- *Eliminazione delle generalizzazioni*, in cui si sostituiscono tutte le generalizzazioni presenti nello schema E/R con altri costrutti equivalenti, per permettere la loro traduzione agevole nel modello relazionale.
- *Accorpamento/partizione di entità e associazioni*, in cui si decide, eventualmente, se suddividere o accorpare alcune entità, in base a come queste sono coinvolte nelle varie operazioni.
- *Scelta degli identificatori principali*, in cui si decidono gli identificatori da utilizzare, eventualmente introducendone altri per semplificare delle operazioni.
- *Normalizzazione dello schema E/R*, in cui si fa un'analisi sulla qualità dello schema tramite la procedura di normalizzazione, al fine di evitare ripetizioni che peggiorano considerevolmente la qualità dello schema stesso.

Considerando il fatto che lo schema E/R prodotto durante la fase di progettazione concettuale risulta essere molto semplice e già ottimizzato, essendo usato solo per memorizzare le informazioni necessarie all'applicazione, i passi sopra elencati non sono necessari, per cui si può passare direttamente alla fase di *traduzione nel*

modello relazionale in cui, a partire dallo schema E/R ristrutturato, si ottiene la struttura che i dati devono avere nel modello relazionale. Le tabelle ottenute a valle dall'attività di traduzione nel modello relazionale vengono mostrate nella Tabella 3.2

`MATERIA`(Nome, Colore)
`EVENTO`(Id, Materia, Data, Ora, Descrizione, Notificato)
`MAZZO`(Materia, Nome, Descrizione)
`CARTA`(Materia, Mazzo, Titolo, Contenuto, Priorità)
`NOTA`(Materia, Titolo, Contenuto)

Tabella 3.2. Schema relazionale del database dell'applicazione

4

Implementazione e manuale dell'utente

In questo capitolo verrà descritta l'implementazione delle pagine e delle funzionalità principali dell'applicazione, illustrando in particolare le componenti utilizzate e il codice che realizza la logica per il corretto funzionamento dell'app.

4.1 Database dell'applicazione

Prima di discutere nello specifico sull'implementazione delle varie schermate che l'applicazione presenta è necessario considerare l'implementazione del database SQ-Lite, visto che buona parte delle funzionalità dell'app faranno riferimento a questo database per prelevare e salvare i dati d'interesse.

Per quanto riguarda l'implementazione del database è stato, innanzitutto, necessario realizzare dei *modelli*, usati dall'applicazione per le informazioni presenti nel database. Grazie a tali modelli è possibile utilizzare i dati di interesse intuitivamente, sfruttando i paradigmi della programmazione orientata agli oggetti.

Nei Listati 4.1 - 4.5 vengono mostrati tutti i modelli che sono stati implementati per l'applicazione, rappresentati nelle classi `CalendarEvent`, `Card`, `Deck`, `Subject` e `Note`. I modelli contengono i campi e metodi necessari per prelevare e inserire le informazioni relative all'oggetto, assieme a dei metodi di supporto usati per ottenere delle informazioni più complesse (come il metodo `getFormattedDate()` della classe `CalendarEvent`, che restituisce la data dell'evento in forma testuale).

```
public class CalendarEvent {
    private int id;
    private String subject;
    private String date;
    private String hour;
    private String description;
    private int notified;

    public CalendarEvent(int id, String subject, String date, String hour, String description, int notified) {
        this.id = id;
        this.notified = notified;
        this.subject = subject;
        this.date = date;
        this.hour = hour;
        this.description = description;
    }

    public int getId() {
        return id;
    }
```

```
    public void setId(int id) {
        this.id = id;
    }

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }

    public String getHour() {
        return hour;
    }

    public void setHour(String hour) {
        this.hour = hour;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public int getNotified() {
        return notified;
    }

    public void setNotified(int notified) {
        this.notified = notified;
    }

    public String getFormattedDate(){
        String[] values = date.split("/");
        return values[2] + " " + FormatMonth(values[1]) + " " + values[0];
    }

    private String FormatMonth(String month){
        switch(month){
            case "01":
                return "Gennaio";
            case "02":
                return "Febbraio";
            case "03":
                return "Marzo";
            case "04":
                return "Aprile";
            case "05":
                return "Maggio";
            case "06":
                return "Giugno";
            case "07":
                return "Luglio";
            case "08":
                return "Agosto";
            case "09":
                return "Settembre";
            case "10":
                return "Ottobre";
            case "11":
                return "Novembre";
            case "12":
                return "Dicembre";
            default:
                Log.e("ERROR:", "Month not found");
                return "";
        }
    }
}
```

Listato 4.1. Definizione del modello `CalendarEvent`

```
public class Deck {
    private String subject;
    private String description;
    private String name;
```

```
    public Deck(String subject, String description, String name) {
        this.subject = subject;
        this.description = description;
        this.name = name;
    }

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Listato 4.2. Definizione del modello `Deck`

```
    public class Card implements Parcelable {
        private String subject;
        private final String deckName;
        private String title;
        private String content;
        private int priority;

        public Card(String subject, String deckName, String title, String content, int priority) {
            this.subject = subject;
            this.deckName = deckName;
            this.title = title;
            this.content = content;
            this.priority = priority;
        }

        private Card(Parcel in){
            subject = in.readString();
            deckName = in.readString();
            title = in.readString();
            content = in.readString();
            priority = in.readInt();
        }

        public String getSubject() {
            return subject;
        }

        public void setSubject(String materia) {
            this.subject = materia;
        }

        public String getDeckName() {
            return deckName;
        }

        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        public String getContent() {
            return content;
        }

        public void setContent(String content) {
            this.content = content;
        }

        public int getPriority() {
            return priority;
        }

        public void setPriority(int priority) {
```

```
        this.priority = priority;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(subject);
        dest.writeString(deckName);
        dest.writeString(title);
        dest.writeString(content);
        dest.writeInt(priority);
    }

    public static final Parcelable.Creator<Card> CREATOR = new Parcelable.Creator<Card>(){
        @Override
        public Card createFromParcel(Parcel source) {
            return new Card(source);
        }

        @Override
        public Card[] newArray(int size) {
            return new Card[size];
        }
    };
}
```

Listato 4.3. Definizione del modello `Card`

```
public class Note {
    private String subject;
    private String title;
    private String content;

    public Note(String subject, String title, String content) {
        this.subject = subject;
        this.title = title;
        this.content = content;
    }

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

Listato 4.4. Definizione del modello `Note`

```
public class Subject {
    private String name;
    private String color;

    public Subject(String name){
        this.name = name;
        this.color = "";
    }

    public Subject(String name, String color) {
        this.name = name;
        this.color = color;
    }

    public String getName() {
```

```
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public String getColor() {
            return color;
        }

        public void setColor(String color) {
            this.color = color;
        }

        @Override
        public boolean equals(Object obj) {
            if(obj instanceof String){
                return name.equals(obj);
            }
            return super.equals(obj);
        }
    }
```

Listato 4.5. Definizione del modello `Subject`

Dopo aver definito i modelli è, ora, necessario definire le tabelle e le query usate per crearle e distruggerle, che verranno, in seguito, utilizzate dal DBMS SQLite di Android. A questo fine si è introdotto un *Contract*, ovvero una classe usata come contenitore per le tabelle e le loro colonne. In questo modo è possibile separare la struttura della tabella dalle sue operazioni, permettendo al programmatore, per esempio, di modificare il nome di alcune delle colonne della tabella senza alcuna ripercussione sulla logica delle query.

Il Listato 4.6 mostra la classe `DBContract`, che implementa il Contract relativo al database dell'app. È possibile notare come ogni tabella presente nel Contract è modellata tramite una classe statica, che implementa l'interfaccia `BaseColumns` (interfaccia che offre alcuni campi aggiuntivi, utilizzabili dal database). Queste classi, assieme ai loro attributi, potranno, in seguito, essere accedute da qualsiasi parte del programma.

```
class DBContract {
    private DBContract(){}

    public static class SubjectEntry implements BaseColumns{
        public static final String TABLE_NAME = "subject";
        public static final String COLUMN_NAME_SUBJNAME = "name";
        public static final String COLUMN_NAME_COLOR = "color";
        public static final String SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS " + TABLE_NAME;
        public static final String SQL_CREATE_TABLE =
                "CREATE TABLE " + TABLE_NAME + "(" + COLUMN_NAME_SUBJNAME
                + " TEXT PRIMARY KEY, " + COLUMN_NAME_COLOR + " TEXT)";
    }

    public static class EventEntry implements BaseColumns{
        public static final String TABLE_NAME = "event";
        public static final String COLUMN_NAME_SUBJECT = "subject";
        public static final String COLUMN_NAME_DATE = "date";
        public static final String COLUMN_NAME_HOUR = "hour";
        public static final String COLUMN_NAME_DESCRIPTION = "description";
        public static final String COLUMN_NAME_NOTIFIED = "notified";
        public static final String SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS " + TABLE_NAME;
        public static final String SQL_CREATE_TABLE =
                "CREATE TABLE " + TABLE_NAME + "(" + _ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
                COLUMN_NAME_SUBJECT + " TEXT," + COLUMN_NAME_DESCRIPTION + " TEXT," + COLUMN_NAME_DATE
                + " TEXT," + COLUMN_NAME_HOUR + " TEXT, " + COLUMN_NAME_NOTIFIED + " INTEGER, FOREIGN KEY("
                + COLUMN_NAME_SUBJECT + ") REFERENCES " + SubjectEntry.TABLE_NAME + "(" +
                SubjectEntry.COLUMN_NAME_SUBJNAME + ") ON DELETE CASCADE ON UPDATE CASCADE)";
    }

    public static class DeckEntry implements BaseColumns{
        public static final String TABLE_NAME = "deck";
        public static final String COLUMN_NAME_SUBJECT = "subject";
        public static final String COLUMN_NAME_DECKNAME = "name";
        public static final String COLUMN_NAME_DESCRIPTION = "description";
        public static final String SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS " + TABLE_NAME;
        public static final String SQL_CREATE_TABLE =
                "CREATE TABLE " + TABLE_NAME + "(" + COLUMN_NAME_SUBJECT + " TEXT," + COLUMN_NAME_DECKNAME +
```

```
            " TEXT," + COLUMN_NAME_DESCRIPTION + " TEXT, PRIMARY KEY(" + COLUMN_NAME_SUBJECT + "," +
            COLUMN_NAME_DECKNAME + "), " + "FOREIGN KEY(" + COLUMN_NAME_SUBJECT + ") REFERENCES " +
            SubjectEntry.TABLE_NAME + "(" + SubjectEntry.COLUMN_NAME_SUBJNAME + ") ON DELETE CASCADE ON
            UPDATE CASCADE)";
    }

    public static class CardEntry implements BaseColumns{
        public static final String TABLE_NAME = "card";
        public static final String COLUMN_NAME_SUBJECT = "subject";
        public static final String COLUMN_NAME_DECK = "deck";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_CONTENT = "content";
        public static final String COLUMN_NAME_PRIORITY = "priority";
        public static final String SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS " + TABLE_NAME;
        public static final String SQL_CREATE_TABLE =
                "CREATE TABLE " + TABLE_NAME + "(" + COLUMN_NAME_SUBJECT + " TEXT," + COLUMN_NAME_DECK + " TEXT,"
                + COLUMN_NAME_TITLE + " TEXT," + COLUMN_NAME_CONTENT + " TEXT, " + COLUMN_NAME_PRIORITY +
                " INT DEFAULT 0, " + "PRIMARY KEY(" + COLUMN_NAME_SUBJECT + "," + COLUMN_NAME_DECK + "," +
                COLUMN_NAME_TITLE + "), " + "FOREIGN KEY(" + COLUMN_NAME_SUBJECT + ", " + COLUMN_NAME_DECK + ")
                REFERENCES " + DeckEntry.TABLE_NAME + "(" + DeckEntry.COLUMN_NAME_SUBJECT + ", " +
                DeckEntry.COLUMN_NAME_DECKNAME + ") ON DELETE CASCADE ON UPDATE CASCADE)";
    }

    public static class NoteEntry implements BaseColumns{
        public static final String TABLE_NAME = "note";
        public static final String COLUMN_NAME_SUBJECT = "subject";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_CONTENT = "content";
        public static final String SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS " + TABLE_NAME;
        public static final String SQL_CREATE_TABLE =
                "CREATE TABLE " + TABLE_NAME + "(" + COLUMN_NAME_SUBJECT + " TEXT," + COLUMN_NAME_TITLE + " TEXT,"
                + COLUMN_NAME_CONTENT + " TEXT, PRIMARY KEY(" + COLUMN_NAME_SUBJECT + "," + COLUMN_NAME_TITLE
                + "), " + "FOREIGN KEY(" + COLUMN_NAME_SUBJECT + ") REFERENCES " + SubjectEntry.TABLE_NAME +
                "(" + SubjectEntry.COLUMN_NAME_SUBJNAME + ") ON DELETE CASCADE ON UPDATE CASCADE)";
    }
}
```

Listato 4.6. Definizione della classe `DBContract`

L'ultima cosa che rimane è l'implementazione di una classe helper, che si interfaccerà al database e fornirà dei metodi per effettuare le operazioni CRUD sui suoi elementi, che potranno essere richiamate dalle altre parti dell'applicazione. A tal fine, è stata creata una classe `AppDbHelper`; quest'ultima, per invocare le funzionalità relative a SQLite, dovrà estendere la classe `SQLiteOpenHelper`, configurandone le operazioni tramite i metodi sovrascritti ed offrendo le funzionalità CRUD tramite dei metodi che sono stati implementati.

Nel Listato 4.7 sono raffigurati, a titolo di esempio, alcuni dei metodi implementati nella classe `AppDbHelper` per le operazioni CRUD sulle materie. Tutti gli altri metodi saranno strutturati in modo simile, facendo uso di un oggetto di tipo `SQLiteDatabase` che offre tutte le funzionalità del database SQLite.

```
public void insertSubject(Subject subject){
    SQLiteDatabase db = getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(DBContract.SubjectEntry.COLUMN_NAME_SUBJNAME, subject.getName());
    values.put(DBContract.SubjectEntry.COLUMN_NAME_COLOR, subject.getColor());

    db.insert(DBContract.SubjectEntry.TABLE_NAME, null, values);
    db.close();
}

public ArrayList<Subject> getSubjects(){
    ArrayList<Subject> subjects = new ArrayList<>();
    String query = "SELECT * FROM " + DBContract.SubjectEntry.TABLE_NAME;

    SQLiteDatabase db = getWritableDatabase();
    Cursor cursor = db.rawQuery(query, null);

    if(cursor.moveToFirst()){
        do{
            subjects.add(new Subject(
                    cursor.getString(cursor.getColumnIndex(DBContract.SubjectEntry.COLUMN_NAME_SUBJNAME)),
                    cursor.getString(cursor.getColumnIndex(DBContract.SubjectEntry.COLUMN_NAME_COLOR))
            ));
        } while(cursor.moveToNext());
    }

    cursor.close();
```

```
        db.close();
        return subjects;
    }

    public ArrayList<String> getSubjectNames(){
        ArrayList<String> subjNames = new ArrayList<>();
        String query = "SELECT " + DBContract.SubjectEntry.COLUMN_NAME_SUBJNAME + " FROM " +
                DBContract.SubjectEntry.TABLE_NAME;

        SQLiteDatabase db = getWritableDatabase();
        Cursor cursor = db.rawQuery(query, null);

        if(cursor.moveToFirst()){
            do{
                subjNames.add(cursor.getString(cursor.getColumnIndex(DBContract.SubjectEntry.COLUMN_NAME_SUBJNAME)));
            } while(cursor.moveToNext());
        }

        cursor.close();
        db.close();
        return subjNames;
    }

    public boolean deleteSubject(String subjName){
        SQLiteDatabase db = getWritableDatabase();
        boolean delete = db.delete(DBContract.SubjectEntry.TABLE_NAME, DBContract.SubjectEntry.COLUMN_NAME_SUBJNAME
        + " =? ", new String[]{subjName}) > 0;
        db.close();
        return delete;
    }

    public boolean updateSubject(Subject subject, String oldName){
        SQLiteDatabase db = getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put(DBContract.SubjectEntry.COLUMN_NAME_SUBJNAME, subject.getName());
        values.put(DBContract.SubjectEntry.COLUMN_NAME_COLOR, subject.getColor());

        boolean update = db.update(DBContract.SubjectEntry.TABLE_NAME, values,
                DBContract.SubjectEntry.COLUMN_NAME_SUBJNAME + " =? ", new String[]{oldName}) > 0;
        db.close();
        return update;
    }
```

Listato 4.7. Metodi della classe `AppDbHelper` che implementano le operazioni CRUD per le materie

4.2 Pagina principale

La pagina principale realizzata per l'applicazione è visibile nella Figura 4.1. Per implementare il sistema di navigazione tra le sue sezioni, considerato in fase di progettazione, si è fatto uso di una *BottomNavigationView*, che corrisponde alla barra presente sulla parte bassa dell'applicazione, assieme a dei *Fragment*, che rappresentano una porzione dell'interfaccia utente della pagina. Scegliendo le opzioni all'interno della BottomNavigationView è possibile cambiare il Fragment che è al momento visibile, consentendo all'utente di scegliere una delle sezioni della pagina.

Nel Listato 4.8 viene mostrata l'implementazione della classe `MainActivity`, che rappresenta l'Activity associata a questa pagina. È possibile notare, in particolare, come l'Activity, dopo aver inizializzato le sue componenti e prelevato le impostazioni nel metodo `onResume()`, sceglie il Fragment da mostrare all'utente tramite il metodo `renderFragment()`, in base all'ultima sezione visitata (salvata come impostazione dell'app), all'opzione selezionata nella BottomNavigationView, o prelevando direttamente il Fragment che era precedentemente visibile, nel caso in cui l'Activity viene ricostruita in seguito ad un cambio di orientamento.

Un altro metodo di rilievo, presente nella classe, è `showSubjectNotice()`, usato per mostrare una notifica tramite una *snackbar*, nel caso in cui cerchi di aggiungere

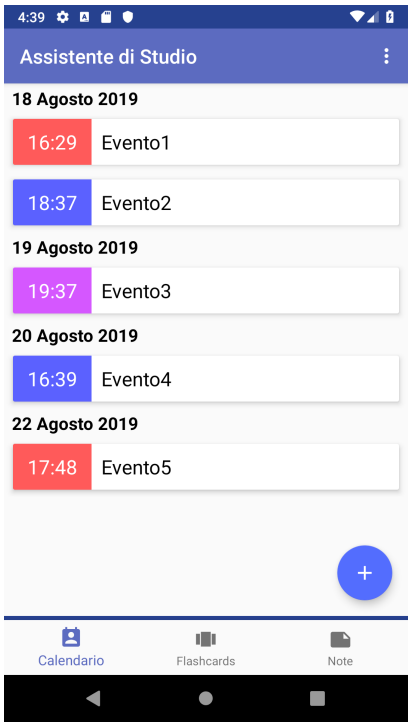

Figura 4.1. Screenshot della pagina principale dell'applicazione, in cui è visibile la sezione dedicata alla gestione degli eventi

un elemento qualsiasi in assenza di materie. Anche il metodo `showFlashcardsNotice()` viene usato per mostrare una snackbar, ma viene chiamato nel caso in cui si cerca di avviare l'attività di studio su un mazzo che non ha carte.

Oltre alle funzionalità sopra descritte sono anche presenti metodi usati per avviare le altre Activity, corrispondenti alle altre pagine dell'app, insieme a metodi che implementano le funzionalità della toolbar dell'applicazione (da cui è possibile accedere alla pagina per la gestione delle materie, alla pagina delle impostazioni o alla pagina di aiuto).

```
public class MainActivity extends AppCompatActivity {
    private BottomNavigationView bottomNavigationView;
    private Boolean calendarPreference;
    private SharedPreferences sharedPref;
    private int fragmentId;
    private Fragment selectedFragment;
    private Fragment currentFragment;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Objects.requireNonNull(getSupportActionBar()).setTitle("Assistente di Studio");
        bottomNavigationView = findViewById(R.id.bottom_navigation);
        bottomNavigationView.setOnNavigationItemSelectedListener(
            new BottomNavigationView.OnNavigationItemSelectedListener() {
            @Override
            public boolean onNavigationItemSelected(@NonNull MenuItem menuItem) {
                showFragment(menuItem.getItemId());
                return true;
            }
        });
```

```
    android.support.v7.preference.PreferenceManager.setDefaultValues(this, R.xml.pref_main, false);
}

@Override
protected void onResume(){
    super.onResume();
    getSharedPref(this);
    renderFragment();
}

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

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_activity_menu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()){
        case R.id.subject_menu: {
            startSubjectManagementActivity();
            return true;
        }
        case R.id.settings:
            Intent i1 = new Intent(this, SettingsActivity.class);
            startActivity(i1);
            return true;
        case R.id.help:
            Intent i2 = new Intent(this, HelpActivity.class);
            startActivity(i2);
            return true;
        default:
            Toast.makeText(this, "Default", Toast.LENGTH_SHORT).show();
    }
    return true;
}

private int menuSelector(int id){

    currentFragment = getSupportFragmentManager().findFragmentByTag("current_fragment");
    selectedFragment = null;

    if (id == 0 && fragmentId != 0) {
        id = fragmentId;
    }

    switch(id){
        case R.id.calendar_menu:
            if(calendarPreference){
                selectedFragment = new CalendarFragmentWithCalendar();
            } else {
                selectedFragment = new CalendarFragment();
            }
            break;
        case R.id.flashcards_menu:
            selectedFragment = new FlashcardsFragment();
            break;
        case R.id.notes_menu:
            selectedFragment = new NotesFragment();
            break;
        default:
            id = R.id.calendar_menu;
    }
    return id;
}

private void renderFragment(){
    showFragment(fragmentId);
}

private void showFragment(int id){
    int tempId = menuSelector(id);
    bottomNavigationView.getMenu().findItem(tempId).setChecked(true);
    if (currentFragment == null || (selectedFragment != null &&
            !currentFragment.getClass().equals(selectedFragment.getClass()))){
        getSupportFragmentManager().beginTransaction()
                .replace(R.id.fragment_container, selectedFragment, "current_fragment").commit();
        fragmentId = bottomNavigationView.getSelectedItemId();
    }
}

private void getSharedPref(Context context){
    sharedPref = android.support.v7.preference.PreferenceManager.getDefaultSharedPreferences(context);
    calendarPreference = sharedPref.getBoolean(SettingsActivity.CALENDAR_KEY, false);
    boolean notificationPreference = sharedPref.getBoolean(SettingsActivity.NOTIFICATION_KEY, false);
    fragmentId = sharedPref.getInt("fragmentPref", bottomNavigationView.getMenu().getItem(0).getItemId());
```

```
            if(notificationPreference) {
                AlarmReceiver.setupAlarm(this, notificationPreference, false);
            }
        }

        private void saveFragment(){
            SharedPreferences.Editor editor = sharedPref.edit();
            editor.putInt("fragmentPref", fragmentId);
            editor.apply();
        }

        public void showSubjectNotice(){
            Snackbar s = Snackbar.make(
                    findViewById(R.id.fragment_container),
                    "Devi aggiungere una materia prima di proseguire!",
                    Snackbar.LENGTH_LONG
            ).setAction("Gestisci Materie", new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    startSubjectManagementActivity();
                }
            });
            s.show();
        }

        public void showFlashcardsNotice(final String subject, final String deckName){
            Snackbar s = Snackbar.make(
                    findViewById(R.id.fragment_container),
                    "Devi aggiungere almeno una carta prima di proseguire!",
                    Snackbar.LENGTH_LONG
            ).setAction("Gestisci Mazzo", new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    startDeckManagementActivity(subject, deckName);
                }
            });
            s.show();
        }

        private void startSubjectManagementActivity(){
            Intent i = new Intent(this, SubjectManagementActivity.class);
            startActivity(i);
        }

        private void startDeckManagementActivity(String subject, String deckName){
            Intent i = new Intent(this, DeckManagementActivity.class);

            Bundle bundle = new Bundle();
            bundle.putString("subject", subject);
            bundle.putString("deck_name", deckName);
            i.putExtra("deck_data", bundle);

            startActivity(i);
        }
    }
```

Listato 4.8. Definizione della classe `MainActivity`

Dopo aver descritto l'implementazione degli elementi della pagina principale, verranno, ora, descritti nello specifico le sezioni in cui la pagina è suddivisa.

4.2.1 Gestione degli eventi

Come è già stato accennato durante la fase di progettazione, sono presenti due tipi di layout per quanto riguarda la gestione degli eventi: il primo, mostrato nella Figura 4.1, presenta una lista di eventi ordinati in base alla data, mentre il secondo, illustrato nella Figura 4.2, presenta un calendario e una lista per gli eventi programmati per il giorno selezionato. Alle due schermate sono stati associati due Fragment diversi e, in base all'impostazione selezionata dall'utente, verrà scelto il Fragment associato alla schermata che l'utente intende usare.

Per quanto riguarda la gestione degli eventi senza l'utilizzo del calendario, vengono prima di tutto prelevati tutti gli eventi, ordinati per data e ora, dopodiché viene utilizzata una *RecyclerView* (una versione più avanzata e flessibile di una ListView, che garantisce il riuso degli elementi risparmiando sulle prestazioni) per mostrare tali eventi sullo schermo. Per raggruppare gli eventi in base al giorno, è stata realizzata

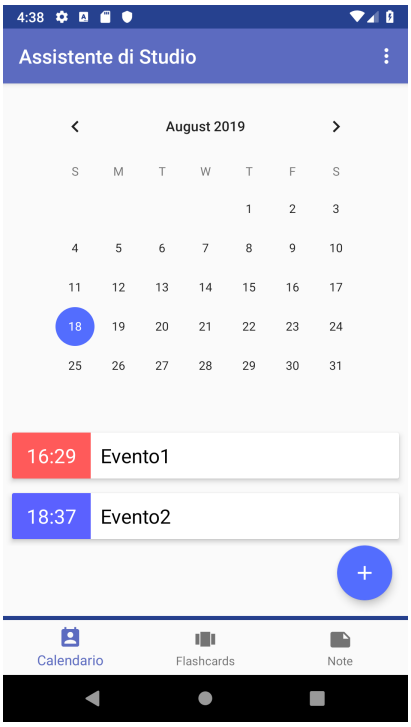

Figura 4.2. Screenshot della sezione riguardante la gestione degli eventi (con calendario)

un'implementazione dell'adapter (responsabile per l'interfacciamento della sorgente di dati con la view) con degli elementi personalizzati. Gli elementi della RecyclerView, infatti, presentano due tipi layout: il primo è quello tradizionale, che contiene l'elemento da visualizzare, mentre il secondo contiene, oltre all'elemento, anche il titolo del gruppo a cui appartiene. Alternando in modo appropriato questi due layout tramite l'adapter è quindi possibile effettivamente creare dei raggruppamenti per gli elementi della lista degli eventi.

Il Listato 4.9 mostra l'implementazione dell'adapter relativo alla RecyclerView. È possibile notare come, tramite il metodo `getItemViewType()`, si possa decidere il tipo di layout da assegnare all'elemento specifico (in questo caso, avendo una lista di eventi ordinati rispetto alla data, il layout viene cambiato ogni volta che il giorno dell'evento corrente è diverso rispetto a quello precedente). In questo modo si ha che ad ogni elemento è associato un ViewHolder specifico, che contiene il suo layout. Tramite il metodo `onCreateViewHolder()` viene, quindi, istanziato il ViewHolder specifico per l'elemento, che ne modella il contenuto, e, nel metodo `onBindViewHolder()`, viene impostata la view che gli è stata assegnata. Ad ogni elemento è associato, anche, un menù contestuale, definito nel metodo `createContextMenu()`, con il quale è possibile aggiungere le opzioni di modifica ed eliminazione, che saranno, in seguito, gestite dal Fragment. È possibile, infine, notare come ad ogni elemento della lista viene associato il colore della propria materia. Questo collegamento avviene nel metodo `onBindViewHolder()`, che chiama il metodo `findColor()` per prelevare il

colore associato alla materia.

```
class CalendarRVAdapter extends RecyclerView.Adapter<RecyclerView.ViewHolder> {

    private final ArrayList<CalendarEvent> eventList;
    private final ArrayList<Subject> subjects;
    private static final int TYPE_ITEM = 0;
    private static final int TYPE_HEADER = 1;
    private final boolean hasCalendar;

    class CalendarViewHolder extends RecyclerView.ViewHolder implements View.OnCreateContextMenuListener {
        final TextView hourTextView;
        final TextView descriptionTextView;

        CalendarViewHolder(@NonNull View itemView) {
            super(itemView);
            hourTextView = itemView.findViewById(R.id.calendar_hour);
            descriptionTextView = itemView.findViewById(R.id.calendar_description);
            itemView.setOnCreateContextMenuListener(this);
        }

        @Override
        public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo) {
            createContextMenu(menu, this);
        }

        void bind(String hour, String description, String color){
            hourTextView.setBackgroundColor(Color.parseColor(color));
            hourTextView.getBackground().setAlpha(168);
            hourTextView.setText(hour);
            descriptionTextView.setText(description);
        }
    }

    class CalendarHeaderViewHolder extends RecyclerView.ViewHolder implements View.OnCreateContextMenuListener {
        final TextView hourTextView;
        final TextView descriptionTextView;
        final CardView eventCardView;
        final TextView dateHeader;

        CalendarHeaderViewHolder(@NonNull View itemView) {
            super(itemView);
            dateHeader = itemView.findViewById(R.id.date_header);
            eventCardView = itemView.findViewById(R.id.calendar_cv);
            hourTextView = itemView.findViewById(R.id.calendar_hour);
            descriptionTextView = itemView.findViewById(R.id.calendar_description);
            eventCardView.setOnCreateContextMenuListener(this);
        }

        @Override
        public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo) {
            createContextMenu(menu, this);
        }

        void bind(String date, String hour, String description, String color){
            hourTextView.setBackgroundColor(Color.parseColor(color));
            hourTextView.getBackground().setAlpha(168);
            hourTextView.setText(hour);
            descriptionTextView.setText(description);
            dateHeader.setText(date);
        }
    }

    CalendarRVAdapter(ArrayList<CalendarEvent> eventList, ArrayList<Subject> subjects, boolean hasCalendar){
        this.eventList = eventList;
        this.subjects = subjects;
        this.hasCalendar = hasCalendar;
    }

    @Override
    public int getItemViewType(int position) {
        if(hasCalendar || (position != 0 &&
                eventList.get(position - 1).getDate().equals(eventList.get(position).getDate()))) {
            return TYPE_ITEM;
        } else {
            return TYPE_HEADER;
        }
    }

    @Override
    public int getItemCount() {
        return eventList.size();
    }

    @Override
    public void onBindViewHolder(@NonNull RecyclerView.ViewHolder calendarViewHolder, int i) {
        CalendarEvent current = eventList.get(i);
        String color = findColor(eventList.get(i).getSubject());
        if(calendarViewHolder instanceof CalendarViewHolder){
            ((CalendarViewHolder) calendarViewHolder).bind(current.getHour(), current.getDescription(), color);
        } else {
            ((CalendarHeaderViewHolder) calendarViewHolder)
```

```
                    .bind(current.getFormattedDate(), current.getHour(), current.getDescription(), color);
            }
        }

        @NonNull
        @Override
        public RecyclerView.ViewHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int viewType) {
            View v;
            RecyclerView.ViewHolder cvh;
            if(viewType == TYPE_ITEM){
                v = LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.calendar_item, viewGroup, false);
                cvh = new CalendarViewHolder(v);
            } else {
                v = LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.calendar_header, viewGroup, false);
                cvh = new CalendarHeaderViewHolder(v);
            }
            return cvh;
        }

        @Override
        public void onAttachedToRecyclerView(@NonNull RecyclerView recyclerView) {
            super.onAttachedToRecyclerView(recyclerView);
        }

        private void createContextMenu(ContextMenu menu, RecyclerView.ViewHolder holder){
            menu.setHeaderTitle("Opzioni Evento");
            menu.add(holder.getAdapterPosition(), 0, 0, "Modifica");
            menu.add(holder.getAdapterPosition(), 1, 1, "Elimina");
        }

        private String findColor(String subjectName){
            for(Subject subject : subjects){
                if(subject.getName().equals(subjectName))
                    return subject.getColor();
            }
            return null;
        }
    }
```

Listato 4.9. Definizione della classe `CalendarRVAdapter`

Per quanto riguarda la gestione degli eventi con calendario, invece, è necessario, innanzitutto, prelevare gli eventi previsti per la giornata odierna, essendo il calendario inizialmente selezionato su quel giorno, dopodiché, quando l'utente seleziona la data desiderata dal calendario, la lista degli eventi verrà ricostruita mostrando gli eventi programmati per il giorno selezionato.

Nel Listato 4.10 viene mostrata, in particolare, questa procedura; più specificatamente, nel metodo `onStart()`, al primo avvio, verrà inizializzato l'adapter relativo alla RecyclerView del Fragment e verranno passati ad esso gli eventi previsti per la giornata corrente, mentre, per le richieste successive, l'ArrayList di eventi verrà ripopolata in base alla giornata selezionata, tramite il metodo `refreshRecyclerView()`.

Nel Listato 4.11, invece, viene mostrato l'event listener relativo alla *CalendarView* (usata per realizzare il calendario nel Fragment), che aggiorna la data selezionata e chiama il metodo `refreshRecyclerView()` per aggiornare la lista di eventi con la nuova data.

```
    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        firstVisit = true;
        dbHelper = new AppDbHelper(getContext());
        selectedDate = dateFormat.format(Calendar.getInstance().getTime());
        selectedDateCalendar = Calendar.getInstance();
        super.onCreate(savedInstanceState);
    }

    @Override
    public void onStart() {
        if(firstVisit){
            events = dbHelper.getEventsByDate(selectedDate);
            subjects = dbHelper.getSubjects();
            adapter = new CalendarRVAdapter(events, subjects, true);
            rv.setAdapter(adapter);
            firstVisit = false;
        } else {
```

```
        subjects.clear();
        subjects.addAll(dbHelper.getSubjects());
        refreshRecyclerView();
    }
    super.onStart();
}

private void refreshRecyclerView(){
    events.clear();
    events.addAll(dbHelper.getEventsByDate(selectedDate));
    adapter.notifyDataSetChanged();
}
```

Listato 4.10. Metodi della classe `CalendarFragmentWithCalendar` usati per l'inizializzazione della lista degli eventi

```
calendarView.setOnDateChangeListener(new CalendarView.OnDateChangeListener() {
        @Override
        public void onSelectedDayChange(CalendarView view, int year, int month, int dayOfMonth) {
            selectedDateCalendar.set(year, month, dayOfMonth);
            selectedDate = dateFormat.format(selectedDateCalendar.getTime());
            refreshRecyclerView();
        }
});
```

Listato 4.11. Event listener della CalendarView usata per aggiornare la lista di eventi

Tramite il pulsante presente sullo schermo nell'angolo in basso a destra è infine possibile accedere alla schermata di aggiunta di un evento, implementata mediante un'Activity separata, illustrata nel Listato 4.12.

Tale Activity conterrà, oltre agli elementi necessari per inserire i dati di rilievo (uno *Spinner* per scegliere la materia, e degli *EditText* per inserire data, ora e descrizione), anche un *DatePicker* e un *TimePicker*, che verranno mostrati tramite dei dialog al click dei rispettivi pulsanti, nel caso in cui l'utente voglia usarli per cambiare l'ora o la data dell'evento. Tramite i suoi event listener, implementati nell'Activity mediante i metodi `onDateSet()` e `onTimeSet()`, vengono compilati in modo appropriato i campi relativi alla data e all'ora. All'aggiunta dell'evento vengono prima controllati i dati inseriti, per verificare che siano nel formato corretto (facendo uso del metodo `isValidFormat()`); dopodiché l'elemento viene inserito nel database (tramite il metodo `newOrEdited()`) e si ritorna alla pagina principale. Se la data relativa all'evento è una data passata, viene anche mostrato un dialog per chiedere all'utente se intende effettivamente inserire tale evento.

Per modificare un elemento viene chiamata la stessa Activity usata per l'aggiunta di un evento, passando ad essa i dati da modificare tramite un bundle. L'Activity controllerà, nel metodo `onCreate()`, se è presente un bundle; in caso positivo, vengono prelevati i dati, che verranno, poi, inseriti all'interno degli elementi della form, permettendo all'utente di modificare l'evento.

```
public class AddOrModifyEventActivity extends AppCompatActivity implements DatePickerDialog.OnDateSetListener,
        TimePickerDialog.OnTimeSetListener {
    private EditText dateEditText;
    private EditText hourEditText;
    private EditText descriptionEditText;
    private AppDbHelper dbHelper;
    private SimpleDateFormat hourFormat;
    private SimpleDateFormat ymdFormat;
    private SimpleDateFormat dmyFormat;
    private boolean isInEditMode;
    private int id;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.event_modify_activity);
```

```
String oldDescription;

dbHelper = new AppDbHelper(this);
ArrayList<String> subjects = dbHelper.getSubjectNames();

Button changeDateButton = findViewById(R.id.modify_date_button);
Button changeHourButton = findViewById(R.id.modify_hour_button);
Button addEventButton = findViewById(R.id.add_event_button);
dateEditText = findViewById(R.id.event_date_field);
hourEditText = findViewById(R.id.event_hour_field);
descriptionEditText = findViewById(R.id.event_description_field);
final Spinner subjectSpinner = findViewById(R.id.event_subject_spinner);

ActionBar actionBar = getSupportActionBar();
if(actionBar != null)
    actionBar.setDisplayHomeAsUpEnabled(true);

ymdFormat = new SimpleDateFormat("yyyy/MM/dd");
dmyFormat = new SimpleDateFormat("dd/MM/yyyy");
hourFormat = new SimpleDateFormat("HH:mm");

ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
    R.layout.subject_spinner_item,
    R.id.subject_item,
    subjects
);
subjectSpinner.setAdapter(adapter);

Intent i = getIntent();
Bundle b = i.getBundleExtra("event_data");
if(b != null){
    if(actionBar != null) actionBar.setTitle("Modifica Evento");
    addEventButton.setText(R.string.edit_event);
    id = b.getInt("id");
    String oldDate = b.getString("date");
    String oldHour = b.getString("hour");
    String oldSubject = b.getString("subject");
    oldDescription = b.getString("description");

    try{
        dateEditText.setText(dmyFormat.format(ymdFormat.parse(oldDate)));
    } catch (Exception e){
        Log.e("ERROR:", e.getMessage());
    }
    hourEditText.setText(oldHour);
    subjectSpinner.setSelection(subjects.indexOf(oldSubject));
    descriptionEditText.setText(oldDescription);
    isInEditMode = true;
} else {
    if(actionBar != null) actionBar.setTitle("Aggiungi Evento");
    isInEditMode = false;
}

changeDateButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        DatePickerDialogFragment datePickerDialogFragment = new DatePickerDialogFragment();
        datePickerDialogFragment.show(getSupportFragmentManager(), "date_picker");
    }
});

changeHourButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        TimePickerDialogFragment timePickerDialogFragment = new TimePickerDialogFragment();
        timePickerDialogFragment.show(getSupportFragmentManager(), "time_picker");
    }
});

addEventButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String hour = hourEditText.getText().toString();
        String description = descriptionEditText.getText().toString();

        if(description.isEmpty()){
            showErrorDialog("È necessario inserire una descrizione dell'evento per proseguire!");
        } else if(!isValidFormat("dd/MM/yyyy", dateEditText.getText().toString()) ||
                !isValidFormat("HH:mm", hourEditText.getText().toString())){
            showErrorDialog("Il formato della data e/o dell'ora non è corretto. " +
                    "Per favore, inserisci i dati nel formato indicato!");
        } else try {
            String date = ymdFormat.format(dmyFormat.parse(dateEditText.getText().toString()));
            final CalendarEvent eventToAdd = new CalendarEvent(
                    0,
                    subjectSpinner.getSelectedItem().toString(),
                    date,
                    hour,
                    description,
                    0
            );
            boolean future = AlarmReceiver.compareDates(date, hour);

            if (!future) {
```

```
                String message = isInEditMode ?
                        "L'evento che stai aggiungendo riguarda un momento già trascorso. " +
                        "Sei sicuro di volerlo modificare ugualmente?" :
                        "L'evento che stai aggiungendo riguarda un momento già trascorso. " +
                        "Sei sicuro di volerlo aggiungere ugualmente?";
                new AlertDialog.Builder(AddOrModifyEventActivity.this)
                        .setTitle("Evento passato")
                        .setMessage(message)
                        .setPositiveButton("Si", new DialogInterface.OnClickListener() {
                            public void onClick(DialogInterface dialog, int which) {
                                newOrEdited(eventToAdd);
                                finish();
                            }
                        })
                        .setNegativeButton("No", null)
                        .show();
            } else {
                newOrEdited(eventToAdd);
                finish();
            }

        } catch(Exception e){
            showErrorDialog("Il formato della data e/o dell'ora non è corretto. " +
                    "Per favore, inserisci i dati nel formato indicato!");
            Log.e("ERROR:", e.getMessage());
        }

    }
});
}

private void newOrEdited (CalendarEvent eventToAdd){
    if(!isInEditMode){
        dbHelper.insertEvent(eventToAdd);
    } else {
        eventToAdd.setId(id);
        eventToAdd.setNotified(0);
        dbHelper.updateEvent(eventToAdd);
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    dbHelper.close();
}

@Override
public void onBackPressed() {
    finish();
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == android.R.id.home) {
        onBackPressed();
        return true;
    }
    Log.e("App", "È stata selezionata un'opzione che non dovrebbe esistere");
    return false;
}

@Override
public void onDateSet(DatePicker view, int year, int month, int dayOfMonth) {
    SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
    Calendar c = Calendar.getInstance();
    c.set(year, month, dayOfMonth);
    String date = format.format(c.getTime());
    dateEditText.setText(date);
}

@Override
public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
    Calendar c = Calendar.getInstance();
    c.set(Calendar.MINUTE, minute);
    c.set(Calendar.HOUR_OF_DAY, hourOfDay);
    String hour = hourFormat.format(c.getTime());
    hourEditText.setText(hour);
}

private void showErrorDialog(String message){
    ErrorDialog errorDialog = ErrorDialog.newInstance(message);
    errorDialog.show(getSupportFragmentManager(), "error_dialog");
}

private boolean isValidFormat(String format, String value) {
    Date date = null;
    try {
        SimpleDateFormat sdf = new SimpleDateFormat(format);
        date = sdf.parse(value);
        if (!value.equals(sdf.format(date))) {
            date = null;
        }
    } catch (ParseException ex) {
```

```
            ex.printStackTrace();
        }
        return date != null;
    }
}
```

Listato 4.12. Definizione della classe `AddOrModifyEventActivity`

4.2.2 Gestione dei mazzi

Nella Figura 4.3 viene illustrata la sezione dedicata alla gestione dei mazzi. Per realizzare tale sezione, come nel caso della gestione degli eventi, è stata usata una RecyclerView e, anche in questo caso, si è realizzato un adapter per permettere il raggruppamento degli elementi, in questo caso in base alla materia.

Figura 4.3. Screenshot della sezione riguardante la gestione dei mazzi

Tuttavia, al contrario della gestione degli eventi, su ciascuno degli elementi della lista (il cui layout è dato da una CardView), sono stati inseriti dei pulsanti, che permettono di avviare l'attività di studio su tale mazzo (nel caso in cui siano presenti delle carte) o la pagina di gestione del mazzo. Per il resto, l'implementazione del Fragment associato alla sezione è del tutto analoga a quella della gestione degli eventi, con la differenza che vengono prelevati, al posto degli eventi, tutti i mazzi, che verranno, in seguito, mostrati tramite la RecyclerView, come è possibile vedere nel Listato 4.13.

```
1       public void onStart() {
2           dbHelper = new AppDbHelper(getContext());
3           if(firstVisit){
4               decks = dbHelper.getDecks();
5               subjects = dbHelper.getSubjects();
6               adapter = new FlashcardsRVAdapter(getContext(), decks, subjects, dbHelper);
7               rv.setAdapter(adapter);
8               firstVisit = false;
9           } else {
10              subjects.clear();
11              subjects.addAll(dbHelper.getSubjects());
12              decks.clear();
13              decks.addAll(dbHelper.getDecks());
14              adapter.notifyDataSetChanged();
15          }
16          super.onStart();
17      }
```

Listato 4.13. Definizione del metodo `onStart()` della classe `FlashcardsFragment`

Per aggiungere o modificare un mazzo si è fatto uso di un semplice dialog, in quanto sono presenti poche informazioni da inserire. L'implementazione del dialog è mostrata nel Listato 4.14. La classe `AddDeckDialog`, che rappresenta il dialog usato, estende la classe `DialogFragment`, che ne gestisce il ciclo di vita. Per implementare le funzionalità necessarie per aggiungere e modificare elementi, comunicando con il Fragment, è stato necessario definire un'interfaccia che verrà, poi, implementata nel Fragment stesso. Tramite il metodo `onAttach()` è, quindi, stato possibile prelevare il listener implementato dal Fragment (come raffigurato nel Listato 4.15), per poi utilizzarlo all'interno del dialog, permettendo la comunicazione tra le due componenti.

```
public class AddDeckDialog extends DialogFragment {
    private EditText deckTitleField;
    private EditText deckDescriptionField;
    private Spinner subjectSpinner;
    private AppDbHelper dbHelper;
    private Deck deckToAdd;
    private AddDeckListener listener;

    public interface AddDeckListener{
        void onDeckAddedOrEdited();
    }

    public static AddDeckDialog newInstance(String subject, String name, String description){
        AddDeckDialog diag = new AddDeckDialog();
        Bundle args = new Bundle();
        args.putString("subject", subject);
        args.putString("deck_name", name);
        args.putString("description", description);
        diag.setArguments(args);
        return diag;
    }

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        Fragment fragment = Objects.requireNonNull(
            getActivity()).getSupportFragmentManager().findFragmentByTag("current_fragment"
        );
        try {
            if(fragment != null)
                listener = (AddDeckListener) fragment;
            else
                listener = (AddDeckListener) context;
        } catch (ClassCastException e) {
            throw new ClassCastException(getActivity().toString() +
                    " must implement the DeleteItemListener interface!");
        }
    }

    @NonNull
    @Override
    public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
        dbHelper = new AppDbHelper(getContext());
        ArrayList<String> subjectNames = dbHelper.getSubjectNames();

        AlertDialog.Builder builder = new AlertDialog.Builder(getContext());
        View dialogView = LayoutInflater.from(getContext()).inflate(R.layout.deck_modify_dialog, null);
        deckTitleField = dialogView.findViewById(R.id.deck_title_field);
```

```
        deckDescriptionField = dialogView.findViewById(R.id.deck_description_field);
        subjectSpinner = dialogView.findViewById(R.id.deck_subject_spinner);
        ArrayAdapter<String> adapter = new ArrayAdapter<>(
            Objects.requireNonNull(getContext()),
            R.layout.subject_spinner_item,
            R.id.subject_item,
            subjectNames
        );
        subjectSpinner.setAdapter(adapter);
        builder.setView(dialogView);

        if(getArguments() != null){
            builder.setTitle("Modifica Mazzo");
            final String oldSubject = getArguments().getString("subject");
            final String oldName = getArguments().getString("deck_name");
            final String oldDescription = getArguments().getString("description");
            deckTitleField.setText(oldName);
            deckDescriptionField.setText(oldDescription);
            subjectSpinner.setSelection(subjectNames.indexOf(oldSubject));
            builder.setPositiveButton("Modifica", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    if(deckDescriptionField.getText().toString().isEmpty() ||
                           deckTitleField.getText().toString().isEmpty()){
                        showErrorDialog();
                    } else {
                        deckToAdd = new Deck(
                                subjectSpinner.getSelectedItem().toString(),
                                deckDescriptionField.getText().toString(),
                                deckTitleField.getText().toString()
                        );
                        dbHelper.updateDeck(deckToAdd, oldSubject, oldName);
                        listener.onDeckAddedOrEdited();
                    }
                }
            });
        } else {
            builder.setTitle("Aggiungi Mazzo");
            builder.setPositiveButton("Aggiungi", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    if(deckDescriptionField.getText().toString().isEmpty() ||
                           deckTitleField.getText().toString().isEmpty()){
                        showErrorDialog();
                    } else {
                        deckToAdd = new Deck(
                                subjectSpinner.getSelectedItem().toString(),
                                deckDescriptionField.getText().toString(),
                                deckTitleField.getText().toString()
                        );
                        dbHelper.insertDeck(deckToAdd);
                        listener.onDeckAddedOrEdited();
                    }
                }
            });
        }
        builder.setNegativeButton("Annulla", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                dialog.dismiss();
            }
        });
        return builder.create();
    }

    private void showErrorDialog(){
        ErrorDialog errorDialog =
        ErrorDialog.newInstance("Per procedere è necessario avere entrambi i campi compilati!");
        errorDialog.show(Objects.requireNonNull(getFragmentManager()), "error_dialog");
    }
}
```

Listato 4.14. Definizione della classe `AddDeckDialog`

```
@Override
public void onDeckAddedOrEdited() {
    decks.clear();
    decks.addAll(dbHelper.getDecks());
    adapter.notifyDataSetChanged();
}
```

Listato 4.15. Listener implementato dalla classe `FlashcardsFragment` per ricaricare la lista dei mazzi

4.2.3 Gestione delle note

Nella Figura 4.4 è raffigurata la sezione dedicata alla gestione delle note. La sua implementazione è simile a quella delle altre due sezioni della pagina principale, in quanto, anche in questo caso, si fa uso di una RecyclerView, con un adapter che permette il raggruppamento degli elementi in base alla materia. L'unica cosa che cambia, effettivamente, è il layout del singolo elemento, essendo il tipo di dato da rappresentare fondamentalmente differente.

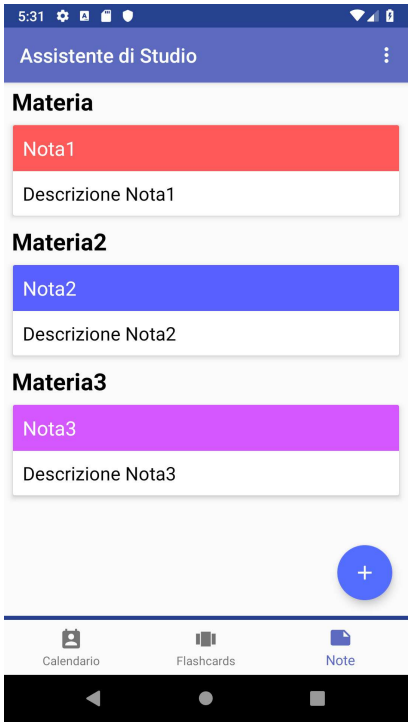

Figura 4.4. Screenshot della sezione riguardante la gestione delle note

Per quanto rigurda l'aggiunta e la modifica di una nota, analogamente all'aggiunta o alla modifica di un evento, si è fatto uso di un'unica Activity, che contiene i campi necessari per la compilazione (materia, titolo della nota, contenuto della stessa). Se vengono passati dei dati tramite un bundle, l'Activity ha la funzione di modificare una nota, inserendo i dati da modificare nella form dopo averli prelevati, mentre, in caso contrario, viene aggiunta la nota. Dopo aver cliccato sul pulsante dedicato all'aggiunta/modifica di una nota, viene prima controllato se tutti i campi sono compilati; successivamente, si effettua l'operazione richiesta sul database, ritornando alla schermata principale.

4.3 Gestione e studio delle flashcard

4.3.1 Gestione delle flashcard

Per quanto riguarda la sezione di gestione delle carte di un mazzo, visibile nella Figura 4.5, è stata usata un'altra Activity, denominata `DeckManagementActivity`, mostrata nel Listato 4.16.

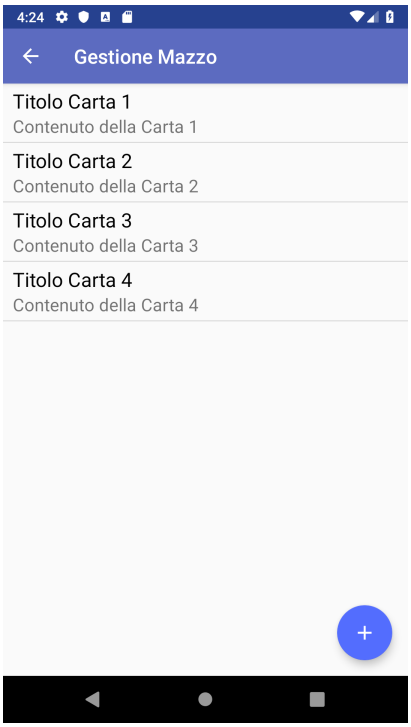

Figura 4.5. Screenshot della pagina relativa alla gestione delle carte di un mazzo

In questa Activity vengono passati, tramite un bundle, i dati relativi al mazzo. Dopo aver prelevato dal database tutte le carte relative al mazzo, queste verranno mostrate sulla schermata mediante l'uso di una semplice ListView, in quanto, in questo caso, non è richiesto l'uso di un layout fortemente personalizzato. Per la lista è stato utilizzato un adapter personalizzato, simile a quelli visti precedentemente, e l'interfaccia assegnata ad ogni suo elemento è composta soltanto dal titolo e dalla descrizione della carta.

L'aggiunta e la modifica della carta avvengono tramite un dialog, analogo a quello utilizzato per aggiungere o modificare un mazzo; nei metodi `onCardAdded()` e `onCardEdited()` dell'Activity sono implementati i listener associati al dialog.

```
public class DeckManagementActivity extends AppCompatActivity implements DeleteDialog.DeleteItemListener,
        AddCardDialog.AddCardListener {
    private ArrayList<Card> cards;
    private CardAdapter adapter;
```

```
private AppDbHelper dbHelper;
private String deckSubject;
private String deckName;

@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_deck_management);

    ActionBar actionBar = getSupportActionBar();
    Objects.requireNonNull(actionBar).setTitle("Gestione Mazzo");
    actionBar.setDisplayHomeAsUpEnabled(true);

    Intent i = getIntent();
    Bundle bundle = i.getBundleExtra("deck_data");
    deckSubject = bundle.getString("subject");
    deckName = bundle.getString("deck_name");

    dbHelper = new AppDbHelper(this);

    cards = dbHelper.getCards(deckSubject, deckName);

    ListView cardsListView = findViewById(R.id.cards_list_view);
    adapter = new CardAdapter(this, R.layout.card_item, R.id.card_title, cards);
    cardsListView.setAdapter(adapter);
    cardsListView.setLongClickable(true);
    registerForContextMenu(cardsListView);

    FloatingActionButton fab = findViewById(R.id.add_button);
    fab.setOnClickListener(new View.OnClickListener(){
        @Override
        public void onClick(View v) {
            AddCardDialog addCardDialog = new AddCardDialog();
            addCardDialog.show(getSupportFragmentManager(), "add_card");
        }
    });
}

@Override
protected void onDestroy() {
    dbHelper.close();
    super.onDestroy();
}

@Override
public void onBackPressed() {
    finish();
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == android.R.id.home) {
        onBackPressed();
        return true;
    }
    Log.e("App", "È stata selezionata un opzione che non dovrebbe esistere");
    return false;
}

@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    menu.setHeaderTitle("Opzioni Carta");
    menu.add("Modifica");
    menu.add("Elimina");
}

@Override
public boolean onContextItemSelected(MenuItem item) {
    final AdapterView.AdapterContextMenuInfo info = (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
    if(item.getTitle().equals("Modifica")){
        AddCardDialog addCardDialog = AddCardDialog.newInstance(
                cards.get(info.position).getTitle(),
                cards.get(info.position).getContent(),
                info.position
        );
        addCardDialog.show(getSupportFragmentManager(), "edit_card");
    } else if (item.getTitle() == "Elimina") {
        DeleteDialog deleteDialog = DeleteDialog.newInstance(info.position,
                "Elimina Materia",
                "Vuoi davvero eliminare la seguente materia?");
        deleteDialog.show(getSupportFragmentManager(), "delete");
    } else {
        Toast.makeText(this, "Default", Toast.LENGTH_SHORT).show();
    }
    return true;
}

@Override
public void onDeleteItem(int position) {
    dbHelper.deleteCard(deckSubject, deckName, cards.get(position).getTitle());
    cards.remove(position);
    adapter.notifyDataSetChanged();
}
```

```
        @Override
        public void onCardAdded(String title, String content) {
            Card toAdd = new Card(deckSubject, deckName, title, content, 0);
            dbHelper.insertCard(toAdd);
            cards.add(toAdd);
            adapter.notifyDataSetChanged();
        }

        @Override
        public void onCardEdited(String title, String content, int position) {
            String oldTitle = cards.get(position).getTitle();
            cards.get(position).setTitle(title);
            cards.get(position).setContent(content);
            dbHelper.updateCard(cards.get(position), oldTitle);
            adapter.notifyDataSetChanged();
        }
    }
```

Listato 4.16. Definizione della classe `DeckManagementActivity`

4.3.2 Studio delle flashcard

Nella Figura 4.6 viene mostrata la pagina dedicata allo studio tramite flashcard, mentre, nel Listato 4.17, viene illustrata l'implementazione dell'Activity associata a tale schermata.

Figura 4.6. Screenshot della schermata di studio tramite le flashcard

Per poter implementare il sistema di studio è necessario, innanzitutto, prelevare tutte le carte del mazzo di interesse, in ordine decrescente rispetto alla loro priorità;

successivamente, le carte dovranno essere mostrate all'utente una alla volta, facendo vedere prima una faccia della carta, corrispondente al titolo del concetto, e poi l'altra, corrispondente al suo contenuto. Successivamente, in base a come lo studente si è trovato con il concetto espresso dalla carta, verrà deciso se aumentare o diminuire la priorità di quest'ultima, e si passerà alla prossima carta, continuando con la procedura fino a quando non terminano le carte del mazzo o l'utente decide di tornare indietro alla pagina principale.

La schermata è fondamentalmente composta da una CardView, in cui è inserito il contenuto di interesse, e da tre pulsanti, che permettono rispettivamente di:

- mostrare il contenuto dalla carta, se al momento è mostrato il suo titolo;
- aumentare la priorità della carta, se il concetto è stato ritenuto difficile da ricordare;
- diminuire la priorità della carta, se il concetto è stato ritenuto facile da ricordare.

Se sulla CardView è presente il titolo della carta viene visualizzato solo il pulsante per mostrarne il contenuto, mentre, in caso contrario, vengono mostrati gli altri due pulsanti.

Per quanto riguarda l'implementazione dell'Activity, nel metodo `onCreate()` vengono inizializzate le componenti principali della schermata, come i pulsanti, la lista delle carte (che viene caricata dal database oppure, se l'Activity è stata ricreata in seguito ad un cambiamento dell'orientamento, viene passata dall'Activity appena distrutta, in modo da avere i dati sempre aggiornati), e la variabile `isShowingContent`, che indica se in quel momento viene mostrato il contenuto della carta o meno. L'inizializzazione di questi elementi è possibile grazie ai dati, riguardanti il mazzo considerato per lo studio, che vengono passati all'Activity tramite un bundle.

Una volta che tutti i dati d'interesse sono stati inizializzati, la CardView viene aggiornata con il titolo della prima carta (oppure, se l'Activity è stata ricreata, riparte dall'ultimo stato in cui si trovava). Il metodo `onClick()` della classe implementa i click listener dei pulsanti, utilizzati per aggiornare lo stato dell'Activity. In particolare:

- Se il pulsante cliccato è quello usato per mostrare il contenuto, viene aggiornata la CardView con il contenuto della carta, e viene chiamato il metodo `setFeedbackVisible()`, che nasconde tale pulsante, mostrando, invece, quelli usati per il feedback sulla carta.
- Se il pulsante cliccato è uno di quelli di feedback, usati per modificare la priorità della carta, viene aggiornata la carta con la nuova priorità e viene chiamato il metodo `setNextCard()`, che aggiorna la schermata con la carta successiva, mostrando, di nuovo, il pulsante per evidenziare il contenuto, e nascondendo i pulsanti di feedback. Se la carta successiva non è presente viene, semplicemente, terminata l'Activity, riportando l'utente alla pagina principale.

```
public class FlashcardsStudyActivity extends AppCompatActivity implements View.OnClickListener {
    private AppDbHelper dbHelper;
    private boolean isShowingContent;
    private ArrayList<Card> cards;
    private TextView cardTextView;
    private Button easyButton;
    private Button hardButton;
```

```
private Button switchButton;
private int currentCardPosition;
private TextView feedbackLabel;

@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.flashcards_study_layout);

    ActionBar actionBar = getSupportActionBar();
    Objects.requireNonNull(actionBar).setTitle("Studio Mazzo");
    actionBar.setDisplayHomeAsUpEnabled(true);

    feedbackLabel = findViewById(R.id.feedback_label);
    easyButton = findViewById(R.id.easy_btn);
    hardButton = findViewById(R.id.hard_btn);
    switchButton = findViewById(R.id.switch_btn);
    cardTextView = findViewById(R.id.flashcard_content_textview);
    currentCardPosition = 0;
    isShowingContent = false;

    easyButton.setOnClickListener(this);
    hardButton.setOnClickListener(this);
    switchButton.setOnClickListener(this);

    Intent i = getIntent();
    Bundle bundle = i.getBundleExtra("deck_data");
    String deckSubject = bundle.getString("subject");
    String deckName = bundle.getString("deck_name");

    dbHelper = new AppDbHelper(this);

    if(savedInstanceState != null){
        isShowingContent = savedInstanceState.getBoolean("is_showing_content");
        currentCardPosition = savedInstanceState.getInt("current_position");
        cards = savedInstanceState.getParcelableArrayList("cards");
        if(isShowingContent){
            cardTextView.setText(Objects.requireNonNull(cards).get(currentCardPosition).getContent());
            cardTextView.setLines(30);
            cardTextView.setEllipsize(TextUtils.TruncateAt.END);
            setFeedbackVisible();
        } else {
            cardTextView.setText(Objects.requireNonNull(cards).get(currentCardPosition).getTitle());
        }
    } else {
        cards = dbHelper.getCards(deckSubject, deckName);
        try {
            cardTextView.setText(cards.get(currentCardPosition).getTitle());
        } catch(Exception e) {
            onBackPressed();
        }
    }
}

@Override
public void onBackPressed() {
    finish();
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == android.R.id.home) {
        onBackPressed();
        return true;
    }
    Log.e("App", "È stata selezionata un opzione che non dovrebbe esistere");
    return false;
}

@Override
public void onClick(View v) {
    switch(v.getId()){
        case R.id.switch_btn:
            if(!isShowingContent){
                cardTextView.setText(cards.get(currentCardPosition).getContent());
                cardTextView.setLines(30);
                cardTextView.setEllipsize(TextUtils.TruncateAt.END);
                setFeedbackVisible();
                isShowingContent = true;
            }
            break;
        case R.id.easy_btn:
            if(isShowingContent){
                Card currentCard = cards.get(currentCardPosition);
                int currentPriority = currentCard.getPriority();
                if(currentPriority > 0){
                    currentCard.setPriority(currentPriority - 1);
                    dbHelper.updateCard(currentCard, currentCard.getTitle());
                }
                setNextCard();
                isShowingContent = false;
            }
            break;
        case R.id.hard_btn:
```

```
                if(isShowingContent){
                    Card currentCard = cards.get(currentCardPosition);
                    int currentPriority = currentCard.getPriority();
                    currentCard.setPriority(currentPriority + 1);
                    dbHelper.updateCard(currentCard, currentCard.getTitle());
                    setNextCard();
                    isShowingContent = false;
                }
                break;
            default:
                Toast.makeText(this, "default", Toast.LENGTH_SHORT).show();
                break;
        }
    }

    @Override
    protected void onDestroy() {
        dbHelper.close();
        super.onDestroy();
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("current_position", currentCardPosition);
        outState.putBoolean("is_showing_content", isShowingContent);
        outState.putParcelableArrayList("cards", cards);
    }

    private void setNextCard(){
        currentCardPosition++;
        if(currentCardPosition == cards.size()){
            finish();
        } else {
            cardTextView.setText(cards.get(currentCardPosition).getTitle());
            cardTextView.setLines(30);
            cardTextView.setEllipsize(TextUtils.TruncateAt.END);
            setFeedbackInvisible();
        }
    }

    private void setFeedbackInvisible(){
        easyButton.setVisibility(View.INVISIBLE);
        hardButton.setVisibility(View.INVISIBLE);
        feedbackLabel.setVisibility(View.INVISIBLE);
        switchButton.setVisibility(View.VISIBLE);
    }

    private void setFeedbackVisible(){
        easyButton.setVisibility(View.VISIBLE);
        hardButton.setVisibility(View.VISIBLE);
        feedbackLabel.setVisibility(View.VISIBLE);
        switchButton.setVisibility(View.INVISIBLE);
    }
}
```

Listato 4.17. Definizione della classe `FlashcardsStudyActivity`

4.4 Gestione delle materie

Per quanto riguarda la pagina relativa alla gestione delle materie, mostrata nella Figura 4.7, la sua implementazione è analoga a quella della schermata di gestione delle carte di un mazzo, con la differenza che nella ListView ogni elemento è composto sia dal nome della materia che da un FrameLayout, che ha il colore della materia, utilizzato per mostrare all'utente il colore usato per catalogare gli elementi associati alla materia.

Per quanto riguarda l'aggiunta o la modifica di una materia, invece, si è fatto uso di un dialog, presente nella Figura 4.8, che contiene gli EditText usati per l'inserimento del nome e del colore della materia (in formato esadecimale), assieme a un FrameLayout, usato per rappresentare visivamente il colore e per aprire, al suo click, un ColorPicker, che permette all'utente di selezionare il colore in modo intuitivo.

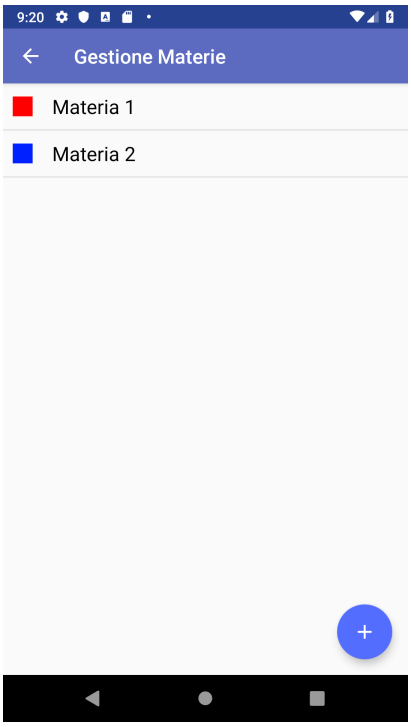

Figura 4.7. Screenshot della pagina relativa alla gestione delle materie

Nel Listato 4.18 viene mostrato il codice relativo all'implementazione di tale dialog. Per fare in modo che il colore venga aggiornato correttamente, senza problemi di parsing, si è fatto uso della funzione `colorParser()`. Quest'ultima controlla, tramite un'espressione regolare, se il formato del colore è esadecimale (eventualmente, anche, escludendo il cancelletto, che verrà aggiunto dalla funzione stessa) e, in caso positivo, lo aggiorna, mentre, in caso negativo, non si procede con l'aggiornamento. Tale funzione viene chiamata in due occasioni:

- Quando si modifica l'EditText relativo al colore. Nel metodo `onCreateDialog()`, infatti, è stato aggiunto un TextChangedListener alla casella di testo, chiamando, ad ogni modifica, la funzione per validare il nuovo colore, ed aggiornando, in caso di successo, il colore raffigurato nel FrameLayout.
- Quando si intende modificare o aggiungere la materia, come illustrato nel metodo `checkConfirm()`, in cui, dopo aver controllato se entrambi i campi sono compilati, viene chiamato il metodo `colorParser()` con il parametro `isConfirmed` impostato a `true`, così che, se la validazione non va a buon fine, viene mostrato un messaggio di errore.

Per quanto riguarda il ColorPicker si è deciso di sfruttare una libreria esterna, presente su GitHub, che rende immediata l'apertura di un dialog contenente un ColorPicker, che permette all'utente di scegliere il nuovo colore. L'implementazione del ColorPicker viene effettuata tramite il metodo `onCreateDialog()`, all'interno

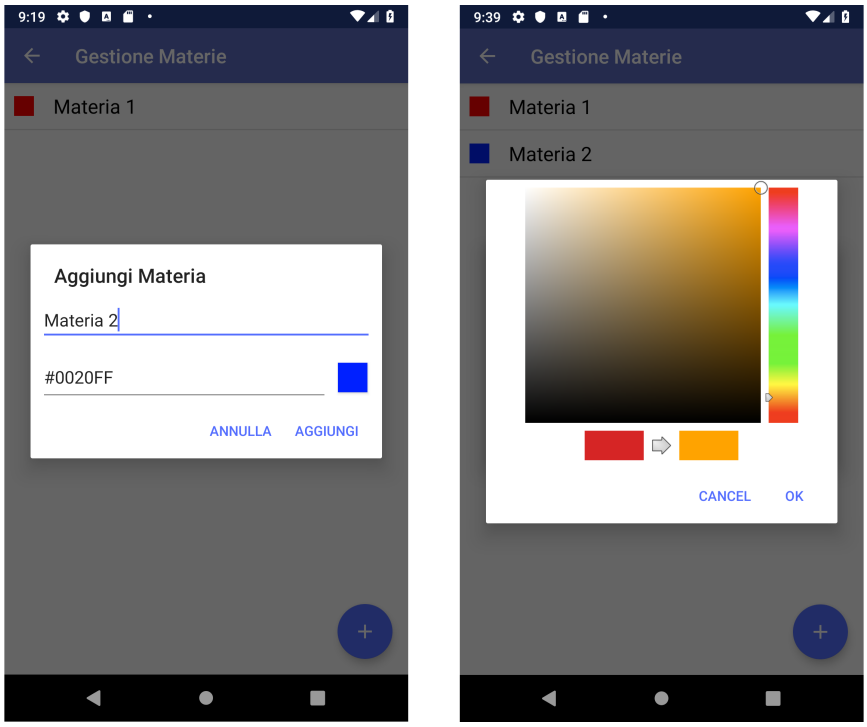

Figura 4.8. Dialog usato per aggiungere o modificare una materia

del click listener della FrameView. In essa viene chiamata la libreria, e, in caso di conferma, si aggiorna il colore da inserire nella materia.

```
public class AddSubjectDialog extends DialogFragment {

    private int parsedColor;
    private String newColor;
    private AddSubjectListener listener;

    public interface AddSubjectListener{
        void onSubjectAdded(String name, String color);
        void onSubjectEdited(String name, String color, int position);
    }

    public static AddSubjectDialog newInstance(String name, String color, int position) {
        Bundle args = new Bundle();
        args.putString("name", name);
        args.putString("color", color);
        args.putInt("position", position);
        AddSubjectDialog fragment = new AddSubjectDialog();
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        try {
            listener = (AddSubjectListener) context;
        } catch (ClassCastException e) {
            throw new ClassCastException(getActivity().toString() +
                    " must implement the DeleteItemListener interface!");
        }
    }

    private boolean colorParser(String color, boolean isConfirmed) {
        try {
            boolean b2 = Pattern.matches("\\p{XDigit}{6}", color);
            if (b2) {
                color = "#" + color;
```

```
            }
            boolean b1 = Pattern.matches("#\\p{XDigit}{6}", color);
            if (b1) {
                newColor = color;
                parsedColor = Color.parseColor(newColor);
                return true;
            } else {
                parsedColor = Color.parseColor("#FFFFFF");
                if (isConfirmed) {
                    showErrorDialog(true);
                }
                return false;
            }

        } catch (Exception e){
            parsedColor = Color.parseColor("#FFFFFF");
            if (isConfirmed) {
                showErrorDialog(true);
            }
            return false;
        }
    }

    private void showErrorDialog(boolean error){
        String message;
        if (error){
            message = "Il colore non è nel formato corretto. Assicurati di aver usato il formato esadecimale!";
        } else {
            message = "Per aggiungere la materia è necessario che entrambi i campi siano compilati!";
        }
        ErrorDialog errorDialog = ErrorDialog.newInstance(message);
        errorDialog.show(Objects.requireNonNull(getFragmentManager()), "error_dialog");
    }

    private void checkConfirm(boolean isntNew, int position,
           EditText subjectNameEditText, EditText subjectColorEditText){
        String newName = subjectNameEditText.getText().toString();
        newColor = subjectColorEditText.getText().toString();
        boolean isEmpty = !newName.isEmpty() && !newColor.isEmpty();

        if(isEmpty){
            boolean validColor = colorParser(newColor, true);
            if(validColor){
                if (isntNew)
                    listener.onSubjectEdited(newName, newColor, position);
                else
                    listener.onSubjectAdded(newName, newColor);
            }
        } else {
            showErrorDialog(isEmpty);
        }
    }

    @NonNull
    @Override
    public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(Objects.requireNonNull(getContext()));
        View dialogView = LayoutInflater.from(getContext()).inflate(R.layout.subject_modify_dialog, null);
        builder.setView(dialogView);
        final EditText subjectNameEditText = dialogView.findViewById(R.id.subject_name_field);
        final EditText subjectColorEditText = dialogView.findViewById(R.id.subject_color_field);
        final Button subjectColorFrameView = dialogView.findViewById(R.id.subject_color);

        subjectColorEditText.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start, int count, int after){}

            @Override
            public void onTextChanged(CharSequence s, int start, int before, int count) {
                if (s.length() > 5) {
                    try {
                        colorParser(s.toString(), false);
                        subjectColorFrameView.setBackgroundColor(parsedColor);
                    } catch (Exception e) {
                        subjectColorFrameView.setBackgroundColor(
                            ContextCompat.getColor(Objects.requireNonNull(getContext()), R.color.white)
                        );
                    }
                }
            }

            @Override
            public void afterTextChanged(Editable s){}
        });

        subjectColorFrameView.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                openColorPicker();
            }

            private void openColorPicker() {

                String s = subjectColorEditText.getText().toString();
```

```
                try {
                    colorParser(s, false);
                } finally {

                    AmbilWarnaDialog colorPicker = new AmbilWarnaDialog(getContext(), parsedColor,
                    new AmbilWarnaDialog.OnAmbilWarnaListener() {
                        @Override
                        public void onCancel(AmbilWarnaDialog dialog) {
                            dismiss();
                        }

                        @Override
                        public void onOk(AmbilWarnaDialog dialog, int color) {
                            parsedColor = color;
                            subjectColorEditText.setText(String.format("#%06X", (0x00FFFFFF & parsedColor)));
                            subjectColorFrameView.setBackgroundColor(parsedColor);
                        }
                    });
                    colorPicker.show();
                }
            }
        });

        if(getArguments() != null) {
            final int position = getArguments().getInt("position");
            String color = getArguments().getString("color");
            subjectNameEditText.setText(getArguments().getString("name"));
            subjectColorEditText.setText(color);
            subjectColorFrameView.setBackgroundColor(Color.parseColor(color));
            builder.setTitle("Modifica Materia");
            builder.setPositiveButton("Modifica", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    checkConfirm(true, position, subjectNameEditText, subjectColorEditText);
                }
            });
        } else {
            builder.setTitle("Aggiungi Materia");
            builder.setPositiveButton("Aggiungi", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    checkConfirm(false,0, subjectNameEditText, subjectColorEditText);
                }
            });
        }
        builder.setNegativeButton("Annulla", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                dialog.dismiss();
            }
        });
        return builder.create();

    }
}
```

Listato 4.18. Definizione della classe `AddSubjectDialog`

4.5 Impostazioni e aiuto

Nella Figura 4.9 vengono mostrate la pagina delle impostazioni e quella di aiuto dell'applicazione.

Per implementare la pagina delle impostazioni si è fatto uso di un *PreferenceFragment*, ovvero un Fragment dedicato appositamente all'implementazione di schermate di configurazione. Nei Listati 4.19 - 4.21 vengono mostrati, nello specifico, le implementazioni di tutte le componenti della pagina. In particolare, nell'Activity associata alla pagina, nel metodo `onCreate()`, viene esplicitamente dichiarato il Fragment come suo contenuto, mentre, nel codice associato al Fragment, è stato semplicemente caricato il file XML relativo alle preferenze, contente degli interruttori per configurare le scelte relative alla visualizzazione del calendario e all'abilitazione delle notifiche.

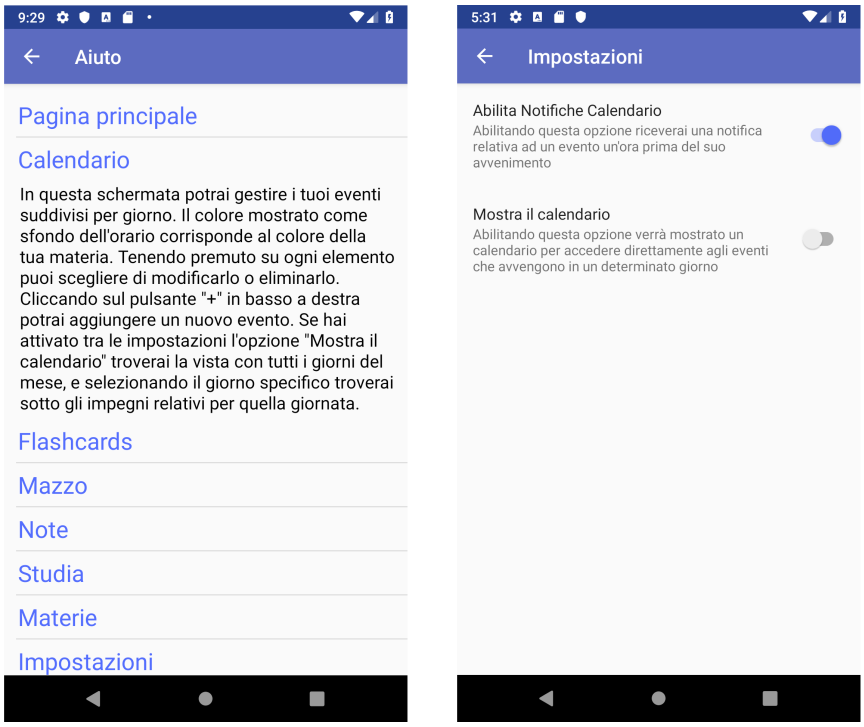

Figura 4.9. Screenshot delle pagine delle impostazioni e di aiuto

```
public class SettingsActivity extends AppCompatActivity {

    public static final String NOTIFICATION_KEY = "set_notification";
    public static final String CALENDAR_KEY = "set_calendar";

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportFragmentManager().beginTransaction()
                .replace(android.R.id.content, new SettingsFragment())
                .commit();
    }
}
```

Listato 4.19. Definizione della classe `SettingsActivity`

```
public class SettingsFragment extends PreferenceFragmentCompat {

    @Override
    public void onCreatePreferences(Bundle bundle, String s) {
        setPreferencesFromResource(R.xml.pref_main, s);
    }

}
```

Listato 4.20. Definizione della classe `SettingsFragment`

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.preference.PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <android.support.v7.preference.SwitchPreferenceCompat
```

```
        android:title="Abilita Notifiche Calendario"
        android:defaultValue="false"
        app:iconSpaceReserved="false"
        android:key="set_notification"
        android:summary="Abilitando questa opzione riceverai una notifica
        relativa ad un evento un'ora prima del suo avvenimento"/>

    <android.support.v7.preference.SwitchPreferenceCompat
        android:title="Mostra il calendario"
        android:defaultValue="false"
        app:iconSpaceReserved="false"
        android:key="set_calendar"
        android:summary="Abilitando questa opzione verrà mostrato un calendario per accedere
        direttamente agli eventi che avvengono in un determinato giorno"/>

</android.support.v7.preference.PreferenceScreen>
```

Listato 4.21. Contenuto del file XML relativo alle preferenze nelle impostazioni

Per quanto riguarda la pagina di aiuto, invece, la sua implementazione è mostrata nei Listati 4.22 - 4.23. Nell'Activity associata alla schermata è presente soltanto una *ExpandableListView*, ovvero una ListView particolare che permette ad ogni suo elemento di essere espanso con degli elementi aggiuntivi. A tale ListView è stato associato un adapter personalizzato, che estende la classe `BaseExpandableListAdapter` e contiene sia i titoli delle sezioni che le loro descrizioni, definendo, anche, le TextView usate per rappresentare tali informazioni.

```
public class HelpActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_help);

        ActionBar actionBar = getSupportActionBar();
        Objects.requireNonNull(actionBar).setTitle("Aiuto");
        actionBar.setDisplayHomeAsUpEnabled(true);

        ExpandableListView expandableListView = findViewById(R.id.faq_list_view);
        HelpAdapter adapter = new HelpAdapter(this);
        expandableListView.setAdapter(adapter);
    }

    @Override
    public void onBackPressed() {
        finish();
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId() == android.R.id.home) {
            onBackPressed();
            return true;
        }
        Log.e("App", "È stata selezionata un opzione che non dovrebbe esistere");
        return false;
    }
}
```

Listato 4.22. Definizione della classe `HelpActivity`

```
public class HelpAdapter extends BaseExpandableListAdapter {

    private final Context context;

    private final String [] groupNames;
    private final String [][] childNames;

    public HelpAdapter(Context context) {

        this.context = context;
        groupNames = new String[]{context.getResources().getString(R.string.h_title_1),
                context.getResources().getString(R.string.fragments_event_title),
                context.getResources().getString(R.string.fragments_cards_title),
                context.getResources().getString(R.string.h_title_4),
                context.getResources().getString(R.string.fragments_notes_title),
                context.getResources().getString(R.string.h_title_6),
                context.getResources().getString(R.string.h_title_7),
```

```
                context.getResources().getString(R.string.h_title_8),
                context.getResources().getString(R.string.h_title_9)
        };
        childNames = new String[][]{
                {context.getResources().getString(R.string.h_child_1)},
                {context.getResources().getString(R.string.h_child_2)},
                {context.getResources().getString(R.string.h_child_3)},
                {context.getResources().getString(R.string.h_child_4)},
                {context.getResources().getString(R.string.h_child_5)},
                {context.getResources().getString(R.string.h_child_6)},
                {context.getResources().getString(R.string.h_child_7)},
                {context.getResources().getString(R.string.h_child_8)},
                {context.getResources().getString(R.string.h_child_9)}
        };
    }

    @Override
    public int getGroupCount() {
        return groupNames.length;
    }

    @Override
    public int getChildrenCount(int groupPosition) {
        return childNames[groupPosition].length;
    }

    @Override
    public Object getGroup(int groupPosition) {
        return groupNames[groupPosition];
    }

    @Override
    public Object getChild(int groupPosition, int childPosition) {
        return childNames[groupPosition][childPosition];
    }

    @Override
    public long getGroupId(int groupPosition) {
        return groupPosition;
    }

    @Override
    public long getChildId(int groupPosition, int childPosition) {
        return childPosition;
    }

    @Override
    public boolean hasStableIds() {
        return false;
    }

    @Override
    public View getGroupView(int groupPosition, boolean isExpanded, View convertView, ViewGroup parent) {

        TextView textView = new TextView(context);
        textView.setText((groupNames[groupPosition]));
        textView.setPadding(5, 15, 20, 15);
        textView.setTextColor(ContextCompat.getColor(context, R.color.colorAccent));
        textView.setTextSize(24);

        return textView;
    }

    @Override
    public View getChildView(int groupPosition, int childPosition, boolean isLastChild,
            View convertView, ViewGroup parent) {
        TextView textView = new TextView(context);
        textView.setText((childNames[groupPosition][childPosition]));
        textView.setPadding(10, 0, 25, 10);
        textView.setTextColor(ContextCompat.getColor(context, R.color.black));
        textView.setTextSize(18);

        return textView;
    }

    @Override
    public boolean isChildSelectable(int groupPosition, int childPosition) {
        return false;
    }
}
```

Listato 4.23. Definizione della classe `HelpAdapter`

5

Conclusioni

Nella presente tesi è stata progettata ed implementata un'applicazione che ha l'obiettivo di supportare l'organizzazione delle attività di uno studente. Tale applicazione è composta da tre componenti principali: un calendario, uno strumento per prendere delle note, e uno strumento per ottimizzare l'attività di studio sfruttando il concetto di flashcard, sempre più utilizzato come strumento di apprendimento.

Si è partiti, innanzitutto, con una breve introduzione sul sistema operativo Android, e sugli strumenti che vengono normalmente utilizzati per realizzare delle applicazioni native su tale piattaforma.

Si è, quindi, cominciato a discutere sull'applicazione oggetto della presente tesi, a partire dall'analisi dei requisiti relativa ad essa. Durante questa fase, in particolare, sono stati ottenuti i requisiti da soddisfare nella progettazione dell'app, ed è stato, anche, introdotto un diagramma dei casi d'uso per evidenziare le funzionalità che l'applicazione deve presentare, e come questa deve interagire con l'utente.

Dopo aver ricavato i requisiti dell'applicazione si è successivamente entrati nel dettaglio sulla sua progettazione, delineando la struttura da adottare, assieme alla corrispettiva interfaccia grafica (introducendo, anche, dei wireframe che mostrano le sue caratteristiche principali), alle funzionalità e alla rappresentazione dei dati di interesse in un modello di tipo relazionale.

Grazie al lavoro svolto durante la fase di progettazione è stato, quindi, possibile passare all'implementazione dell'app, realizzando tutte le funzionalità precedentemente delineate. Nella tesi, in particolare, sono state descritte le componenti utilizzate per realizzare le funzionalità principali dell'applicazione, introducendo anche dei listati per illustrare la loro implementazione.

L'applicazione realizzata, seppur risulti completa rispetto alle funzionalità inizialmente progettate, può essere ulteriormente affinata al fine di renderla più flessibile e di migliorarne l'esperienza utente.

Un primo miglioramento che è possibile effettuare per l'applicazione consiste nell'utilizzare, oltre ad un database locale, anche un servizio web di backend dedicato alla conservazione dei dati dell'utente, implementando eventualmente un sistema di autenticazione. Seppur la soluzione attuale sia, infatti, adeguata per conservare i dati, questi andranno irrimediabilmente perduti una volta che l'app viene disinstallata, per cui potrebbe essere necessario anche un sistema per conservare tutte le informazioni di interesse una volta che queste non sono più presenti nel dispositivo.

Un altro possibile miglioramento per l'applicazione potrebbe essere quello di usare un algoritmo più complesso per il calcolo della priorità delle carte. Il sistema corrente infatti, seppur adeguato, potrebbe non selezionare in modo corretto i concetti con cui lo studente ha maggiori difficoltà. Si potrebbe, inoltre, anche effettuare una modifica nell'applicazione e nel suo database per implementare forme multimediali differenti in una carta, usando, oltre al testo, anche delle immagini.

Un ultimo possibile miglioramento per l'applicazione potrebbe, infine, essere quello di implementare un sistema per formattare le informazioni all'interno delle note, dei mazzi, delle carte e degli eventi. Così facendo, l'utente potrebbe mettere in rilievo le informazioni più importanti, per facilitarne la lettura.

Riferimenti bibliografici

1. Qual è la differenza tra Wireframe, Prototipo e Mockup? `http://www.fabiomarasco.it/la-differenza-tra-wireframe-prototipo-e-mockup/`, 2015.
2. Bottom Navigation Android Example using Fragments. `https://www.simplifiedcoding.net/bottom-navigation-android-example/`, 2018.
3. Android. `https://it.wikipedia.org/wiki/Android`, 2019.
4. App mobile ibride, native o web: le differenze. `https://www.html.it/faq/app-mobile-ibride-native-o-web-le-differenze/`, 2019.
5. Create a Card-Based Layout. `https://developer.android.com/guide/topics/ui/layout/cardview/`, 2019.
6. Create a Card-Based Layout. `https://developer.android.com/guide/topics/ui/layout/cardview/`, 2019.
7. Create a List with RecyclerView. `https://developer.android.com/guide/topics/ui/layout/recyclerview/`, 2019.
8. Flashcard. `https://it.wikipedia.org/wiki/Flashcard`, 2019.
9. I requisiti. `http://www.rpolillo.it/faciledausare/Cap.7.htm`, 2019.
10. Progettazione di basi di dati relazionali. `https://users.dimi.uniud.it/~massimo.franceschet/teatro-sql/progettazione.html/`, 2019.
11. Settings with PreferenceFragment. `https://guides.codepath.com/android/settings-with-preferencefragment/`, 2019.
12. Unified Modeling Language (UML) — Activity Diagrams. `https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/`, 2019.
13. Use Case Diagram. `https://it.wikipedia.org/wiki/Use_Case_Diagram`, 2019.
14. R. Boyer. *Android 9 Application Development Cookbook*. Packt, 2018.
15. P. Camagni, R. Nikolassy, and E. Falzone. *Sviluppare App per Android*. Hoepli, 2017.
16. M. Carli. *Android 9. Guida completa per lo sviluppo di applicazione mobile*. Apogeo, 2019.
17. E. Cisotti and M. Giannino. *Android: Guida completa*. Edizioni LSWR, 2015.
18. F. Collini, M. Bonifazi, A. Martellucci, and S. Sanna. *Android. Programmazione avanzata*. Digital Lifestyle Pro, 2015.
19. I. Darwin. *Android Cookbook: Problems and Solutions for Android Developers*. Oreilly, 2017.
20. M. Gargenta and M. Nakamura. *Sviluppare con Android. Realizzare applicazioni mobili con Java ed Eclipse*. Hoepli, 2014.
21. G. Grandinetti. *Android studio. Sviluppare vere applicazione Android partendo da zero*. Edizionifutura.Com, 2014.
22. T. Hagos. *Learn Android Studio 3: Efficient Android App Development*. Apress, 2015.

23. C. Haseman. *Creare App per Android. Progettazione e sviluppo.* Mondadori Informatica, 2012.
24. C. Larman. *Applicare UML e i pattern. Analisi e progettazione orientata agli oggetti.* Pearson, 2016.
25. B. Phillips, C. Stewart, and K. Marsicano. *Android Programming: The Big Nerd Ranch Guide.* Big Nerd Ranch Guides, 2017.
26. H. Schildt. *Java: The Complete Reference, Eleventh Edition.* McGraw-Hill Education, 2018.
27. D. Sillars. *Sviluppare applicazioni Android ad alte prestazioni.* Tecniche Nuove, 2017.
28. N. Smyth. *Android Studio 3.4 Development Essentials - Java Edition: Developing Android 9 Apps Using Android Studio 3.4, Java and Android Jetpack.* Payload Media, 2019.