ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Un'applicazione Android per la gestione ed il monitoraggio dei flussi monetari personali

Relatore:
Chiar.mo Prof.
Luciano Bononi

Correlatore:
Dott.
Luca Bedogni

Presentata da:
Lorenzo Vainigli

Sessione III
Anno Accademico 2015-16

A mia mamma,
che mi ha permesso di intraprendere
questa strada e mi ha sempre supportato.

A me stesso,
perché non mi sono mai arreso e,
con impegno e dedizione,
ho raggiunto questo traguardo.

Sommario

Tenere traccia delle proprie entrate e delle proprie uscite monetarie può risultare molto utile se si vogliono avere degli indicatori sulla qualità della gestione del proprio denaro.

In questo testo viene presentata un'applicazione per dispositivi Android con la quale l'utente può salvare tutti i suoi movimenti monetari, con la possibilità di condividerli con altri utenti e di effettuare previsioni su eventi futuri.

Indice

1 Introduzione **15**

2 Stato dell'arte **17**

3 Architettura **21**
3.1 Movimenti . . . 21
3.2 Categorie . . . 21
3.3 Registri . . . 22
3.4 Previsioni . . . 22
3.5 Multiutenza . . . 23
3.6 Backup . . . 24
3.7 Report . . . 24
3.8 Notifiche . . . 24
3.9 Comunicazione client-server . . . 25
3.10 Sicurezza . . . 26

4 Implementazione dell'app Android **27**
4.1 Versioni supportate . . . 27
4.2 Pacchetti . . . 27
4.3 Activity, Fragment e Dialog . . . 28
4.3.1 LoginActivity . . . 28
4.3.2 RegisterActivity . . . 28
4.3.3 HomeActivity . . . 29
4.3.4 HomeFragment . . . 29
4.3.5 AccountsFragment . . . 30
4.3.6 CategoriesFragment . . . 31
4.3.7 MovementsListFragment . . . 32
4.3.8 MovementActivity . . . 33
4.3.9 FilterActivity . . . 33
4.3.10 DepreciationsFragment . . . 34
4.3.11 StatisticsFragment . . . 34
4.3.12 DatabasesFragment . . . 34

4.3.13 ReportsFragment . . . 35
4.3.14 BackupsFragment . . . 36
4.3.15 SettingsActivity . . . 36
4.3.16 IconsActivity . . . 36
4.4 Database . . . 37
4.5 DatabaseController . . . 38
4.6 DatabaseInfoController . . . 38
4.7 Model . . . 39
4.8 Data Access Object . . . 40
4.9 Adapter . . . 42
4.10 Controller . . . 43
4.10.1 FiltersController . . . 43
4.10.2 UserPrefsController . . . 44
4.10.3 ServerController . . . 45
4.11 Asynchronous Task . . . 45
4.11.1 La classe AsyncTask . . . 45
4.11.2 LoginAsyncTask . . . 46
4.11.3 RegisterAsyncTask . . . 46
4.12 Builder . . . 46
4.13 Receiver . . . 46
4.14 Utilità . . . 47
4.15 Librerie esterne . . . 47
4.15.1 HelloCharts . . . 47
4.15.2 iTextPDF . . . 47
4.15.3 VolleyPlus . . . 48
4.16 Testing . . . 48

5 Implementazione del server 49
5.1 Database . . . 49
5.2 API . . . 50
5.2.1 Login . . . 51
5.2.2 Registrazione . . . 52
5.2.3 Sincronizzazione dei registri . . . 53
5.2.3.1 Confronto . . . 53
5.2.3.2 Upload del database sul server . . . 54
5.2.3.3 Download del database sul device . . . 55
5.3 Testing . . . 56

6 Conclusioni e sviluppi futuri 57

Glossario 61

Acronimi 65

Riferimenti **67**

Elenco delle figure

1.1 Notazione dei diagrammi UML utilizzata in questa tesi. 16

3.1 Schema del flusso monetario nell'applicazione. 22
3.2 Esempio grafico di calcolo di una previsione. 23
3.3 Schema dell'architettura dell'applicazione *Denario*. 25
3.4 Schema di comunicazione tra client e server 26

4.1 Schermata di login 29
4.2 Schermata di registrazione 29
4.3 Diagramma delle classi di `LoginActivity` e `RegisterActivity`. 29
4.4 Diagramma delle classi di `HomeActivity`. 30
4.5 Schermata home 30
4.6 Menù principale 30
4.7 Diagramma delle classi di `HomeFragment`. 30
4.8 Diagramma delle classi di `AccountsFragment`. 31
4.9 Schermata che mostra la lista dei depositi. 31
4.10 Schermata che mostra la lista delle categorie. 31
4.11 Diagramma delle classi di `CategoriesFragment`, `CategoriesTabFragment`, `ExpensesTabFragment` e `IncomingsTabFragment`. 32
4.12 Lista dei movimenti registrati. 33
4.13 Schermata di visualizzazione di una previsione. 33
4.14 Schermata di impostazione dei filtri. 34
4.15 Esempio di notifica. 34
4.16 Diagramma delle classi di `MovementsListFragment`, `MovementsActivity` e `FilterActivity`. 35
4.17 Diagramma delle classi di *DepreciationsFragment*. 35
4.18 Diagramma delle classi di `SettingsActivity`. 36
4.19 Diagramma delle classi di `IconsActivity`. 37
4.20 Diagramma del database dell'applicazione *Android*. 38
4.21 Diagramma delle classi di `DatabaseController`. 40
4.22 Diagramma delle classi del pacchetto `models`. 41
4.23 Diagramma di sequenza di un *D.A.O.* 41

4.24 Diagramma delle classi di un *D.A.O.* e un *M.D.A.O.* 42
4.25 Diagramma delle classi di un *adapter* generico. 43
4.26 Diagramma delle classi di `BootUpReceiver`. 46

5.1 Gerarchia delle cartelle del server. 49
5.2 Diagramma ER del database del server. 50

Elenco delle tabelle

2.1 Tabella riassuntiva delle funzionalità di ogni applicazione. 20

4.1 Tabella `Utenti`. 37
4.2 Tabella `Registri`. 39
4.3 Tabella `Movimenti`. 39
4.4 Tabella `Categorie`. 40

5.1 Tabella `Utenti`. 50
5.2 Tabella `T_databases`. 51
5.3 Tabella `Permessi`. 51

Capitolo 1

Introduzione

Tutti noi, nel nostro vivere quotidiano, effettuiamo azioni che si ripetono giorno per giorno che, con i ritmi frenetici della nostra vita, sono difficili da curare nel minimo dettaglio.

Eppure, molte di queste azioni che oramai facciamo in modo automatico, sono dettate dal nostro modo di pensare razionalmente. Seguiamo un modello di ragionamento che presenta sempre le stesse proprietà, sulla base del quale decidiamo come comportarci. Nulla a che fare con la fantasia.

Possiamo tradurre il nostro ragionamento in uno schema da far eseguire ad una macchina. Risparmieremmo energia e guadagneremmo tempo utile. Perché non farlo allora?

Esattamente in questa ottica di ragionamento collochiamo il problema del monitoraggio del nostro denaro, poiché abbiamo delle spese da fare quasi tutti giorni e quindi una gestione accorta dei propri soldi non è mai un male. Tenere uno scaffale di libri contabili su cui annotiamo le nostre entrate e le nostre spese è roba da medioevo. Ci affidiamo all'informatica, nata proprio per effettuare computazioni su modelli matematici, ma i numerosi programmi "general purpose" non fanno al caso nostro. La loro natura infatti, li porta a non essere strumenti sofisticati praticamente per ogni problema.

Nasce allora il mio desiderio di progettare un modello che più si adatta al problema del budget tracking e sviluppare un'applicazione mobile che aiuti l'utente nella gestione dei suoi flussi monetari. Mi sono affidato alle applicazioni per dispositivi mobili poiché questo settore oramai si presenta come il fulcro dell'innovazione informatica.

Denario, nome che si ispira ad un'antica moneta di epoca romana, è un'applicazione per *Android* progettata per organizzare i flussi monetari personali. L'utente, con il suo dispositivo, può catalogare i movimenti di denaro personali, sia in entrata che in uscita, al fine di produrre dati aggregati come statistiche, grafici e previsioni che meglio sintetizzano la qualità della gestione del patri-

monio personale. L'applicazione dispone anche di un server grazie al quale è stato possibile inserire i registri condivisi: due o più utenti, infatti, possono condividere parte dei propri movimenti.

In questo testo viene esaminata l'applicazione passo per passo, dall'aspetto architetturale a quello implementativo, esponendone le funzioni principali e le modalità con le quali sono state realizzate.

Notazione

Durante la redazione di questa tesi sono state utilizzate molte immagini e alcune formattazioni del testo per rendere più intuibili i concetti che verranno affrontati.

Le parole chiave sono raggruppate come segue, ogni gruppo con un proprio stile.

- I nomi di `file`, `cartelle`, `pacchetti`, `classi`, `parametri`, `metodi`, `variabili` e `tabelle` dei database sono scritti con un font monospazio.
- Le *parole chiave* che possiedono una definizione nel glossario sono scritte con una formattazione in corsivo.
- Nomi di **pulsanti o voci dei menù** che l'utente può premere quando utilizza l'applicazione sono evidenziati in grassetto.

Trattandosi di programmazione orientata agli oggetti l'uso di diagrammi diventa fondamentale per capire il comportamento di ogni classe. La sintassi utilizzata è quella di Unified Modeling Language (UML) e rappresentata in figura 1.1. Esaminando proprio questa figura abbiamo la classe `Class A` che è definita nel pacchetto `Package`, la classe `Class B` estende la `Class A` e utilizza la classe `Class C`, nel senso che nel codice della classe `Class B` vengono istanziati oggetti di tipo `Class C`. Infine, la classe `Class C` utilizza la risorsa `Resource`.

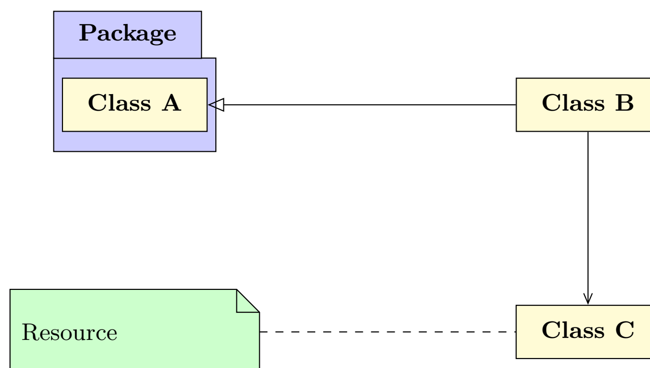

Figura 1.1: Notazione dei diagrammi UML utilizzata in questa tesi.

Capitolo 2

Stato dell'arte

Il tema della gestione o del monitoraggio dei flussi monetari (*budget tracking* in inglese) non è una disciplina che viene trattata negli articoli scientifici, ma piuttosto una categoria di applicazioni (sia mobile che web) che possono giocare un ruolo fondamentale per l'individuo che vuole controllare i suoi introiti e le sue spese.

Per capire quali sono gli standard per questa tipologia di applicazioni, sono state scelte dal Google Play Store cinque tra le applicazioni di *budget tracking* più scaricate dagli utenti *Android*. Le applicazioni in questione sono Money Lover [11], Money Manager [12], Monefy [10], AndroMoney [1] ed Expense IQ [4].

Categorie Inizialmente va notato che ogni applicazione che traccia i flussi monetari ha bisogno di una suddivisione dei movimenti in categorie. Questo è un primo passo fondamentale per iniziare a modellare la realtà, in fin dei conti, non si possono paragonare le spese per il caffè quotidiano con le spese per l'affitto, come non si possono paragonare gli introiti dello stipendio con quelli per una prestazione occasionale.

Va inserita una nota per l'applicazione *Money Lover* che implementa un'interfaccia apposita per le bollette con promemoria per la data di scadenza.7

Depositi multipli Il modello più comune è quello in cui i movimenti in entrata vengono caratterizzati da una provenienza e quelli in uscita da una destinazione. Ovviamente in questo percorso il denaro passa per i depositi e anche in questo caso nessuna applicazione fa eccezioni: tutte danno la possibilità di definire uno o più depositi per l'utente (es. portafoglio, conto bancario).

Multivaluta La possibilità di scegliere la valuta per gli importi è una caratteristica che accomuna tutte le applicazioni prese in esame, poiché il loro target di

utenza corrisponde al mercato globale. Tuttavia è anche vero che difficilmente si avranno, per lo stesso utente, movimenti di diverse valute.

Esportazione e backup dei dati Per completare la parte delle funzioni offerte da tutte le applicazioni, menzioniamo anche la possibilità di esportare i dati, ossia di effettuare *backup* e l'eventualità di proteggere, trattandosi di dati sensibili, i propri *movimenti* con una password.

Movimenti ricorrenti Nella realtà ci sono delle transazioni periodiche come l'affitto e lo stipendio. Molto utile in questo caso sarebbe la possibilità, nel caso che vi siano movimenti futuri di cui già conosciamo la data e l'importo, di inserire con una sola azione più istanze dello stesso movimento. Infatti, in questo caso, tutte le applicazioni ad eccezione di Monefy offrono questa funzione.

Filtri Quando si ha a che fare con una grande quantità di dati diventa necessario usufruire di strumenti di ricerca oppure di filtraggio, in modo da poter osservare i dati da diversi punti di vista. Le applicazioni Money Manager, AndroMoney ed Expense IQ offrono dei sistemi ben strutturati di filtraggio, con la possibilità di impostare più parametri, mentre le altre hanno dei filtri preimpostati.

Sincronizzazione Avere i dati salvati solo in locale può essere un pericolo in caso di malfunzionamento del dispositivo e per questo Money Lover, Monefy, AndroMoney ed Expense IQ mettono a disposizione un meccanismo di sincronizzazione remota. L'applicazione Money Lover richiede una registrazione per poterla utilizzare, mentre per le altre si può scegliere un proprio account di *cloud storage* sul quale caricare i dati.

Statistiche In Money Lover, Money Manager e Expense IQ possiamo trovare delle schermate con grafici e somme che riescono a riassumere un determinato periodo oppure una determinata categoria. Nelle altre due applicazione, invece, questa funzione è più limitata.

Pianificazione del budget Uno dei modi per sfruttare a fondo la suddivisione dei movimenti in categorie è la possibilità di assegnare a ciascuna un limite di budget utilizzabile. Non a caso tre applicazioni delle cinque scelte hanno questa funzione.

In Denario questa funzione, seppur interessante, non è stata implementata perché avrebbe richiesto molto tempo.

Altre funzioni L'applicazione Money Lover, che oggettivamente risulta essere quella più completa, dà la possibilità di creare degli eventi ai quali attribuire alcuni movimenti (per esempio, le spese di un viaggio) in modo tale da "separarli" dal resto dei dati. Anche i debiti e i crediti sono presi in considerazione, infatti si possono inserire transazioni di questi due tipi.

Sono state appena esposte le principali funzionalità delle applicazioni prese in esame, di cui è riportato uno schema nella tabella 2.1

Funzioni inedite L'applicazione Denario, trattata in questa tesi, si prefigge di implementare due funzioni inedite: la ripartizione di un movimento nel suo periodo e un algoritmo per effettuare delle previsioni su probabili movimenti futuri

Funzione	Money Lover	Money Manager	Monefy	AndroMoney	Expense IQ
Suddivisione in categorie	Divisione tra categorie di entrata, di spesa, di debiti e prestiti. Schermata con elenco di tutte le categorie e sotto-categorie.	Due livelli di categorie definibili in fase di inserimento di un movimento.	Divisione tra categorie di entrata e di spesa.	Due livelli di categorie divise in tre tipologie: uscite, entrate e trasferimenti.	Categorie e sotto-categorie di entrata e di uscita.
Creazione di depositi multipli	Solo due per la versione gratuita.	Definibili scegliendo un macro-gruppo di appartenenza.	Sì	Sì	Sì con possibilità di inserire un saldo iniziale per ogni deposito.
Multivaluta	Definibile per ogni movimento.	Definita in modo globale.	-	Definita in modo globale.	Definita in modo globale ma possibilità di scelta diversa per i conti (depositi) e per i promemoria.
Esportazione dati	CSV ed Excel.	Excel	CSV	CSV	CSV e QIF.
Backup	Manuali ed automatici.	Solo manuali in locale ed esportabile via e-mail	Solo manuali in locale.	Solo manuali in locale.	Manuali ed automatici con opzione per automatica eliminazione dei backup più vecchi.
Inserimento di movimenti ricorrenti	Inserimento ripetibile per/ogni n giorni, settimane, mesi o anni per x volte, fino a data relativa o per sempre.	Per ogni movimento si può scegliere il periodo di ripetizione	-	Si può attribuire una periodicità ad ogni movimento.	Inserimento ripetibile per/ogni n giorni, settimane, mesi o anni per x volte, fino a data relativa o per sempre.
Filtri	Movimenti raggruppati per categoria o per transazione.	Per categorie e solo di supporto alle statistiche.	Per periodo e per deposito.	Ogni categoria può essere inclusa o esclusa dal filtraggio.	Si possono visualizzare le transazioni per ogni conto o categoria.
Sincronizzazione remota	Automatica e manuale.	-	-	Sì	-
Statistiche	Definite su spese, entrate o utile e calcolate su base mensile. Distinguibili nel tempo o per categoria.	Per periodo e per categoria, sempre distinte tra guadagno e spesa.	Solo totale e differenza sul periodo di riferimento.	Su periodo e categorie. Disponibili grafici a torta e a barre con interfaccia molto sofisticata.	Possibilità di scelta tra alcune impostazioni predefinite sul periodo e sulla tipologia di movimento (es. report delle entrate mensili).
Pianificazione del budget	Definibili come "budget" o "risparmi".	Solo per le categorie di spesa.	Solo su base mensile e globale rispetto alle categorie di spesa.	Definibile per le spese su base giornaliera, settimanale, mensile e annuale.	Sì

Tabella 2.1: Tabella riassuntiva delle funzionalità di ogni applicazione.

Capitolo 3

Architettura

Per iniziare ad esplorare l'applicazione Denario, vediamo quali solo le funzioni e le caratteristiche che possiede.

3.1 Movimenti

I movimenti sono le entità di base sulle quali è costruita l'applicazione. Sono strutture dati che raccolgono tutte le informazioni che caratterizzano un movimento di denaro, come data, descrizione e importo. Ogni flusso di denaro ha una fonte e una destinazione che sono rappresentate da categorie (figura 3.1).

Infine, ci sono dei movimenti che hanno un periodo di competenza, per esempio, il pagamento di una rata di affitto o una bolletta. L'utente può allora registrare un periodo di competenza in cui l'impatto della spesa deve essere ripartito.

Filtri Quando si ha a disposizione un numero molto grande di movimenti può risultare difficile consultarli. Per questo è stata sviluppata una funzione per il filtraggio dei movimenti nella quale si possono impostare i parametri desiderati.

3.2 Categorie

Tutte le applicazioni di *budget tracking* offrono la suddivisione dei movimenti in categorie perché è una funzione fondamentale al fine di catalogare i flussi monetari. Denario fa di questa funzione uno dei suoi punti cardine. Grazie a ciò, l'utente può esaminare più a fondo ogni tipo di movimento al fine di adottare strategie diverse a seconda delle categorie che più gli interessano.

Le categorie si dividono in categorie di *fonte*, categorie di *deposito* e categorie di *destinazione*. Come illustrato in figura 3.1, il denaro che proviene dalle categorie di fonte viene immagazzinato, tramite un'operazione di *entrata*, nei

depositi dell'utente, dove rimane a sua disposizione. L'utente quindi è libero di spostare il denaro da un deposito all'altro tramite un *trasferimento*. I soldi stanziati nei depositi è destinato a coprire le *uscite*, ovvero gli spostamenti di denaro da un deposito ad una destinazione.

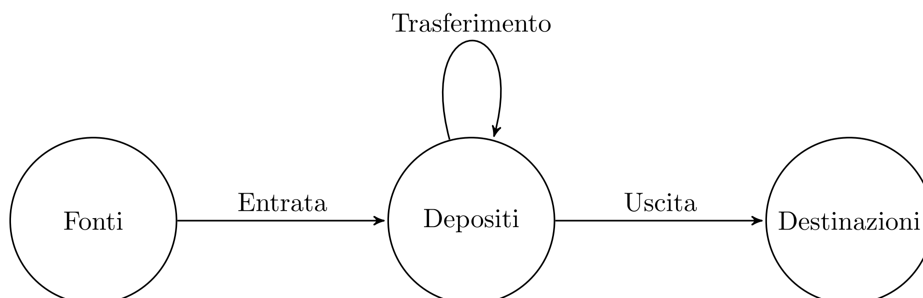

Figura 3.1: Schema del flusso monetario nell'applicazione.

3.3 Registri

I movimenti vengono organizzati in *registri*, che aggiungono un ulteriore livello alla gerarchia creata dall'applicazione. Questo ulteriore raggruppamento, dopo le categorie, è stato pensato per permettere a due o più utenti di condividere una parte dei propri movimenti con altri utenti, mantenendo la riservatezza sugli altri dati.

Registri condivisi L'applicazione supporta la creazione di database condivisibili con altri utenti. Proprio come accade con i conti che presentano più di un intestatario, è possibile aggiungere registri che possono essere modificati sia dall'utente stesso, sia da altri utenti. Ogni registro è protetto da una password e agli utenti che vogliono condividerlo basta semplicemente scambiarsi la password del database.

3.4 Previsioni

Se si ha un registro di tutti i movimenti di una determinata categoria, può essere utile cercare di stimare l'entità del prossimo movimento. L'applicazione effettua una semplice previsione basata sulle registrazioni passate e restituisce un range di importo e di tempo dentro al quale si prevede che si verificherà il prossimo movimento, come illustrato in figura 3.2. Per fare una previsione sul prossimo movimento si ricorre a due concetti della statistica: la media e lo scarto quadratico medio (o deviazione standard).

Sia data una successione di numeri reali x_i con $i = 1..n$. La **media**, intesa come media aritmetica ed indicata con la lettera μ, è definita come:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Questo valore appena introdotto riassume tutti i valori degli x_i in un unico valore μ. Questo però non è abbastanza soddisfacente poiché l'obiettivo è restituire all'utente un intervallo di valori, non un singolo numero.

A questo scopo si introduce lo **scarto quadratico medio**, indicato con la lettera σ e calcolato con la formula:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(|x_i - \mu|)^2}{n}}$$

Adesso si può definire la previsione come l'intervallo di valori

$$[\mu - \sigma, \mu + \sigma]$$

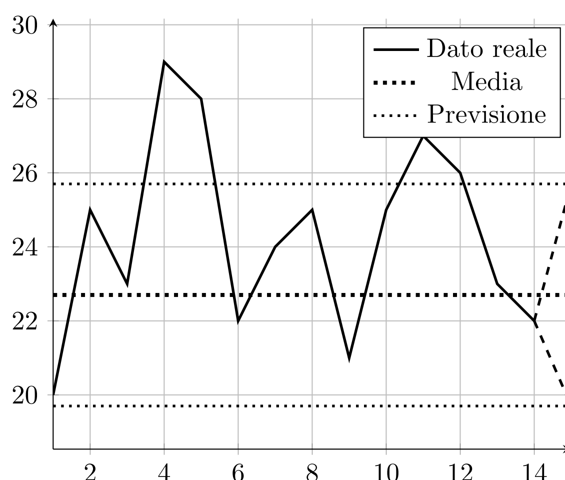

Figura 3.2: Esempio grafico di calcolo di una previsione.

3.5 Multiutenza

La maggioranza del dispositivi con sistema operativo *Android* sono accessori ad uso personale, quasi sempre non condiviso con altri soggetti. Tuttavia ci sono casi in cui si ha bisogno di trasferire i propri dati su un altro dipositivo oppure

effettuare l'accesso da uno altro smartphone. I dati dei movimenti registrati nell'applicazione vengono, a questo scopo, copiati su un server remoto, da cui è possibile scaricarli successivamente.

Protezione dei dati sensibili Ogni utente ha accesso solo ai propri dati attraverso una connessione sicura *HyperText Transfer Protocol over Secure Socket Layer (HTTPS)*. I dati dell'utente non viaggiano mai in chiaro e sul server sono memorizzati in forma cifrata.

3.6 Backup

I *backup* dei dati, in informatica, sono una buonissima contromisura per prevenire la loro perdita. Di ogni registro può essere salvata la sua istanza in qualsiasi momento per essere spostata su un altro supporto di archiviazione.

3.7 Report

Una ulteriore funzione offerta dall'applicazione è la possibilità di generare report, ossia documenti riassuntivi, dei movimenti, per i quali verranno stampati i dati secondo il seguente schema:

- tabella dei movimenti registrati nel primo registro;
- tabella dei movimenti registrati nel secondo registro;
- ...
- tabella dei movimenti registrati nel n-esimo registro;
- indicatori su tutti i movimenti di tutti i registri (totali correnti, totali futuri e differenze).

I file generati sono in formato Portable Document Format (PDF), in modo da essere consultati con altre applicazioni oppure stampati.

3.8 Notifiche

Oramai tutte le applicazioni mobile fanno uso di notifiche poiché sono uno strumento molto utile per attirare l'attenzione dell'utente quando l'applicazione ha a disposizione informazioni importanti.

Denario utilizza questo principio per avvisare l'utente con delle notifiche all'accensione del dispositivo quando ci sono spese programmate o previsioni per i prossimi giorni. In questo caso, l'utente può scegliere il numero di giorni su cui fare il controllo (per esempio, 3 o 7 o 14 giorni).

3.9 Comunicazione client-server

Come accennato nella sezione precedente, l'applicazione *Android* Denario è supportata da un *server* in cui vengono salvati i dati degli utenti.

Una volta installata l'applicazione sul proprio dispositivo, l'utente deve effettuare la registrazione connettendosi al server. In questo modo, può usufruire di un profilo al quale sono associati i file ed i registri (database) che verranno sincronizzati in remoto. Alla successiva operazione di login non sarà necessario connettersi al server, in quanto le credenziali corrette saranno salvate in locale.

La sincronizzazione dei database sul server è il modo con cui viene soddisfatto il principio di condivisione dei dati che caratterizza questa applicazione; basterà infatti avere i permessi e conoscere la password per scaricare un registro condiviso da un altro utente.

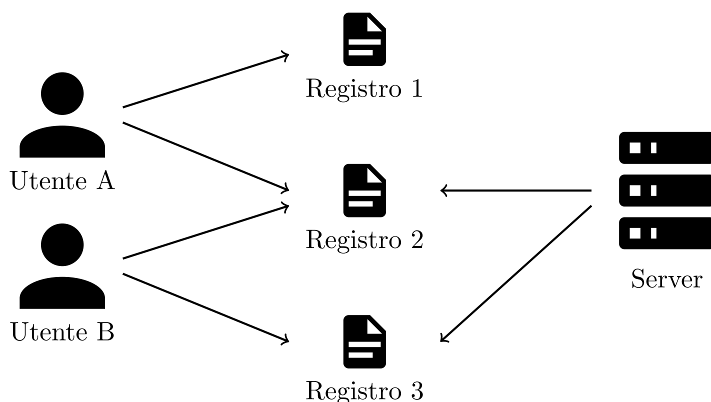

Figura 3.3: Schema dell'architettura dell'applicazione *Denario*.

Secondo quanto descritto nella figura 3.3, l'utente A dispone di due registri: il registro 1, non sincronizzato sul server ed il registro 2, di cui il server mantiene una copia. L'utente B ha accesso condiviso con l'utente A al registro 2, mentre il registro 3, seppur sincronizzato in remoto, ha accesso limitato al solo utente B. Lo scambio di dati tra *client* (applicazione *Android*) e *server* avviene secondo il seguente schema:

1. L'applicazione, utilizzando la libreria `Volley` di *Android* effettua una richiesta *HTTPS POST* ad una delle url del server che forniscono le *Application Program Interface (API)*;

2. Gli script del server processano la richiesta e restituiscono un file Javascript Object Notation (JSON) contenente un codice che identifica l'esito della computazione.

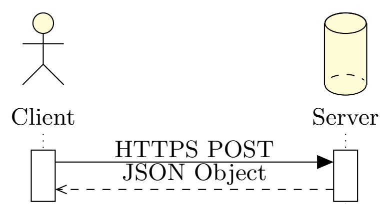

Figura 3.4: Schema di comunicazione tra client e server

3.10 Sicurezza

Le informazioni sui movimenti di denaro sono considerati dati sensibili, pertanto è necessario fornire meccanismi che assicurino all'utente che, anche in caso di perdita o furto dei dati, chiunque non abbia i diritti per accedervi non possa farlo.

In ambiente locale, ovvero sul dispositivo mobile, in fase di creazione di ogni registro, è chiesta una password di protezione del file. L'utente può decidere se cifrare i file sul dispositivo oppure mantenerli in chiaro.

In remoto, dove i file di tutti gli utenti vengono salvati in una cartella comune, diventa obbligatorio che siano cifrati.

In fase di trasferimento dei file da client a server e viceversa viene utilizzata una connessione sicura.

Capitolo 4

Implementazione dell'applicazione Android

Di seguito sono descritte tutte le parti che compongono l'applicazione vera e propria. Sarà seguito un approccio *top-down*, partendo dalle *activity* che controllano la *User Inteface (UI)* fino ad arrivare al database, dove vengono salvati i dati persistenti. Per finire, verranno descritti alcuni strumenti di supporto all'esecuzione dell'applicazione.

4.1 Versioni supportate

L'applicazione è stata sviluppata con *target API version* 21 (Lollipop) e *minimum API version* 16 (Jelly Bean). Questo è stato fatto per assicurare una copertura quasi totale di tutti i dispositivi *Android* attualmente in circolazione.

4.2 Pacchetti

Come fornito dalla piattaforma di sviluppo, è stato utilizzato il linguaggio *Java* e le classi che compongono l'applicazione sono suddivise in pacchetti, a seconda del ruolo svolto.

- `activities`: implementano le *activity*, come quella per il login e quella principale.
- `adapters`: implementano gli *adapter* utilizzati per le liste.
- `builders`: classi che costruiscono oggetti o file con una forma specifica (report e backup). Si ispirano al design pattern *builder*.
- `controllers`: classi che forniscono interfacce per gestire i filtri e le richieste al *server*, infatti corrispondono al design pattern *controller*.

- `database`: contiene tutte le classi che si occupano della creazione del database oppure effettuano richieste dirette a quest'ultimo. Si trovano in questo pacchetto le classi di tipo *Data Access Object (DAO)* e *Multiple Data Access Object (MDAO)* .
- `dialogs`: implementano i *dialog*.
- `fragments`: si trovano qui le numerose classi che realizzano i *fragment* dell'applicazione
- `fragments.tabs`: i *fragment* delle categorie e delle statistiche sono divisi in *tab* e qui sono salvate le classi che li implementano.
- `models`: classi che realizzano dei modelli rappresentanti le entità concettuali utilizzate nell'applicazione, come i movimenti, le categorie ed i registri.
- `receivers`: contiene solo la classe che implementa il *receiver* per le notifiche.
- `utils`: classi di metodi con utilizzo frequente per conversioni di tipo o crittografia, che facilmente si prestano al riuso in altri progetti.

4.3 Activity, Fragment e Dialog

Le activity, ovvero le entità principali che compongono l'interfaccia utente di ogni applicazione *Android*, sono le seguenti (I loro nomi corrispondono all'omonima classe *Java* che le implementa).

4.3.1 LoginActivity

Dopo essere stata installata oppure dopo un'operazione di logout, l'applicazione mostra la schermata 4.1, costruita per permettere all'utente di autenticarsi con username e password.

Il tasto **Login** chiama il metodo `authenticate()` della classe `ServerController`.

Il tasto **Registrati** apre una nuova `RegisterActivity`.

4.3.2 RegisterActivity

Simile alla precedente *activity*, la schermata 4.2 permette all'utente di inserire i dati per la registrazione, ovvero nome, username e password. Ovviamente, non può mancare il controllo sulla correttezza della password e all'utente viene chiesto di inserirla due volte.

Il tasto **Registrati** chiama il metodo `register()` della classe `ServerController`.

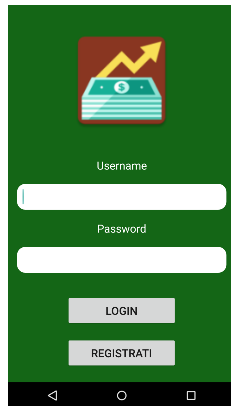

Figura 4.1: Schermata di login

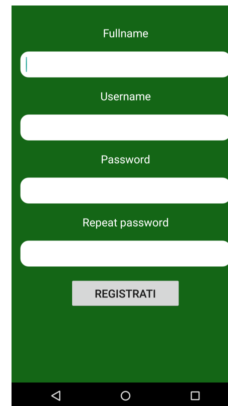

Figura 4.2: Schermata di registrazione

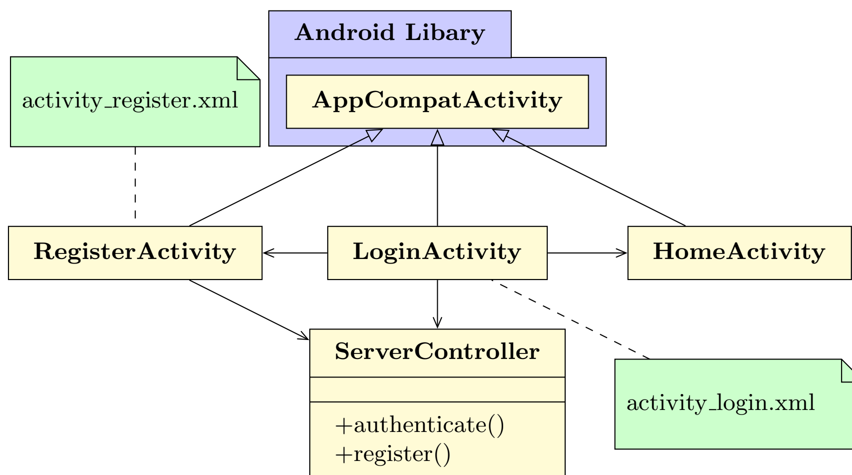

Figura 4.3: Diagramma delle classi di `LoginActivity` e `RegisterActivity`.

4.3.3 HomeActivity

Questa activity è la principale: ospita layout del menù principale che si trova sulla sinistra dell'intefaccia, che è stato implementato con un `Navigation Drawer`. Oltre al menù e la `Action Bar` in alto che contiene il titolo dell'activity, vi è un container che copre quasi tutta la UI e nel quale vengono poi mostrati i *fragment* corrispondenti alle voci del menù principale.

4.3.4 HomeFragment

È la classe che compone la schermata principale e proprio per questo motivo mostra informazioni generali e riassuntive sui dati che ha a disposizione l'applicazione, come un grafico che mostra il saldo attuale, gli ultimi movimenti registrati e i prossimi movimenti in programma.

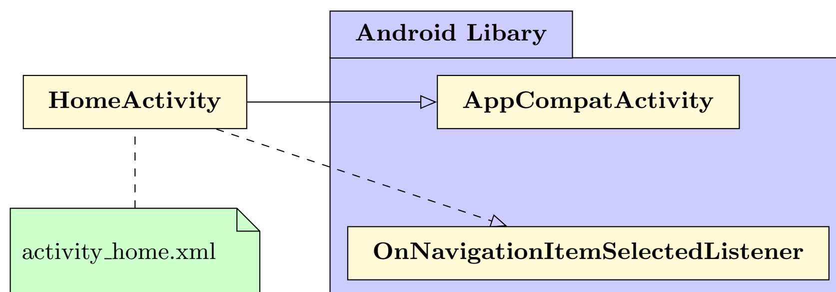

Figura 4.4: Diagramma delle classi di `HomeActivity`.

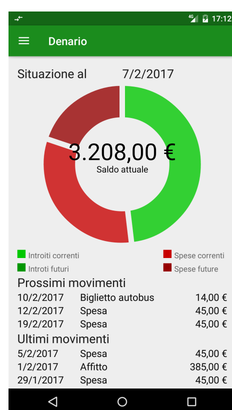

Figura 4.5: Schermata home

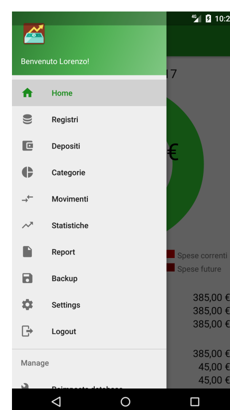

Figura 4.6: Menù principale

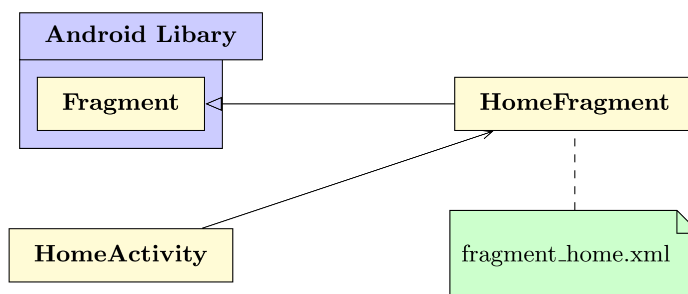

Figura 4.7: Diagramma delle classi di `HomeFragment`.

4.3.5 AccountsFragment

Come già introdotto, i flussi di denaro dell'utente devono necessariamente passare dai suoi depositi. Una semplice lista mostra l'elenco dei depositi definiti dall'utente e la corrispettiva quantità di denaro che contengono, che in realtà non è altro che la differenza tra la somma dei movimenti che hanno il deposito

registrato come categoria di destinazione e la somma dei movimenti che hanno il deposito registrato come categoria di fonte.

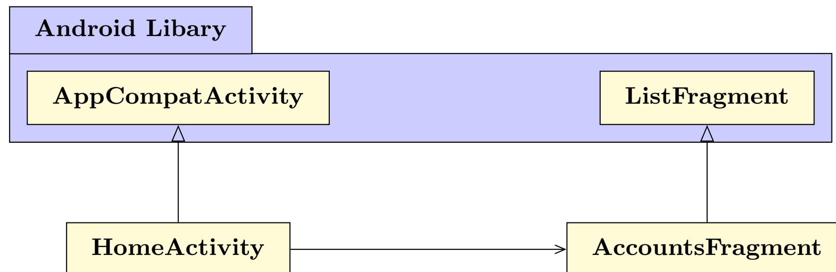

Figura 4.8: Diagramma delle classi di `AccountsFragment`.

4.3.6 CategoriesFragment

Questo *fragment* è concettualmente e visivamente simile al precedente, ma con la differenza che ogni voce (categoria) deve necessariamente appartenere alle categorie di fonte oppure alle categorie di destinazione. Vi è infatti una divisione del *layout* in due *tab*, uno per le fonti e l'altro per le destinazioni:

- `ExpensesTabFragment` mostra una lista di tutte le categorie di uscita (spesa);
- `IncomingsTabFragment` mostra una lista di tutte le categorie di entrata (introito).

Questi due *tab* inseriti con lo scopo di migliorare la navigabilità e, data la loro struttura identica, alcune proprietà comuni sono state trasferite nella classe `CategoriesTabFragment`.

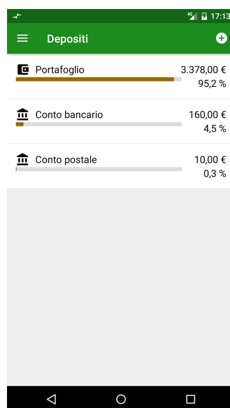

Figura 4.9: Schermata che mostra la lista dei depositi.

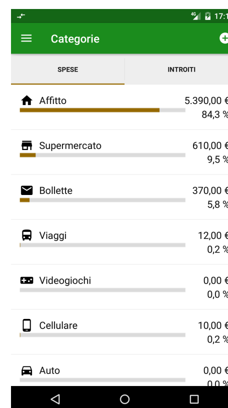

Figura 4.10: Schermata che mostra la lista delle categorie.

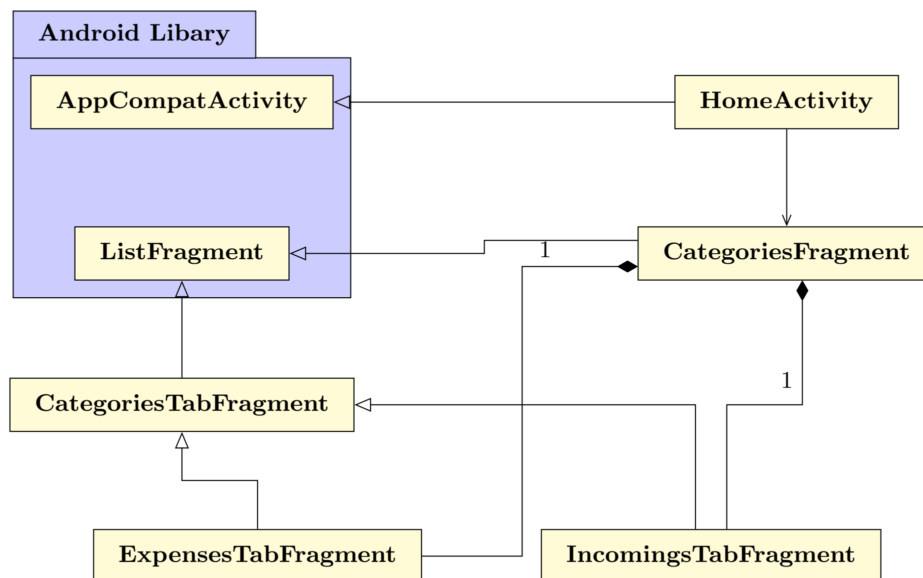

Figura 4.11: Diagramma delle classi di `CategoriesFragment`, `CategoriesTabFragment`, `ExpensesTabFragment` e `IncomingsTabFragment`.

4.3.7 MovementsListFragment

La lista dei movimenti visualizzata in questo *fragment* permette all'utente di controllare i principali dati che caratterizzano i movimenti: data, descrizione e importo. Le categorie di fonte e di destinazione sono rappresentate dalle icone identificative.

Menù Il menù principale di questo *fragment* presenta una serie di voci:

- **Attiva o disattiva filtraggio**: è un pulsante concepito per rendere più comoda l'operazione di filtraggio dei dati. Se il filtraggio è disattivato, il pulsante è rappresentato dall'icona ⊽, mentre se il filtraggio è attivo, compare l'icona ▼.

- **Aggiungi movimento(⊕)**: apre un'instanza di `DetailMovementActivity` da utilizzare per inserire un nuovo movimento.

- **Gestisci filtri**: apre un'*activity* di tipo `FilterActivity` per gestire tutti i parametri di filtraggio.

- **Ordina**: mostra un piccolo dialog dal quale si può scegliere di ordinare la lista dei movimenti per data decrescente o crescente oppure per prezzo decrescente o crescente.

- **Esporta PDF**: effettua una chiamata a `ReportBuilder` e genera un file PDF con tutti i dati dei registri, depositi, categorie e movimenti.

Menù contestuale Effettuando un *long-click* su un elemento della lista si apre un menù contestuale dal quale si scegliere:

- **Dettagli** per aprire la schermata di modifica del movimento rappresentata della classe `MovementActivity`;
- **Elimina** per eliminare il movimento dalla lista e dal database.

4.3.8 MovementActivity

È la schermata che permette all'utente di inserire i dati inerenti al singolo movimento di denaro, ovvero: data, registro, categoria di fonte, categoria di destinazione, importo e, opzionalmente, data di inizio e di fine del periodo di competenza. Viene utilizzata anche in fase di modifica di un movimento già registrato.

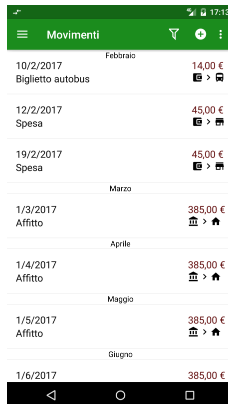

Figura 4.12: Lista dei movimenti registrati.

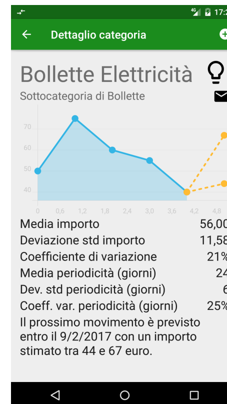

Figura 4.13: Schermata di visualizzazione di una previsione.

4.3.9 FilterActivity

La lista di tutti i movimenti mostrata in `MovementsListFragment` può non essere molto comoda se l'utente è in cerca di informazioni che riguardano solo una selezione di tutti i movimenti. In questo caso, alcune funzioni di filtraggio dei dati possono rivelarsi fondamentali.

L'applicazione dispone di una schermata che permette di impostare ogni parametro del modello dei movimenti: registro, categoria di fonte e di destinazione, range di importo e periodo. Il range di importo è identificato da una somma minima e una massima, mentre il periodo è caratterizzato da una data di inizio e una di fine, venendo poi confrontato con la data di registrazione di ogni movimento. I parametri impostati dall'utente vengono salvati nelle `SharedPreferences`.

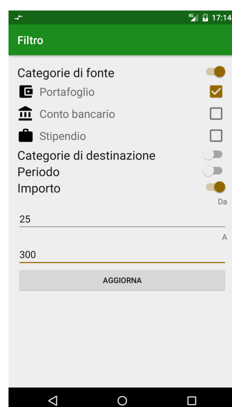

Figura 4.14: Schermata di impostazione dei filtri.

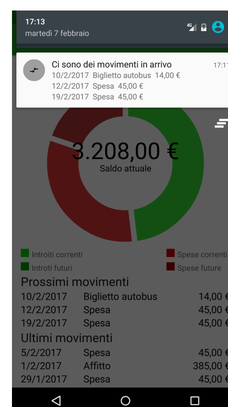

Figura 4.15: Esempio di notifica.

4.3.10 DepreciationsFragment

In questa applicazione c’è la possibilità di inserire un periodo di competenza per ogni singolo movimento. Questa classe crea una schermata che contiene informazioni sulla percentuale dell’importo che è maturata alla data odierna.

4.3.11 StatisticsFragment

In questo fragment vengono mostrati dei dati che sono il risultato di totali, medie e differenze sui movimenti registrati. L’interfaccia è stata divisa in due tab:

- `StatisticAllTabFragment`: contiene due tabelle. La prima mostra tre voci: totale degli introiti, totale delle spese e differenza, cioè il saldo. Ognuna delle precedenti voci è divisa tra movimenti correnti e movimenti futuri. La seconda tabella divide i totali per periodi diversi (settimana, mese e anno) e li confronta con le rispettive medie.

- `StatisticPeriodTabFragment`: l’utente può visualizzare questo fragment per conoscere meglio i totali per ogni periodo, che sia annuali, mensile, settimanale o giornaliero. Si può scegliere, tramite un dialog, il criterio di raggruppamento.

4.3.12 DatabasesFragment

L’utente che desidera gestire i propri registri può farlo attraverso questa interfaccia che contiene una lista di tutti i registri definiti dall’utente, i quali possono essere classificati nelle seguenti tipologie:

- registri privati salvati il locale e non sincronizzati con il server;

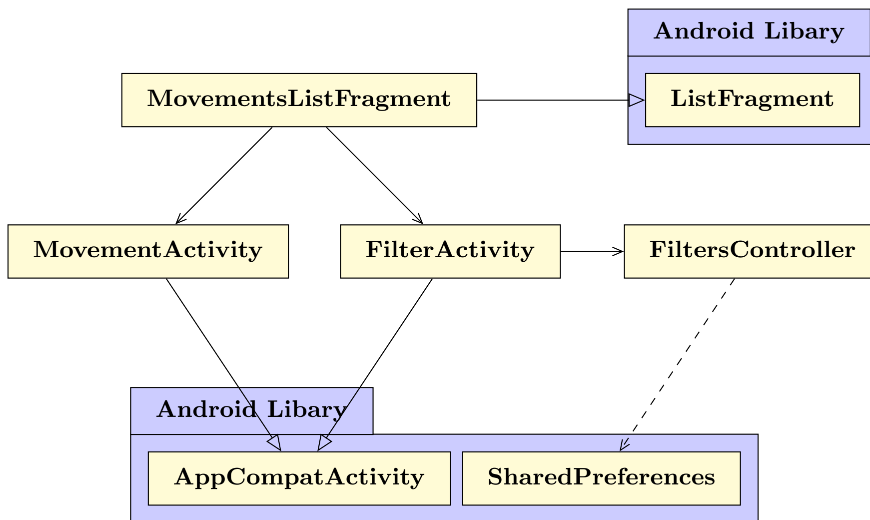

Figura 4.16: Diagramma delle classi di `MovementsListFragment`, `MovementsActivity` e `FilterActivity`.

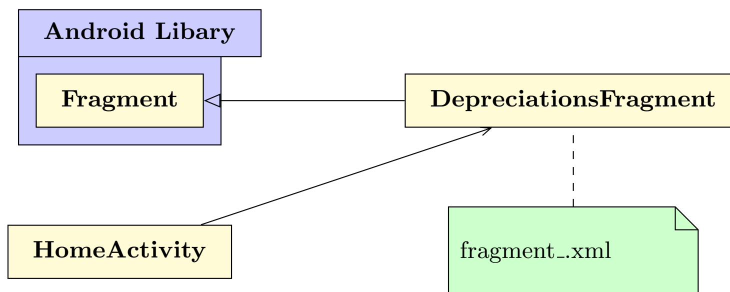

Figura 4.17: Diagramma delle classi di *DepreciationsFragment*.

- registri privati sincronizzati con il server;
- registri condivisi (questa tipologia deve essere sincronizzata con il server).

Menu Il menù di questo fragment contiene solo due voci:

- **Aggiungi registro** (⛁+): permette di aggiungere un registro condiviso scaricandolo dal server. L'utente deve conoscere l'identificativo remoto, la password e inoltre deve avere i permessi per accedervi.
- **Crea registro**: crea un nuovo registro in locale.

4.3.13 ReportsFragment

I file di report generati dal menù del fragment `MovementsListFragment` vengono mostrati in una semplice lista dalla quale si può aprire il file oppure eliminarlo.

Menu contestuale Con il consueto click prolungato su uno degli elementi della lista, in questo caso file *PDF*, si può scegliere una delle seguenti voci:

- **Apri**: consente di aprire il file. Nel dettaglio, viene lanciato un *intent* con l'informazione di aprire un file PDF (`application/pdf`) che viene catturato da un'applicazione del dispositivo in grado di leggere questo tipo di file.

- **Elimina**: elimina il file.

4.3.14 BackupsFragment

Questo fragment è molto simile al precedente, infatti anche qui è presente una lista di file, che in questo caso sono archivi *zip* generati in locale che contengono registri, report e preferenze. In altre parole, tutti i file creati dall'utente vengono raccolti e accantonati con lo scopo di catturare lo stato attuale dell'applicazione. Successivamente l'utente può decidere di ristabilire uno stato precedente riaprendo uno dei backup salvati.

4.3.15 SettingsActivity

L'utente deve aprire questa activity per scegliere le impostazioni globali dell'applicazione, tra le queli l'attivazione delle notifiche. Anche questa classe utilizza le `SharedPreferences`.

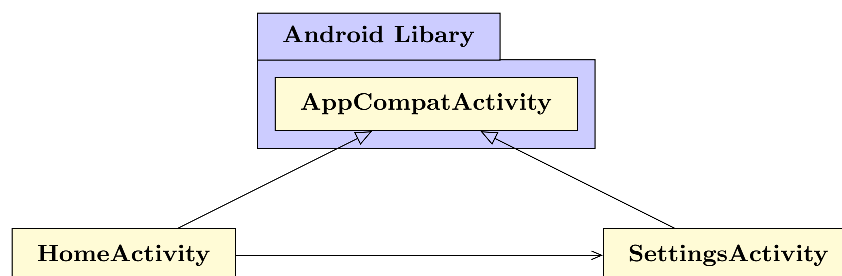

Figura 4.18: Diagramma delle classi di `SettingsActivity`.

4.3.16 IconsActivity

Ad ogni categoria Denario associa un'icona identificativa, in stile material design, che viene scelta dall'utente tra un insieme predefinito dall'applicazione e inserita durante la creazione di ogni categoria, tramite una schermata apposita.

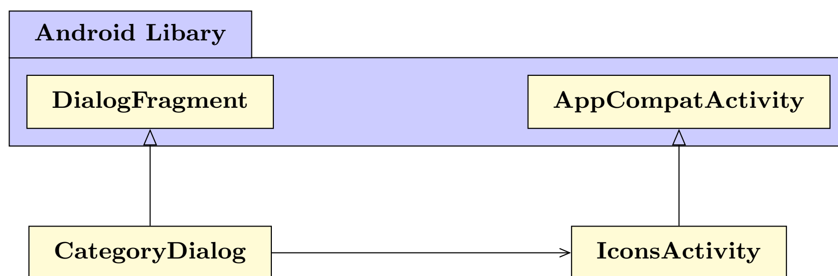

Figura 4.19: Diagramma delle classi di `IconsActivity`.

4.4 Database

Lo scopo principale dell'applicazione è elaborare una grande quantità di dati che rappresentano i movimenti monetari dell'utente. Tutto ciò non può essere rappresentato se non facendo uso di una base di dati. Il database nel quale Denario memorizza i dati, quello riguardante la parte *client* e non la parte *server*, utilizza *SQLite*.

Ricapitolando, abbiamo dei registri in cui vengono memorizzati i movimenti, che a loro volta hanno una categoria di fonte e una categoria di destinazione. Le categorie si distinguono in primarie e secondarie, queste ultime hanno una categoria primaria di appartenenza. Lo schema logico dell'intera base di dati è quello mostrato in figura 4.20.

Per la traduzione dallo schema logico allo schema fisico, ogni registro è stato considerato come una database a sé stante, in modo da poter implementare la condivisione dei registri come semplice condivisione di file.

Allora, nella cartella `databases` interna alla cartella dell'applicazione che si trova nel dispositivo ci sono due tipi di file:

- Il database `info.db` composto dalla tabella `Utenti` (4.1) e dalla tabella `Registri` (4.2) contentente i metadati sui file che rappresentano i registri.
- Uno o più database, in qualità di registri, per ogni utente, che contengono la tabella `Movimenti` (4.3) e la tabella `Categorie` (4.4). I nomi di questi file sono della forma `[USERNANE_UTENTE]_[NOME_REGISTRO].db`.

Campo	**Tipo**	**Riferimento**	**Note**
Nome	Testo		
Username	Testo	Chiave primaria	Username utilizzato per il login.
MD5_password	Testo(32)		Stringa *hash* della password utilizzata per il login.

Tabella 4.1: Tabella `Utenti`.

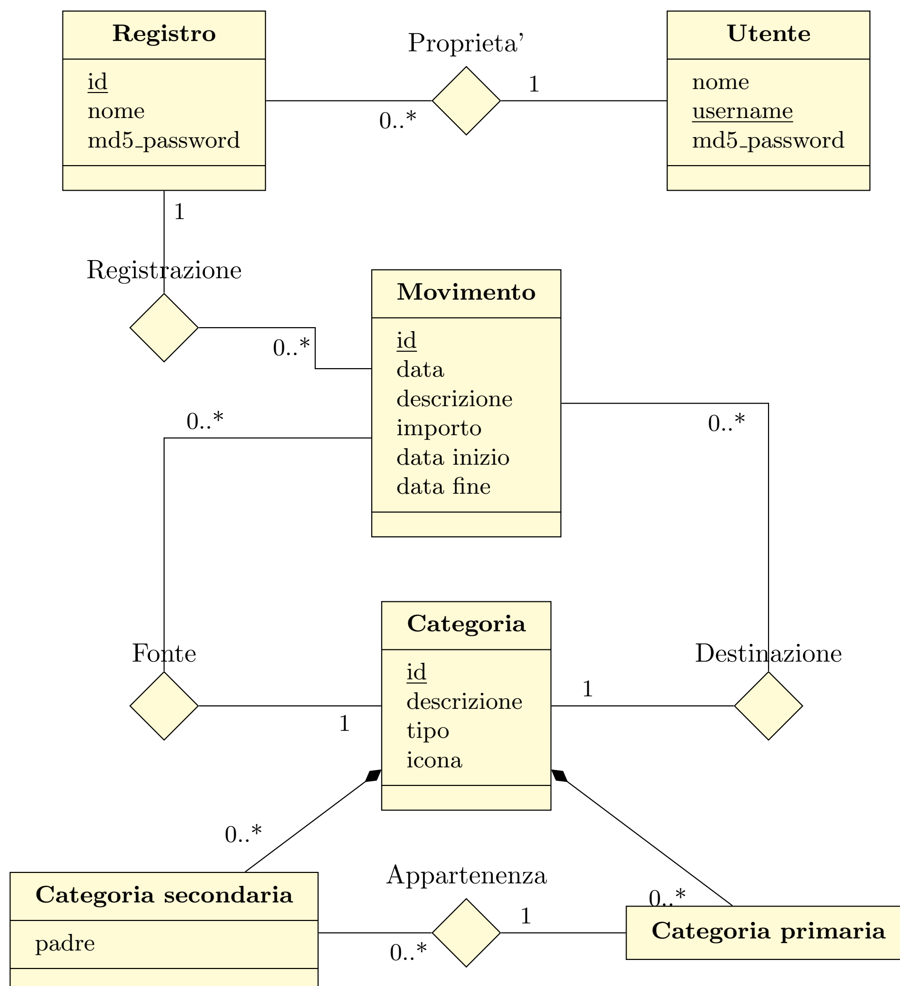

Figura 4.20: Diagramma del database dell'applicazione *Android*.

4.5 DatabaseController

Le API di *Android* forniscono l'interfaccia per creare, aggiornare o eliminare i database *SQLite* attraverso la classe `SQLiteOpenHelper`. Estendendo questa classe astratta si può definire la struttura della base di dati dell'applicazione.

4.6 DatabaseInfoController

Se la classe `DatabaseController` si occupa di gestire il singolo database (o registro), mentre la classe `DatabaseInfoController` si occupa di gestire il database che contiene i *metadato* sui registri.

Campo	Tipo	Riferimento	Note
Id	Intero	Chiave primaria	
Id_remoto		Intero	Identifica il file nel momento in cui viene caricato sul server.
Nome	Testo		Nome del file in locale.
MD5_password	Testo(32)		Stringa *hash* della password con cui è protetto il file.
Proprietario	Testo	Utenti(username)	Username dell'utente creatore del file.

Tabella 4.2: Tabella `Registri`.

Campo	Tipo	Riferimento
Id	Intero	Chiave primaria
Data	Data	
Descrizione	Testo	
Categoria Fonte	Intero	Categorie(Id)
Categoria Destinazione	Intero	Categorie(Id)
Importo	Double	
Data inizio	Data	
Data fine	Data	

Tabella 4.3: Tabella `Movimenti`.

4.7 Model

Le classi del pacchetto `models` sono classi che servono per manipolare i dati che devono essere spostati tra livello di presentazione e database. Ogni *model* è una classe composta da variabili d'istanza private che descrivono l'oggetto ed ha due tipi di metodi:

- metodi *setter*: sono metodi che assegnano valori, tramite uno o più parametri attuali, alle variabili d'istanza;

- metodi *getter*: questi metodi vengono utilizzati per ottenere il valore, cioè accedere, alle variabili d'istanza.

Le variabili d'istanza private, insieme ai *getter* e *setter*, implementato il principio di **incapsulamento** tipico della programmazione ad oggetti.

Le classi del pacchetto `models` sono quelle illustrate in figura 4.22: dove c'è un metodo `setVar(type v)` e un metodo `type getVar()` per ogni variabile `var` di tipo `type`.

Campo	**Tipo**	**Riferimento**	**Note**
Id	Intero	Chiave primaria	
Descrizione	Testo		
Padre	Intero	Categorie(Id)	
Tipo	Intero		Tre valori ammessi: **-1** per indicare una categoria di spesa (destinazione); **0** per indicare una categoria di spesa (fonte o destinazione) o **1** per indicare una categoria di entrata (fonte).

Tabella 4.4: Tabella `Categorie`.

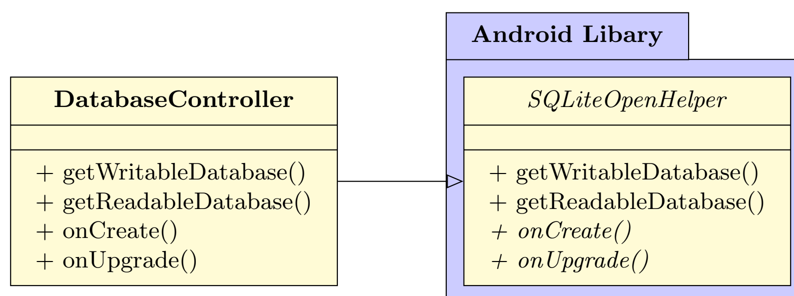

Figura 4.21: Diagramma delle classi di `DatabaseController`.

4.8 Data Access Object

Il *DAO* è un pattern che permette di costruire un nuovo livello nell'architettura dell'applicazione che si occupa di recuperare i dati memorizzati nelle tabelle del *Database Management System (DBMS)* e manipolarli secondo le regole dei livelli superiori. I metodi pubblici tipici di una classe *D.A.O.* sono:

- `getAll()`: seleziona tutte le righe senza inserire alcuna condizione della clausola `where` della query SQL, trasforma i dati che SQLite restituisce sotto forma di `Cursor` in una lista di `model` e li ritorna al chiamante;
- `getOne(id)`: effettua una query con l'obiettivo di trovare l'unico record con l'identificativo `id` e lo ritorna tramite un `model`;
- `insert(model)`: inserisce un nuovo record nella tabella utilizzando i dati passati con l'oggetto `model` (in questo caso la variabile `model.id` deve essere `null` e qualora non fosse il suo valore verrebbe ignorato);
- `update(model)`: aggiorna il record della tabella identificato da `model.id`;
- `delete(id)`: elimina il record della tabella identificato dalla variabile `id`;

Come già accennato in fase di presentazione, *Denario* offre la possibilità di creare più registri, cioè più database. Chiaramente, lo schema descritto sopra non è

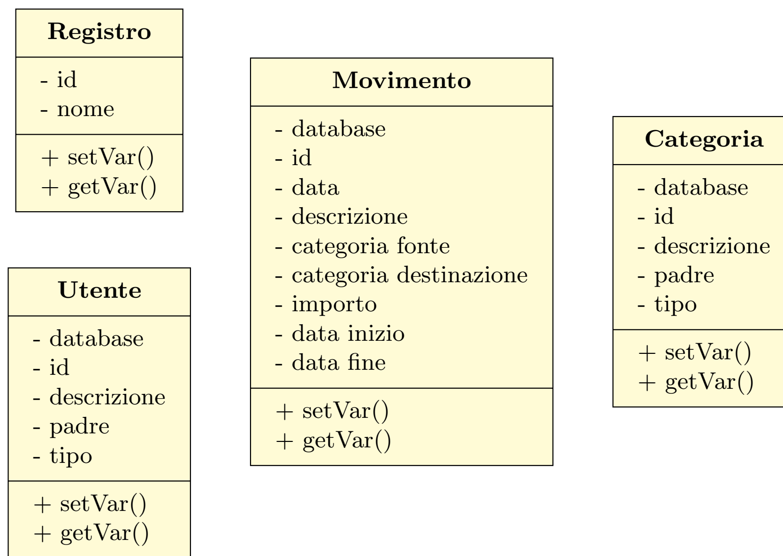

Figura 4.22: Diagramma delle classi del pacchetto `models`.

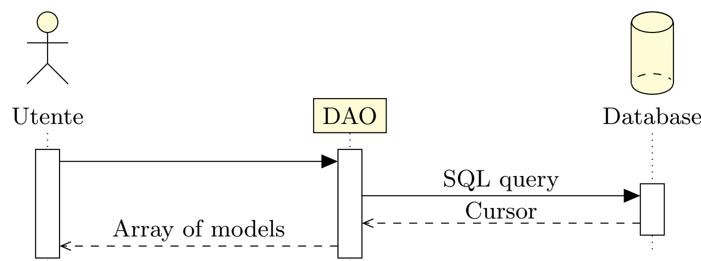

Figura 4.23: Diagramma di sequenza di un *D.A.O.*

più sufficiente in quanto vi è una sola base di dati. In questo caso interviene un nuovo tipo di oggetto definito *MDAO*.

Multiple Data Access Object

Dal punto di vista logico è un'aggregazione di uno o più *DAO*. Ad ogni classe `DaoModel` corrisponde una classe `MdaoModel` che presenta gli stessi metodi implementati in `DaoModel`, con la differenza che ogni metodo `MdaoModel.method()` effettua una chiamata a `DaoModel.method()`. Possono allora essere individuati due tipi di implementazioni diverse:

- metodi *a database singolo*: lo scopo è trovare un singolo record di un database il cui nome o id è passato come parametro. In questo caso si può anche far a meno dell'oggetto `MdaoModel` a favore dell'oggetto `DaoModel`, considerato che sappiamo già qual'è il nostro database di riferimento.

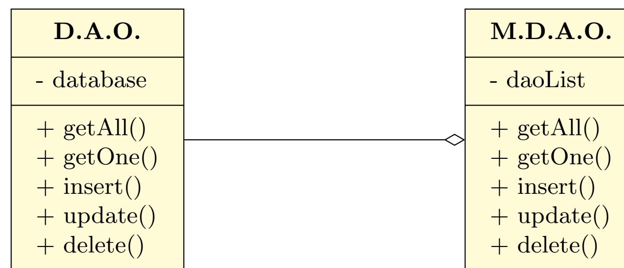

Figura 4.24: Diagramma delle classi di un *D.A.O.* e un *M.D.A.O.*

```
public DaoMovements getByDatabase(String databaseName){
   DaoMovements target = null;
   for (int i = 0; i<mDaoMovements.size(); i++){
      String name = mDaoMovements.get(i).getDatabaseName()
      if (name.equals(databaseName)){
         target = mDaoMovements.get(i);
      }
   }
   return target;
}
```

- metodi *a database multipli*: in questo caso, quando abbiamo bisogno di effettuare una "visita" su tutti i database, dell'oggetto `MdaoModel` diventa davvero utile. É questo il caso del metodo `getAll()`: mentre prima si poteva ottenere la lista degli elementi limitata al singolo database, adesso si può ottenere una lista di tutti gli elementi (del tipo `Model`) di tutti i database.

```
public List<Movement> getAll() {
   List<Movement> movements = new ArrayList<Movement>();
   for (DaoMovements daoMovements : mDaoMovements){
      movements.addAll(daoMovements.getAll());
   }
   return SortUtils.sortListByDate(movements, Const.ORDER_ASC);
}
```

4.9 Adapter

Le classi del pacchetto `adapters` implementano uno dei design pattern più importanti della programmazione ad oggetti, nonché uno dei pattern *Gang of Four (GOF)*: *adapter*.

In questa applicazione viene utilizzato per realizzare le liste (movimenti, report, ecc.) partendo da una array che contiene i dati da mostrare all'utente. I dati dell'array, provenienti dal database, sono contenuti in oggetti del pacchetto `models`. Per realizzare gli adapter, sono state create delle classi che estendono la classe `ArrayAdapter<T>`.

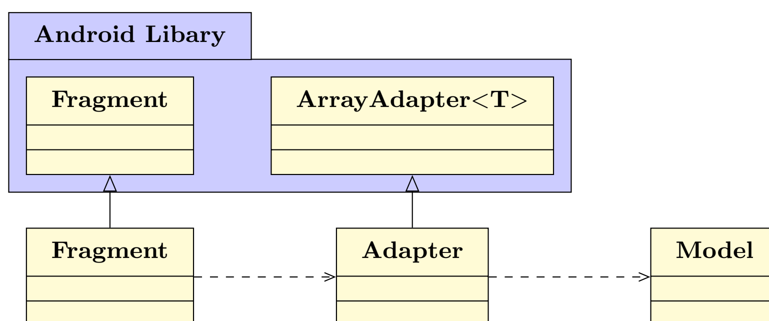

Figura 4.25: Diagramma delle classi di un *adapter* generico.

4.10 Controller

Lo scopo dei *controller* è fornire un livello di astrazione in più da utilizzare quando si accede ad una specifica risorsa, che sia una base di dati, un elenco di preferenze oppure delle *API* per effettuare richieste ad un server.

Gli esempi appena fatti mostrano esattamente il caso in cui l'applicazione utilizza delle classi di questo tipo.

4.10.1 FiltersController

I filtri che operano sui movimenti non sono altro che coppie `<tag, valore>` a cui si accede utilizzando le funzioni offerte dalla classe `SharedPreferences` della libreria *Android*.

Una volta che questa classe inserisce o modifica i parametri, questi sono salvati nel file `FilterPreferences.xml` e possono essere raccolti in più gruppi: filtri dei registri, filtri delle categorie di fonte e di destinazione, filtri sul periodo e filtri sul range di importo.

Inoltre, si divido per la loro funzione secondo quanto segue (ad ogni punto è indicato il nome del tag).

Parametri booleani Servono all'attivazione o alla disattivazione di un gruppo di filtri.

- `active`: variabile che indica se i filtri sono attivi in senso assoluto. Questa variabile contiene il valore `false` se nel menù della schermata della lista

dei movimenti viene visualizzata l'icona , true se invece viene mostrata l'icona .

- active_databases, active_src_categories, active_dst_categories, insieme ad active_date e active_amount: contengono le informazioni sull'attivazione del filtraggio, rispettivamente, sui registri, sulle categorie di fonte, sulle categorie di destinazione, sulla data e sull'importo.

Parametri su stringhe Sono start_date ed end_date, ovvero gli estremi che definiscono il filtro sul periodo. Contengono informazioni di tipo String.

Parametri su stringhe Contengono informazioni di tipo Set<String> e sono tutti quelli per i quali sono definiti degli elenchi di voci, cioè databases, src_categories e dst_categories.

Parametri su numeri a virgola mobile Contengono informazioni di tipo float e sono start_amount e end_amount, ovvero gli estremi che definiscono il filtro sull'importo.

Per ogni tipo di filtro (registri, categorie, ecc..) questa classe mette a disposizione cinque metodi:

```
// Controlla se il filtro e' attivo oppure no
public boolean isEnabled();
// Attiva il filtro
public void enable();
// Disattiva il filtro
public void disable();
// Imposta i dati sul filtro
public void setData(Type data);
// Ritorna i dati sul filtro
public Type getData();
```

Il diagramma *UML* è rappresentato in figura 4.16.

4.10.2 UserPrefsController

Per non rendere sempre necessaria una connessione al server in alcune circostanze, per esempio per l'operazione di login, questa classe, che utilizza le SharedPreferences, si occupa di memorizzare le informazioni dell'utente attualmente loggato.

Di seguito sono elencate le informazioni che gestisce.

- Login attivo, ossia una variabile booleana che indica se c'è un utente attualmente loggato nell'applicazione. Senza questa, ogni volta che l'utente apre Denario dovrebbe effettuare il login.

- Nome completo dell'utente che viene inserito in fase di registrazione e che compare in altro sul pannello del menù principale dell'applicazione.

- Nome utente (username) utilizzato in fase di login nonché identificativo univoco dell'utente.

- Stringa *hash MD5* della password, anch'essa utilizzata durante il login.

  Gli ultimi due dati vengono utilizzati per effettuare un login senza connessione. Ovviamente devono essere prima inizializzati grazie ad un login con connessione al server remoto.

4.10.3 ServerController

Gli oggetti di questa classe vengono istanziati ogni volta che l'applicazione deve effettuare una connessione al server remoto. Ogni metodo poi, per effettuare una richiesta, utilizza una delle classi del pacchetto `asynctasks` dipendentemente dal tipo di azione effettuata dall'utente.

Inoltre, vi sono anche alcuni metodi che utilizzano le `SharedPreferences` per salvare dati sull'utente, quali il nome e la stringa MD5 della password.

4.11 Asynchronous Task

Fanno parte del pacchetto `asynctasks` tutte quelle classi che effettuano richieste al server utilizzando la libreria `Volley` di Android e quelle classi che effettuano computazioni complicate che potrebbero compromettere il corretto funzionamento del `UIThread`, ossia quel *thread* che si occupa di processare i componenti dell'interfaccia utente.

Di seguito sono esaminate le classi del pacchetto.

4.11.1 La classe AsyncTask

Tutte le classi di questo pacchetto estendono la classe `AsyncTask<Params, Progress, Result>` della libreria *Android*.

Effettuare una connessione *HTTPS* può richiedere alcuni secondi di tempo. Questa operazione quindi blocca l'applicazione fino a quando non viene ricevuta una risposta dal server. Un modo per evitare questo problema è creare un *task* asincrono, così facendo viene creato un *thread* esterno a quello che gestisce l'interfaccia utente, che a questo punto non viene bloccata più.

Nell'applicazione *Android* in esame le connessioni al server sono tutte descritte da un *dialog* che aggiorna costantemente l'utente sullo stato.

4.11.2 LoginAsyncTask

Questo *task*, come da nome, si occupa di inviare una richiesta di autenticazione al server a `site1709.web.cs.unibo.it/login.php`.

Dati richiesti: nome utente (username) e password cifrata.

4.11.3 RegisterAsyncTask

Per inviare una richiesta di registrazione l'applicazione effettua una connessione a `site1709.web.cs.unibo.it/register.php`.

Dati richiesti: nome completo dell'utente (fullname), nome utente (username) e password cifrata.

4.12 Builder

Uno dei pattern della *GOF* prende il nome di *builder* e ha lo scopo di separare la logica di costruzione di un oggetto da tutto il resto della classe. In questo caso, la costruzioni dei backup è compito della classe `BackupBuilder`, mentre per i report esiste la classe `ReportBuilder` che fa uso della libreria *iTextPDF* (4.15.2).

4.13 Receiver

In ambiente *Android* i *receiver* sono classi speciali in grado di notificare l'applicazione al verificarsi di un determinato evento si verifica, per esempio l'arrivo di una chiamata oppure il collegamento del dispositivo alla corrente elettrica.

Il sistema di notifiche dell'applicazione utilizza questo metodo per controllare se nei giorni prossimi alla data attuale vi sono movimenti imminenti. Di questo si occupa la classe `BootUpReceiver` che, estendendo la classe `BroadcastReceiver`, invia una notifica all'utente (se necessario) quando la fase di boot del dispositivo è completata.

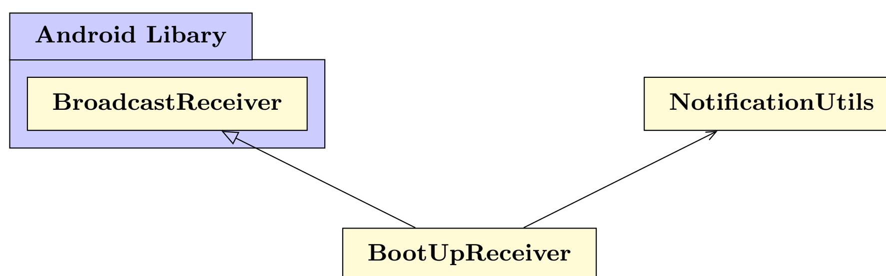

Figura 4.26: Diagramma delle classi di `BootUpReceiver`.

4.14 Utilità

Durante lo sviluppo dell'applicazione è stato molto ricorrente imbattersi in metodi tra cui era necessario instaurare una comunicazione (per esempio quando un metodo effettua la chiamata ad un altro metodo) ma uno dei due disponeva di dati di tipi diversi da quelli dei parametri formali dell'altro. In questi casi sono necessarie piccole funzioni di conversione che è bene raggruppare in classi dedicate proprio per la loro frequente necessità nell'intero progetto.

Le classi del pacchetto `utils` assolvono esattamente questo compito: ad esempio, `StringUtils` fornisce delle funzioni di conversione per il tipo di dato `String` ad altri tipi, così come `DateUtils` e `NumberUtils`.

Una peculiarità dei metodi di queste classi è che sono quasi tutti metodi statici (dichiarati con il comando `static`) e quindi non necessitano della creazione di un oggetto per essere istanziati.

4.15 Librerie esterne

Per lo sviluppo di Denario sono state utilizzate librerie esterne. Tre sono le principali, HelloCharts, iTextPDF e VolleyPlus, mentre altre sono state incluse per ragioni di retrocompatibilità delle *API*.

4.15.1 HelloCharts

HelloCharts è una libreria sviluppata appositamente per android e disponibile su Github all'indirizzo [5], dove sono presenti anche i codici sorgente di un progetto che illustra tutti i grafici costruibili con queste API, oppure sulla repository Maven all'indirizzo [6]. Offre una ampia gamma di tipi di grafici (istogrammi, grafici a torta, ecc...) molto curati graficamente.

Esempi di applicazione di questa libreria sono il grafico a torta nella schermata home (figura 4.5) ed il grafico che mostra la previsione del prossimo movimento per ogni categoria.

4.15.2 iTextPDF

Questa libreria è utilizzata per generare i report in *PDF* ed è disponibile all'indirizzo [7] (*Maven* repository) e tra i metodi offerti troviamo:

- `new Document()` per creare un nuovo documento;
- `new Font()` per configurare un nuovo font;
- `new Paragraph()` per costruire un nuovo paragrafo nel documento;
- `new PdfPTable()` per istanziare una nuova tabella;

- `PdfPTable.addCell()` per aggiungere una cella ad una tabella;
- `Document.add()` per aggiungere oggetti al documento;

Per approfondimenti si rimanda alla documentazione ufficiale [3].

4.15.3 VolleyPlus

La piattaforma *Android* mette a disposizione la libreria `Volley` per implementare richieste *HTTPS*. Tuttavia, considerando che l'applicazione deve effettuare richieste *POST* con file in allegato, le interfacce di Volley diventavano molto complicate.

Si è deciso così di utilizzare la libreria `VolleyPlus` (disponibile su Github [15]) che mette a disposizione la classe `SimpleMultiPartRequest` e i metodi `addStringParam()` e `addFile()`, che permettono di rendere il codice più semplice.

4.16 Testing

La fase di testing dell'applicazione è andata di pari passo con lo sviluppo del codice ed utilizzando un approccio che si ispira ai *metodi agili* dello sviluppo del software.

Le funzioni basilari, come la gestione dei movimenti e delle categorie, hanno subito numerosi controlli e revisioni, pertanto sono le caratteristiche che possiedono il codice più controllato.

Di contro, le funzioni implementate per ultime, come la gestione dei registri ed i *task* che comunicano con il server, avrebbero bisogno di ulteriori controlli che assicurino la loro corretta funzionalità.

Capitolo 5

Implementazione del server

Il server con il quale l'applicazione Denario comunica è stato implementato utilizzando il linguaggio *PHP* ed il suo scopo è di fornire delle *API* in grado di soddisfare alcune funzioni dell'applicazione.

Gli script del server forniscono una via di comunicazione con il database *MySQL*, nel quale risiedono i dati sugli utenti registrati e i *metadati* che riguardano i file sincronizzati. Questi ultimi, invece, vengono salvati in una cartella remota del server.

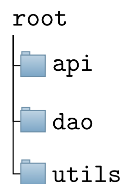

Figura 5.1: Gerarchia delle cartelle del server.

5.1 Database

Il database *MySQL* risiedente nel server presenta uno schema logico composto dall'entità *Utente* e dall'entità *Database*, collegate tra loro da una relazione *Permesso* che identifica la possibilità di un determinato utente di accedere ad un database salvato sul server. Per quanto concerne le molteplicità della relazione, un utente può non avere accesso a nessun database (si consideri il caso in cui l'utente non effettua alcuna sincronizzazione dei registri), mentre un database deve poter essere acceduto da almeno un utente, ossia il creatore. Lo schema fisico è composto quindi da tre tabelle: la tabella *Utenti* creata dall'entità *Utente*, la tabella *T_databases* creata dall'entità *Registro* e la tabella *Permessi* che traduce la relazione molti-a-molti tra le due entità in gioco.

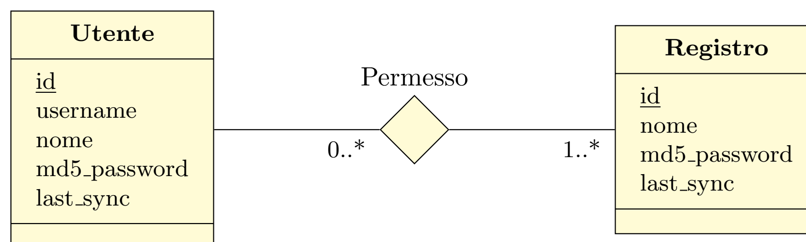

Figura 5.2: Diagramma ER del database del server.

Campo	Tipo	Riferimento	Note
Id	Intero	Chiave primaria	
Username	Testo(20)		20 caratteri sono un limite sufficiente.
Name	Testo(20)		
Md5_password	Testo(32)		Una string di hash dell'algoritmo MD5 deve essere di 32 byte.
Last_sync	Timestamp		Indica il momento di ultima modifica per l'archivio con i file personali caricato in remoto.

Tabella 5.1: Tabella `Utenti`.

5.2 API

Le API fornite dal server richiedono alcuni parametri che devono essere passati tramite una richiesta HTTP POST da parte del chiamante. Ogni script ritorna come risultato un JSON della forma:

```
{
   "response_id": RESPONSE_ID,
   // Altri parametri ausiliari
}
```

dove `RESPONSE_ID` è un intero che può essere uguale ad una delle seguenti costanti:

```
// Success responses
const RESPONSE_LOGIN_OK = 0;
const RESPONSE_REGISTRATION_OK = 1;
const RESPONSE_UPLOAD_OK = 2;
const RESPONSE_DOWNLOAD_OK = 3;
const RESPONSE_DB_SYNCHRONIZED = 4;
const RESPONSE_LOCAL_DB_OUT_OF_DATE = 5;
const RESPONSE_REMOTE_DB_OUT_OF_DATE = 6;
// Communication error responses
```

Campo	**Tipo**	**Riferimento**	**Note**
Id	Intero	Chiave primaria	
Name	Testo		
Md5_password	Testo(32)		Una string di hash dell'algoritmo MD5 deve essere di 32 byte.
Last_sync	Timestamp		Indica il momento di ultima modifica per il file caricato in remoto.

Tabella 5.2: Tabella `T_databases`.

Campo	**Tipo**	**Riferimento**	**Note**
Id_database	Intero	Utenti(id)	La chiave primaria è formata dalla coppia (Id_database, Id_user).
Id_user	Intero	T_databases(id)	

Tabella 5.3: Tabella `Permessi`.

```
const RESPONSE_WRONG_PARAMETERS = -1;
const RESPONSE_WRONG_CREDENTIALS = -2;
// Server error responses
const RESPONSE_ERROR_UPLOADING_FILE = -3;
const RESPONSE_REMOTE_DB_NOT_FOUND = -4;
const RESPONSE_USERNAME_ALREADY_EXISTS = -5;
const RESPONSE_QUERY_ERROR = -6;
```

Per prima cosa ogni script controlla la correttezza della richiesta, ovvero se tutti i parametri sono stati passati come la definizione richiede e successivamente effettua un'autenticazione dell'utente (ovviamente questo non vale per la fase di registrazione). Se i parametri non sono stati passati correttamente, ogni script termina immediatamente restituendo il valore `-1 (RESPONSE_WRONG_PARAMETERS)`.

Invece, se fallisce l'autenticazione dell'utente viene restituito il valore `-2` `(RESPONSE_WRONG_CREDENTIALS)`. Di seguito vengono elencati tutti gli script.

5.2.1 Login

Quando l'applicazione *Android* effettua una richiesta di login instaura una connessione con la pagina `site1709.web.cs.unibo.it/login.php`.

Parametri richiesti

- **username**: username che l'utente inserisce nella schermata di login (figura 4.1).
- **md5_password**: *hash* dell'algoritmo *MD5* della password dell'utente.

Funzionamento Molto semplicemente, lo script effettua una query al database recuperando il record della tabella che contiene i parametri in input: se il record esiste ritorna un JSON della forma:

```
{
  "response_id": 0, // RESPONSE_LOGIN_OK
  "name": "Lorenzo" // Nome utente scelto in fase di registrazione
}
```

mentre in caso di fallimento, ovvero se il record non viene trovato viene ritornato `-2 (RESPONSE_WRONG_CREDENTIALS)`.

5.2.2 Registrazione

Quando l'utente preme il pulsante **Registrati** nella schermata di registrazione (figura 4.2), l'applicazione effettua una richiesta all'indirizzo `site1709.web.cs.unibo.it/register.php`.

Parametri richiesti

- **fullname**: nome completo che l'utente inserisce nella schermata di registrazione (figura 4.2).
- **username**: username che l'utente inserisce nella schermata di registrazione.
- **md5_password**: *hash MD5* della password dell'utente.

Funzionamento Dopo aver controllato che siano stati passati tutti i parametri necessari, lo script controlla che non ci sia già un altro utente con lo stesso username di quello passato come parametro (l'username è l'identificativo univoco per gli utenti). Se questa query non ritorna zero risultati, allora si procede con l'inserimento del nuovo utente nel database.

```
{
  "response_id": RESPONSE_ID
}
```

`RESPONSE_ID` è un valore che può essere:

- `1 (RESPONSE_REGISTRATION_OK)` se non si verifica nessun errore e l'esecuzione viene completata correttamente.
- `-5 (RESPONSE_USERNAME_ALREADY_EXISTS)` se esiste già un utente con username uguale a quello scelto dall'utente che sta per registrarsi.

- `-6 (RESPONSE_QUERY_ERROR)` se fallisce la query per inserire i dati nel database.

5.2.3 Sincronizzazione dei registri

La sincronizzazione dei registri, ovvero dei database su cui sono memorizzati i movimenti dell'utente, avviene in due fasi. In una prima fase di "confronto" vengono comparati il file locale ed il file in remoto e nella seconda fase, sulla base dell'esito della prima, possono verificarsi una delle seguenti azioni:

- se i tempi di ultima modifica dei due file coincidono, l'operazione è terminata;
- se il file in locale ha una data di ultima modifica più recente, quest'ultimo viene caricato sul server andando a sovrascrivere il vecchio file remoto;
- se, viceversa, è il file in remoto ad essere più recente (cosa che può avvenire nel caso che si abbia un database condiviso), questo viene scaricato sul *device* dell' utente sovrascrivendo il file in locale.

5.2.3.1 Confronto

L'applicazione effettua una richiesta all'indirizzo `site1709.web.cs.unibo.it/compare_db.php`.

Parametri richiesti

- **username**: username che l'utente inserisce nella schermata di registrazione (figura 4.2).
- **md5_user_password**: *hash* dell'algoritmo *MD5* della password dell'utente.
- **file_id**: intero identificativo del registro sul database del server.
- **file_time**: istante di ultima modifica del file in locale (in formato UNIX timestamp).
- **md5_file_password**: *hash MD5* della password del registro.

Funzionamento Previo controllo dei parametri e delle credenziali dell'utente, viene interrogato il database del server richiedendo il record del registro con identificativo `file_id`, i cui dati vengono confrontati con i parametri della richiesta *HTTPS POST*.

La risposta quindi è della forma seguente:

```
{
  "response_id": RESPONSE_ID
}
```

con `RESPONSE_ID` che può essere:

- `4 (RESPONSE_DB_SYNCHRONIZED)` se il file in locale e il file in remoto hanno lo stesso istante di ultima modifica.
- `5 (RESPONSE_LOCAL_DB_OUT_OF_DATE)` se sul server è disponibile una copia più recente di quella in locale.
- `6 (RESPONSE_REMOTE_DB_OUT_OF_DATE)` se la copia sul server è obsoleta.
- `-4 (RESPONSE_REMOTE_DB_NOT_FOUND)` se la query non restituisce alcun record.

5.2.3.2 Upload del database sul server

L'applicazione effettua una richiesta all'indirizzo `site1709.web.cs.unibo.it/upload_db.php`.

Parametri richiesti

- **username**: username che l'utente inserisce nella schermata di registrazione (figura 4.2).
- **md5_user_password**: *hash MD5* della password dell'utente.
- **id**: (opzionale) identificativo del registro che viene passato solo in modalità "aggiornamento".
- **filename**: nome del registro.
- **file_time**: istante di ultima modifica del file in locale (in formato UNIX timestamp).
- **md5_file_password**: *hash* dell'algoritmo *MD5* della password del registro.
- **database**: file vero e proprio che deve essere caricato in remoto.

Funzionamento Questo script dispone di due modalità: creazione ed aggiornamento. La modalità viene definita in base al passaggio della variabile **id**, se questa è impostata, lo script esegue in modalità "aggiornamento", altrimenti in modalità "creazione".

In modalità "creazione" il server provvede a fare una semplice inserzione nella tabella dei registri (tabella 5.2) e nella tabella dei permessi (tabella 5.3).

In modalità "aggiornamento" viene prima effettuata una query per recuperare il record del registro e successivamente viene aggiornato. Per concludere, in entrambe le modalità viene caricato il nuovo file nella cartella `databases` del server, con il nome del file della forma `[ID_REMOTO].db`.

Se non si verificano error, la risposta restituita dallo script assume la seguente forma:

```
{
  "response_id": 3, // RESPONSE_UPLOAD_OK
  "record_id": ID_RECORD,
  "time": "yyyy-mm-dd hh:mm:ss"
}
```

mentre se viene riscontrato qualche errore, viene stampata la consueta risposta:

```
{
   "response_id": RESPONSE_ID
}
```

con `RESPONSE_ID` che può essere:

- `-3 (RESPONSE_ERROR_UPLOADING_FILE)` se si verifica un errore durante il caricamento del file.
- `-4 (RESPONSE_REMOTE_DB_NOT_FOUND)` se non viene trovato, in modalità "aggiornamento" nessun record corrispondente all'identificativo.

5.2.3.3 Download del database sul device

L'applicazione effettua una richiesta all'indirizzo `site1709.web.cs.unibo.it/download_db.php`.

Parametri richiesti

- **username**: username dell'utente.
- **md5_user_password**: *hash MD5* della password dell'utente.
- **file_id**: (opzionale) identificativo del registro.

Funzionamento Lo scopo dello script è comunicare al client la *Uniform Resource Locator (URL)* per scaricare il file del registro e quindi, dopo aver effettuato una *query* per cercare l'id nel database remoto, viene restituito un *JSON* così fatto:

```
{
  "response_id": 3, // RESPONSE_DOWNLOAD_OK
  "url":
      "https://lorenzovainigli.altervista.org/denario/databases/[ID_FILE].db"
}
```

In caso di errore invece si ha:

```
{
  "response_id": -4, // RESPONSE_REMOTE_DB_NOT_FOUND
}
```

5.3 Testing

Gli script che compongono il server sono abbastanza semplici e non sono numerosi. In ogni caso, durante la loro fase di testing non sono stati riscontrati *bug*.

Capitolo 6

Conclusioni e sviluppi futuri

In questa tesi è stata presentata un'applicazione Android il cui scopo è di aiutare l'utente nella gestione, organizzazione e monitoraggio sia delle proprie spese, sia delle proprie entrate. Come il lettore ha potuto constatare, *Denario* offre tutte le funzioni tipiche di una buona applicazione di *budget tracking* come la divisione dei movimenti in categorie e la possibilità di effettuare filtraggi sui dati.

Dopo un'introduzione all'argomento, è stato esposto l'aspetto architetturale dell'applicazione, con particolare attenzione ai concetti sui quali si basa. Successivamente si è passati ad una vista più di basso livello, esaminando i vari componenti dell'applicazione e il loro modo di cooperare durante l'esecuzione sul dispositivo dell'utente.

È stato quindi dimostrato che, osservando programmi già affermati ed utilizzati e avvalendosi di una buona fase di progettazione, subordinata ovviamente dallo studio dell'ambiente *Android*, è stato possibile sviluppare un'applicazione in grado di migliorare la vita degli utenti nel loro quotidiano.

Sviluppi futuri

Le funzioni possedute attualmente da Denario sono quelle caratteristiche della maggior parte delle applicazioni della stessa tipologia. L'unica funzione veramente inedita sono le previsioni, di cui sarebbe interessante rendere l'algoritmo più sofisticato. Senza fermarci ad una semplice media o varianza, potremmo prendere in considerazione quali sono i giorni in cui viene registrato un movimento, se c'è una correlazione tra e potrebbe essere analizzato il loro andamento. Inoltre, potrebbe essere utile anche sapere qual'è l'incisione delle entrate o uscite lungo tutto il loro periodo di competenza. Infatti proprio la possibilità di inserire una data di inizio ed una di fine entro le quali "spalmare" il movimento non è solo un'accortezza per capire la percentuale già maturata, ma anche un

modo per calcolare quanto quel movimento incide sulla nostra gestione e se sia opportuno porre dei rimedi.

Calendario Nell'ottica di rendere più sofisticate le statistiche e le previsioni fornite dall'applicazione, nei codici sorgente è stata già implementato un metodo che, data una lista di movimenti, effettua una ripartizione dell'importo sul loro periodo e crea un "calendario" (da qui il nome) con il saldo quotidiano relativo ad ogni giorno. Grazie a questo, potranno essere calcolati totali e medie senza preoccuparsi che un movimento straordinario falsifichi il dato. Vi sono anche il modello `Calendar`, la classe `DaoCalendar` e la tabella calendar.

Un'ulteriore funzione molto interessante che possiedono le applicazioni studiate nel capitolo 2 è la possibilità di definire somme a titolo di budget o di risparmio, che permettono una pianificazione più approfondita della propria gestione monetaria.

Ringraziamenti

I miei ringraziamenti vanno, prima di tutto, al Prof. Luciano Bononi ed al Dott. Luca Bedogni che mi hanno dato la possibilità fare questo lavoro e poi devo dire grazie a tutte quelle persone che hanno contribuito, continuano e continueranno a farlo, alla creazione e all'arricchimento di tutti quei siti web e di quelle documentazioni libere che sono fonte di sapere inesauribile. Grazie a loro, imparare un linguaggio di programmazione è una cosa più facile e molto più accessibile. Volendone citare quelli che mi sono stati più di aiuto, ricordo Android Developers [2], Stack Overflow [14], RegExr [13], Material Design Icons [9] e Latex Stack Exchange [8].

Glossario

Activity Una delle schermate visibili dall'utente dell'applicazione, con il quale si può interagire. 27, 28, 32, 62

Adapter Design pattern GOF il cui scopo è far comunicare due oggetti che possiedono interfacce incompatibili. 12, 27, 42, 43

Android Sistema operativo per dispositivi mobili basato sul kernel Linux. 15, 17, 23, 25, 27, 28, 38, 43, 45, 46, 48, 51, 57, 62, 63

Applicazione Un programma o una serie di programmi in esecuzione su un elaboratore o su un dispositivo mobile. 15, 16, 62

Backup Copia dei file da effettuarsi periodicamente per prevenire la perdita di dati in caso di danneggiamento dei file originali. 18, 24

Boot (Bootstrap) termine che indica l'insieme di operazioni effettuate da un dispositivo dal momento dell'accensione al momento in cui risulta pronto per l'utilizzo. 46

Budget tracking Operazione con la quale si tracciano i movimenti di denaro. 15, 17, 21, 57

Bug Problema nel codice sorgente di un programma che dà origine ad un comportamento anomalo. 56

Builder Design pattern GOF il cui scopo è separare la costruzione di un oggetto dalla sua rappresentazione. 27, 46

Categoria Se intesa come categoria di movimenti, è un insieme degli stessi che si accomunano per la loro funzione (es. spese per l'affitto). 17

Client Componente di una rete informatica che accede ai servizi forniti dal server. 25, 37, 63

Cloud storage Modalità di salvataggio dei dati su elaboratori diversi, ma appartenenti alla stessa rete. 18

Controller Design pattern architetturale che fornisce un'interfaccia comune per la gestione delle richieste. 27, 43

DAO Il Data Access Object (DAO) è un design pattern architetturale che offre dei metodi per recuperare informazioni dal database. 28, 40

Deposito Categorie che identificano le somme di denaro di cui dispone l'utente. 17, 21, 62, 64

Destinazione Se intesa come categoria di destinazione, è la categoria in cui va a finire l'importo del movimento. 21, 64

Dialog Finestra di dialogo che può comparire sulla UI dell'applicazione. 28, 32, 45

Entrata Movimento con cui un importo viene spostato da una fonte ad un deposito. 21

Fonte Se intesa come categoria di fonte, è la categoria da cui proviene l'importo del movimento. 21, 62

Fragment Una parte di una activity, adatta al suo riuso in altre parti dell'applicazione. 28, 29, 34

Hash Funzione non iniettiva che mappa una stringa di lunghezza arbitraria in una di lunghezza predefinita. 45, 51–55, 62

Intent In ambiente Android è una descrizione astratta di un'operazione che deve essere effettuata dall'applicazione stessa oppure da un'altra. 36

Java Linguaggio di programmazione orientato agli oggetti ed utilizzato per sviluppare applicazioni Android. 27, 28

Layout Schema che in Android descrive gli elementi che compaiono nella UI, la loro disposizione e le loro proprietà. 31

Long-click Azione dell'utente che preme una zona dello schermo del dispositivo, mantenendo premuto per circa un secondo. 33

MD5 Funzione hash crittografica realizzata da R. Rivest nel 1991. 45, 51–55

MDAO Il Multiple Data Access Object (MDAO) è un design pattern architetturale simile a DAO, ma che dà la possibilità di effettuare la stessa interrogazione su più database della stessa forma, ma presenti su file distinti. 28

Metadato Informazione che descrive un insieme di dati. 38, 49

Metodo agile Metodo di sviluppo del software che si focalizza sull'obiettivo di consegnare al cliente, in tempi brevi e frequentemente, nuove versioni del prodotto. 48

Minimum API version Versione di Android minima che deve avere il dispositivo per eseguire l'applicazione. 27

Movimento Evento che si verifica quando una somma di denaro viene trasferita da un deposito ad un altro, oppure da una categoria ad un'altra. 17, 18

MySQL Tipo di DBMS relazionale. 49

PHP Linguaggio di programmazione utilizzato soprattutto nella programmazione server-side. 49

POST Metodo di HTTP per inviare dati ad una pagina web. Solitamente è utilizzato nei web form con la codifica `application/x-www-form-urlencoded`. 25, 48, 53

Query Stringa contenente i comandi per effettuare un'interrogazione ad un database. 55

Receiver In ambiente Android è un componente che resta in ascolto per uno specifico evento e, al suo verificarsi, si attiva. 28, 46

Registro Nell'accezione utilizzata in questo testo, una base di dati, definita dall'utente, che contiene informazioni sui movimenti e sulle categorie. 16, 22

Server Componente di una rete informatica che fornisce servizi agli altri componenti, detti client. 16, 25, 27, 37, 61

SQL Structured Query Language (SQL) è linguaggio per la gestione di database relazionali. 63

SQLite Versione di Structured Query Language (SQL) molto più leggera fornita dalla piattaforma Android. 37, 38

Tab Sezione dell'interfaccia dell'applicazione simile alle schede di navigazione dei browser. 28, 31

Target API version Versione di Android sulla quale è state sviluppata e testata l'applicazione e per la quale non sono previsti problemi di compatibilità. 27

Task Porzione di codice dell'applicazione eseguita in modo indipendente dal resto del programma. 45, 46, 48

Thread Sezione del processo di un programma adibito ad un preciso compito (es. UI Thread gestisce la UI). 45

Top-down Tipo di approccio che prevede prima la formulazione del concetto generale e, successivamente, la spiegazione dei dettagli. 27

Trasferimento Movimento con cui un importo viene spostato da un deposito ad un altro deposito. 22

Uscita Movimento con cui un importo viene spostato da un deposito ad una destinazione. 22

ZIP Formato utilizzato per i file compressi. 36

Acronimi

API Application Program Interface. 25, 27, 43, 47, 49, 63

DAO Data Access Object. 28, 40, 41, 62, *Glossario:* DAO

DBMS Database Management System. 40, 63, *Glossario:* DBMS

GOF Gang of Four. 42, 46, 61

HTTPS HyperText Transfer Protocol over Secure Socket Layer. 24, 25, 45, 48, 53

JSON Javascript Object Notation. 25, 50, 52, 55

MDAO Multiple Data Access Object. 28, 41, *Glossario:* MDAO

PDF Portable Document Format. 24, 32, 36

SQL Structured Query Language. 63, *Glossario:* SQL

UI User Inteface. 27, 29, 62, 64

UML Unified Modeling Language. 11, 16

URL Uniform Resource Locator. 55

Riferimenti

[1] *AndroMoney su Google Play Store.* URL: https://play.google.com/store/apps/details?id=com.kpmoney.android.

[2] *Documentazione ufficiale di Android.* URL: https://developer.android.com/.

[3] *Documentazione ufficiale di iTextPDF.* URL: http://developers.itextpdf.com/apis.

[4] *Expense IQ su Google Play Store.* URL: https://play.google.com/store/apps/details?id=com.handyapps.expenseiq.

[5] *HelloCharts su Github.* URL: https://github.com/lecho/hellocharts-android.

[6] *HelloCharts su Maven.* URL: https://mvnrepository.com/artifact/com.github.lecho/hellocharts-library.

[7] *iTextPDF su Maven.* URL: https://mvnrepository.com/artifact/com.itextpdf/itextpdf.

[8] *Latex Stack Exchange.* URL: http://tex.stackexchange.com/.

[9] *Material Design Icons.* URL: https://materialdesignicons.com/.

[10] *Monefy su Google Play Store.* URL: https://play.google.com/store/apps/details?id=com.monefy.app.lite.

[11] *Money Lover su Google Play Store.* URL: https://play.google.com/store/apps/details?id=com.bookmark.money.

[12] *Money Manager su Google Play Store.* URL: https://play.google.com/store/apps/details?id=com.realbyteapps.moneymanagerfree.

[13] *RegExr.* URL: http://regexr.com/.

[14] *Stack Overflow.* URL: http://stackoverflow.com/.

[15] *VolleyPlus su Github.* URL: https://github.com/DWorkS/VolleyPlus.