# Programmazione Avanzata (L-31)
# Relazione del Software jBudget

Daniele Serafini

Matricola: 097845

July 4, 2020

# Contenuti

# 1 Introduzione

## 1.1 Terminologia

Piccola nota sulla terminologia utilizzata all'interno della relazione:

- il termine **account** fa riferimento ai conti (es. conto bancario, conto postale, soldi contanti ecc.)
- il termine **tag** fa riferimento ad una determinata tipologia (es. sport, bollette, casa ecc.)
- il termine **budget** fa riferimento a determinati 'preventivi' sui movimenti associati alle **tag**
- il termine **asset** fa riferimento a conti di tipo positivo
- il termine **liabilities** fa riferimento a conti di tipo negativo

## 1.2 Concetti Principali

La soluzione proposta e' stata sviluppata tenendo conto dei seguenti concetti:

**Account** serve a rappresentare le operazioni che avvengono in un determinato conto, quindi determina le entrate, le uscite e il totale dei debiti o crediti che un conto puo' avere. Per l'applicativo proposto sono state considerate due tipologie di "account": gli asset che rappresentano la disponibilita di denaro e le "liabilities" che rappresentano dei debiti che devono essere consumati.

**Movement** rappresenta un'uscita o un'entrata da un account che ne gestira' il significato.

**Transaction** rappresenta un insieme di movimenti, in quanto possono essere effettuati diversi movimenti che hanno lo stesso contesto.

**Tag** rappresenta la classificazione di movimento/transazione; e' utile ai fini della generazione di statistiche che verranno effettuate per tipologia di movimento.

**Budget** rappresenta, in base ad una specifica **tag**, "l'obbiettivo" di ammontare che ci siamo prefissati

## 2 Struttura dei pacchetti

In questa sezione viene brevemente spiegata la struttura dei pacchetti che sono stati creati ai fini di garantire una corretta leggibilita‘ del codice sorgente.

Sono presenti anche dei pacchetti necessari per la gestione del server; l’idea e‘ quella di poter eseguire l’applicazione come demone, sfruttando gli ‘endpoint‘ come punto di entrata per eseguire diverse azioni e per creare diversi tipi di client. Questi pacchetti non sono descritti in questa relazione ma possono comunque servire come base per implementare la suddetta funzionalita‘.

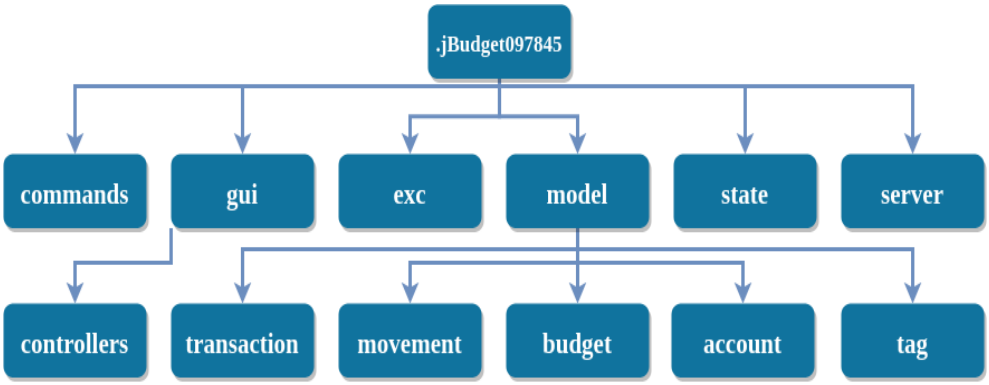


Figure 1: Packages Structure

### 2.1 Model

Nel pacchetto **model** risiedono tutte le classi necessarie per la gestione del modello dove verranno modificati, aggiunti ed eliminati i dati della nostra applicazione. All’interno di questo pacchetto sono presenti i seguenti sotto-pacchetti:

- **transaction** dove viene gestita la creazione e il modello delle transazioni e delle transazioni schedulate.
- **movement** dove viene gestita la creazione e il modello dei movimenti.
- **account** dove viene gestita la creazione e il modello degli account.
- **tag** dove viene gestita la creazione e il modello delle tag.
- **budget** dove viene gestita la creazione e il modello dei budget.

### 2.2 State

All'interno del pacchetto **state** risiedono le classi e interfacce necessarie alla gestione del salvataggio delle istanze che incorporano lo stato dell'applicazione. Le classi attualemnte salvate dall'applicazione sono quelle di tipo **Registry** e quelle di tipo **BudgetManager**.

### 2.3 Gui

All'interno del pacchetto **gui** risiedono le classi che gestiscono le viste dell'interfaccia grafica. E' presente un sotto-pacchetto **controllers** contenente i controller necessari per la gestione delle viste.

### 2.4 Exc

All'interno del pacchetto **exc** risiedono le classi personallizate per la gestione delle eccezioni che riguardano classi specifiche dell'applicazione.

## 3 Astrazione Modello

In questa sezione verranno descritte le interfacce e le enumerazioni che sono state poi implementate dalla classi che costituiscono il modello del nostro applicativo. Sono state create delle interfacce al fine di garantire una buona astrazione dell'architettura del modello, aumentando la modularita' delle classi che andranno poi implementate e lo sviluppo di classi future che estendera' le funzionalita' del modello stesso.

### 3.1 Interfacce

Sono state implementate le seguenti interfacce di base per la costruzione del modello di cui in seguito vedremo l'implementazione:

- **Registry** e' implementata dalle classi che hanno la responsabilita' di gestire tutti i dati dell'applicazione (transazioni, account, tags e transazioni programmate) permettendo la creazione, modifica e cancellazione di essi.
- **Tag** e' implementata dalle classi che definisco una spesa o guadagno
- **Movement** e' implementata dalle classi che hanno la responsabilita' di gestire uno specifico movimento. Permette accesso e modifica dei suoi attributi principali, ovvero, importo, descrizione, account, tag associati e tipo di movimento (definito dall'enumerazione **MovementType**). Il movimento e' associato ad una transazione dalla quale prende la data.

- **Transaction** e‘ implementata dalle classi il quale compito e‘ gestire una transazione. Permette accesso e modifica dei suoi attributi principali, ovvero, data, movimento e lista dei tag e l’ammontare totale della transazione (con la variazione dei movimenti). Quando viene aggiunto/eliminato un movimento devono essere aggiunti/eliminati tutti i tag associati ad esso.

- **ScheduledTransaction** e‘ implementata dalle classi che hanno la responsabilita‘ di gestire una o piu‘ transazioni programmate per una certa data.

- **Account** e‘ implementata dalle classi che che hanno la responsabilita‘ di gestire un determinato account. Permette accesso e modifica dei suoi attributi principali, ovvero, nome, descrizione, saldo iniziale, tipo di account (definito dall’enumerazione **AccountType**) e la lista dei movimenti effettuati (quindi ha una relazione bidirezionale con i movimenti).

- **Budget** e‘ implementata dalle classi che hanno la responsabilita‘ gestire un budget. Ad ogni budget associa ad ogni tag un importo che indica il target per quella spesa/guadagno.

- **BudgetReport** e‘ implementata dalle classi che hanno la responsabilita‘ di generare un report, che mostra gli scostamenti avvenuti su ogni budget.

- **BudgetManager** e‘ implementata dalle classi che hanno la responsabilita‘ di costruire oggetti che implementano l’interfaccia **BudgetReport**, in base ad un **Budget** e ad un **Registry**.

### 3.2 Enumerazioni

Sono state create le seguenti enumerazioni per gestire dei comportamenti da compiere in determinate situazioni:

- **MovementType** [DEBIT — CREDIT] definisce il tipo di movimento (utile nel momento in cui deve essere aggiunto un movimento in un account, in quanto il comportamento di un movimento viene deciso appunto dalle classi che implementano **Account**).

- **AccountType** [ASSET — LIABILITIES] definisce il tipo di account.

# 4 Implementazione

L'architettura scelta per la realizzazione del progetto e‘ il MVC (Model Control View); in questa sezione viene spiegata: l'implementazione del modello, secondo l'astrazione della sezione precedente; come e' stato gestito lo stato dell'applicazione, ovvero in che modo vengono salvati/caricati i dati all'interno dell'applicazione; l'implementazione dell'interfaccia grafica e dei relativi componenti. Per la descrizione dei metodi e‘ possibile generare le Javadoc.

## 4.1 Librerie di Terze Parti

Nel progetto sono state utilizzate librerie di terze parti; principalmente per il salvataggio dello stato dell'applicazione e per l'interfaccia grafica.

Le librerie utilizzate sono le seguenti:

- **javafx** per la creazione dell'interfaccia grafica.
- **jackson** per la serializzazione dello stato del modello in JSON.
- **javatuples** per la creazione delle tuple.

## 4.2 Modello

Il modello e' stato diviso in due categorie principali, ovvero, la prima riguarda tutte le operazioni base per la gestione degli account, transazioni e tag; la seconda invece riguarda le operazioni sui budget, in questo caso la creazione di nuovi budget e la generazione di report.

Le classi della prima categoria sono state implementate come segue:

- **Ledger** e' la classe Singleton che implementa /'interfaccia**Registry**, qui vengono salvate (quasi) tutte le informazioni che poi verranno serializzate; inoltre, e‘ la classe utilizzata dal controller al fine di effettuare modifiche, creazioni e cancellazioni dei dati; si intendono operazioni per quanto riguarda le transazioni (quindi i movimenti), le tag, gli account.
- **GeneralTag** e‘ la classe che implementa l'interfaccia **Tag**, contiene al suo interno un nome e una descrizione.
- **CreditMovement/DebitMovement** sono le classi che implementano l'interfaccia **Movement** queste classi definiscono il comportamento nel caso in cui un movimento

sia di tipo CREDIT oppure di tipo DEBIT. I movimenti vengono inizializzati con una **Transaction** nel costruttore, dalla quale viene presa la data e salvato il riferimento.

- **GeneralTransaction** e' la classe che implementa l'interfaccia **Transaction**, dove viene definito il comportamento generale di una transazione. Una transazione puo' contenere piu' movimenti con relativi tag
- **ScheduledTransactionHandler** e' la classe che implementa l'interfaccia **ScheduledTransaction**, qui viene definito il comportamento di una transazione schedulata, la quale puo' contenere diverse **Transaction**. Tramite la variabile booleana isCompleted possiamo controllare quando le transazioni sono completamente registrate.
- **AssetAccount/LiabilitiesAccount** sono le classi che implementano l'interfaccia **Account**, ognuna definisce il comportamento dell'account in base al tipo; la differenza principale e' che se un account e' di tipo ASSET, i movimenti di tipo CREDIT vengono considerati in positivo mentre i movimenti di tipo DEBIT in negativo; negli account di tipo LIABILITIES invece avviene l'opposto.

I pacchetti contenenti le classi riguardanti transazioni, movimenti e account sono provviste di appositi classi di gestione le quali hanno il compito di gestire l'istanzazione di tali classi.

Le classi della seconda categoria invece sono:

- **BudgetHandler** e' la classe Singleton che implementa l'interfaccia **BudgetManager**; questa classe si occupa di salvare le istanze di classi che implementano l'interfaccia **Budget** e di generare oggetti di classi che implementano l'interfaccia **BudgetReport**.
- **GeneralReport** e' la classe che implementa l'interfaccia **BudgetReport**; questa classe si occupa di creare un report, prendendo in input un **Budget** e una lista di **Movement**.
- **GeneralBudget** e' la classe che implementa l'interfaccia **Budget**; si occupa di salvare per ogni tag presa in input il valore atteso.

### 4.3 Controllore

Alla radice dei pacchetti mostrati nella seconda sezione troviamo la classe **ApplicationController**. Questa classe e' responsabile di effettuare modifiche, cancellazione e aggiunta dei dati all'interno del nostro modello. E' inoltre responsabile di interfacciarsi con la **GUI**,

che appunto invia comandi al controller che girera‘ poi al modello. Il controller interagisce esclusivamente con le classi principali del modello, ovvero classi che implementano **Registry** e classi che implementano **BudgetManager**;

### 4.4 Interfaccia Grafica (View)

Per mostrare il funzionamento del software e‘ stata creata una semplice interfaccia grafica utilizzando la libreria **Javafx**.
Il funzionamento della libreria e‘ molto semplice, e‘ stata creata la classe **ScreenController** per gestire le varie scene che puo‘ assumere l’applicazione, ovvero, ogni scena e‘ stata mappata con una chiave stringa ‘nome‘ al quale e‘ associato il valore stringa che fa riferimento al file .fxml che verra‘ poi ‘renderizzato‘.
I file .fxml sono stati salvati nell’apposita cartella **resources**.
Al momento, con l’interfaccia grafica, posso essere eseguite le seguenti operazioni:

- la creazione di nuovi account.
- la creazione di nuove tag.
- la creazione di nuovi movimenti.
- la creazione di nuovi budget.
- puo‘ essere visualizzata la lista dei movimenti, con possibilita‘ di ordinazione.
- un grafico che, preso in input un range di date, mostra per ogni tag il costo delle spese effettuate, mostrando in quali categorie ci sono state piu‘ spese.

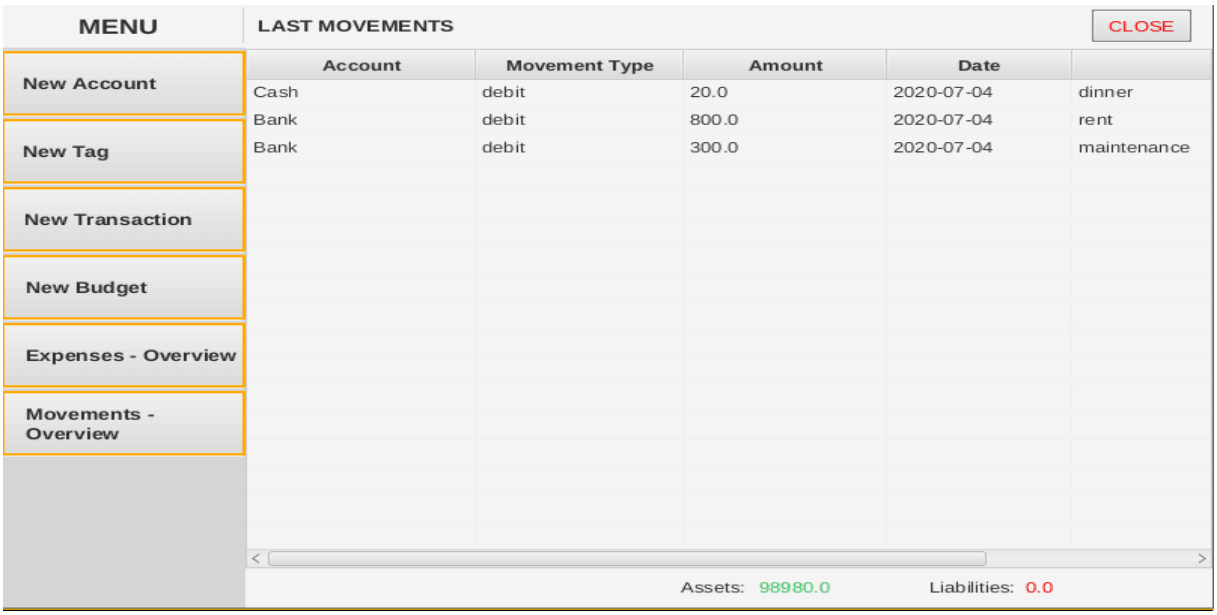


Figure 2: Menu Sample

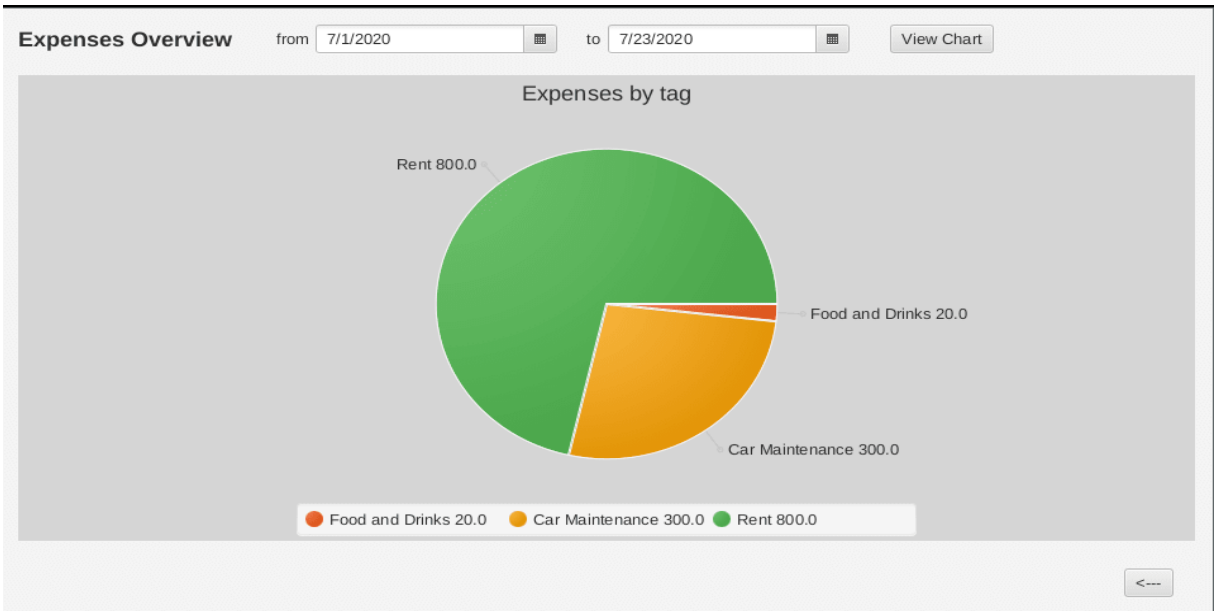


Figure 3: Graph Sample

La [Figure 2] e‘ un sample della visualizzazione del menu dove e‘ possibile vedere gli ultimi movimenti effettuati e tutte le opzioni disponibili.
La [Figure 3] e‘ un sample della visualizzazione del grafo che mostra la differenza della varie spese in base alle tag.

### 4.5 Stato

All'interno del pacchetto **state** troviamo le classi necessarie per il salvataggio dello stato dell'applicazione, quindi il salvataggio di istanze di classi che implementano **Registry** e **BudgetManager**.
La classe che viene salvata in realta‘ e‘ **StateWrapper** la quale contiene le istanze degli oggetti sopra menzionati. Quando gli oggetti vengono caricati, essendo Singleton, dispongono di un metodo apposito per impostare l'istanza che viene impostata nel momento del caricamento.
Tutte le classi che vengono serializzate dispongono di apposite annotazioni definite dalla libreria Jackson per definire in che modo devono essere serializzate/deserializzate. Tutto lo stato viene salvato in un unico JSON chiamato ‘save.json‘, in una cartella nella root principale del progetto, nella cartella ‘data‘.

## 5 Funzionalita‘

Il progetto mediante l'astrazione del modello puo‘ essere facilmente esteso, creando nuove classi che implementano diverse funzionalita‘, ad esempio, se si volesse implementare i trasferimenti nei i vari **Account** basterebbe implementare la classe **Movement** specificando il comportamento del nuovo movimento.

La creazione di nuovi controller e‘ facile in quanto lo stato e‘ incapsulato in due classi (**Registry BudgetManager**).

Essendo tutto lo stato dell'applicazione salvato in un unico JSON, e‘ possibile esportarlo ad esempio alla fine dell'anno. Inoltre puo‘ essere letto in altri linguaggi ricostruendo tutta la history del modello.

## 6 Limitazioni

In quanto gli oggetti salvati in ‘HashMap‘ non possono essere salvati per riferimento, la classe che implementa **Budget**, contenente una ‘HashMap‘ di ¡Tag,Double¿, nel momento

della serializzazione la **Tag** viene automaticamente salvata chiamando il metodo toString(). Per la deserializzazione di tali istanze e per mantenere l'uguaglianza tra le **Tag**, all'interno della classe **GeneralTag** sono stati sovrascritti i metodi equals() e hashcode(), rendendo obbligatorio la sovrascrizione di tali metodi nel caso si voglia creare un altro tipo di **Tag**.

In quanto nell'interfaccia grafica viene mostrata per ogni tag, la quantita‘ di spesa, le tag stesse dovrebbero incorporare uno stato che puo‘ essere INCOME o EXPENSE, in modo da non dover controllare i movimenti.