Relazione Progetto Programmazione Avanzata (Jbudget105126)

ERIK MURTAJ MAT: 105126

20 Giugno 2020

Indice

1 Introduzione

In questa relazione sarà discussa l'implementazione del progetto di **Programmazione Avanzata**, sviluppato in Java.

Per l'implementazione è stato scelto il **pattern MVC**, quindi vi sarà una divisione tra Model, View e Controller. Divisione che verrà rispettata anche a livello di packages, in quanto, per permettere una migliore lettura del codice, il progetto è stato diviso in **3 packages (Model, View, Controller)**.

All'interno di ognuno di essi troveremo un package denominato **"Implementation"** dove vi saranno le varie classi che implementano le varie interfacce precedentemente definite. In aggiunta, all'interno del package **"Model"**, è presente un ulteriore package denominato **"Persistence"**, dove saranno implementate le interfacce e classi allo scopo di garantire la persistenza dei dati della nostra applicazione (funzionalità che sarà implementata nella consegna finale). Nel package View è presente un ulteriore package denominato **"GUI"** dove vi sono le classi relative alla gestione dell'interfaccia grafica.

2 Tecnologie utilizzate

Per questo progetto viene utiliazzato **Gradle** (precisamente la versione 2.6), il quale garantirà l'accesso alle risorse e altre tecnologie implementate durante la build del progetto stesso. È stata utilizzata come JavaVirtualMachine (**JVM**) la versione 11, precisamente AdoptOpenJDK 11.0.7+10.

Per la GUI viene utilizzato **JavaFx**, preferendolo rispetto all'implementazione a linea di comando per scelta personale.

La persistenza è stata garantita scegliendo di salvare i dati su file in formato json. Riguardo ciò, per la serializzazione e deserializzazione degli oggetti viene usata la libreria di Google denominata **Gson**.

Per la gestione delle date è stato scelto per praticità d'uso la libreria **LocalDate** all'interno del package org.joda.time. Ciò permette operazioni semplificate durante la schedulazione delle transazioni.

Viene inoltre utilizzata la libreria **Config** di typesafe.com, per la gestione del file di configurazione dell'applicazione (al momento contenente solo la valuta utilizzata).

3 Descrizione

Quest'applicazione gestisce spese e introiti familiari. Si focalizza su 4 principali entità:

- **Conti**
- **Transazioni (composte da più movimenti)**

- **Categorie**
- **Budget**

L'utente può inserire e rimuovere queste entità, per tenere traccia dell'andamento del bilancio familiare. .

.

Un conto può avere più transazioni e una transazione può avere più movimenti (Si definirà in seguito i vari tipi e le differenze). .

.

L'interfaccia grafica permette l'inserimento di una sola categoria per le transazioni, in quanto, essendo un gestore del bilancio di una famgilia, la gestione di transazioni con più categorie a livello di interazione con l'utente potrebbe risultare complessa. Nonostante ciò, nel back-end, le transazioni hanno una lista di categorie ed operano su di essa; perciò se in futuro si troverà la necessità di gestire più categorie nella stessa transazione potrà esser fatto semplicemente modificando la View. .

.

Una categoria può avere al massimo un figlio. .

.

Oltre alle normali transazioni possono essere inserite transazioni programmate, che avvengono con una certa frequenza o avverranno in futuro. Per quest'ultime, nel calcolo del loro valore, viene considerata la data di inizio, in quanto in quella data si è creato un debito/guadagno. Perciò in quella data la transazione varrà il valore del movimento che si ripete per il numero di volte che dovrà essere ripetuto (compito del schedulatore delle transizioni programmate).
.

.

Un budget può essere definito dall'utente per confrontare un'aspettativa riguardo alle spese/guadagni riguardo una determinata categoria e le spese/guadagni effettivi per quest'ultima. .

.

Non vi possono essere conti uguali (stesso nome, descrizione e tipo), non vi possono essere categorie uguali(stesso nome) e non vi possono essere budget uguali (stesso nome e categoria).

4 Funzionalità

4.1 Funzionalità base

In questo progetto sono state implementate le seguenti funzionalità:

- **Creazione di un/più conto/i**
  - **Calcolo del saldo del conto**
  - **Ottenimento della lista delle tranzioni** (per periodo di tempo o per categoria)

- **Aggiunta di transazioni relative ad un conto**
- **Aggiunta di movimenti relativi ad una transazione**
- **Aggiunta di transazioni programmate** (in base alla frequenza nella ripetizione del pagamento)
- **Aggiunta di categorie** (le quali saranno associate a transazioni e budget)
- **Aggiunta di budget** (i quali saranno associate a transazioni e budget)
  - **Calcolo della differenza tra budget prefissato e spese effettive** (per la determinata categoria a cui fanno riferimento)
- **Rimozione delle precedenti informazioni**
- **Trasferimento di fondi**: è possibile trasferire i fondi da un conto ad un altro.

4.2 Funzionalità avanzate

- **Persistenza dei dati**: sono implementate le classi per la persistenza su file dei dati.
- **Logging**: viene effettuato il logging dell'applicazione.
- **Interfaccia grafica**: sviluppata con l'utilizzo di JavaFx.
- **Creazione di statistiche per spese/ricavi in periodi di tempo**: l'applicazione permette la visualizzazione tramite grafico a linee, dell'andamento del bilancio totale, filtrando se voluto per categoria.

5 Descrizione delle responsabilità

5.1 Model

- (Interface) **Account**: le classi che implementano questa interfaccia hanno la responsabilità di gestire le informazioni di un conto (saldo, transazioni e tipo).
  - (Class) **BasicAccount**: implementazione di base dell'interfaccia Account. Salva le transazioni e le transazioni programmate in una lista e ne definisce il tipo. È possibile ottenere il saldo del conto tramite la funzione getBalance().
  - (Enum) **AccountType**: defnisce il tipo di account (ASSET o LIABILITY):
    - **ASSET**: rappresenta la disponibilità in denaro (conto bancario o cassa), essi aumenteranno il nostro bilancio totale.

        * **LIABILITY**: rappresenta un debito da estinguere (prestito o carta di credito), essi diminuiranno il nostro bilancio totale.

- (Interface) **Budget**: le classi che implementano questa interfaccia hanno la responsabilità di gestire le informazioni di un budget (categoria, nome e aspettativa).
  - (Class) **BasicBudget**: implementazione di base dell'interfaccia Budget.
- (Interface) **BudgetManager**: le classi che implementano questa interfaccia hanno la responsabilità di gestire e generare report dei budget.
  - (Class) **BasicBudgetManager**: implementazione di base dell'interfaccia BudgetManager. Genera un report con la funzione generate_report(), tra l'aspettativa delle spese per quella categoria e le spese effettive per essa.
- (Interface) **Category**: le classi che implementano questa interfaccia hanno la responsabilità di gestire le informazioni di una categoria.
  - (Class) **BasicCategory**: implementazione di base dell'interfaccia Category. Ha un attributo di tipo Category di nome "parent", in modo da permettere la gerarchia tra le categorie.
- (Interface) **Movement**: le classi che implementano questa interfaccia hanno la responsabilità di gestire un singolo movimento.
  - (Class) **BasicMovement**: implementazione di base dell'interfaccia Movement.Gestisce le informazioni relative ad un movimento (nome, data, importo e tipo). Può avere il campo data null (in tal caso sarà la stessa della transazione da cui deriva) oppure avere una propria data (esempio: transazioni programmate costituite da un totale di movimenti ripetituti ogni mese).
  - (Enum) **MovementType**: defnisce il tipo di movimento (INCOME o EXPENSE):
    * **INCOME**: rappresenta un ricavo verso il nostro account. Aumenterà il bilancio della transazione.
    * **EXPENSE**: rappresenta una spesa dal nostro account. Diminuirà il bilancio della transazione.
- (Interface) **Transaction**: le classi che implementano questa interfaccia hanno la responsabilità di gestire le informazioni di una transazione.
  - (Class) **BasicTransaction**: implementazione di base dell'interfaccia Transaction. Può essere composta da uno o più movimenti, che vengono salvati in una lista. Salva inoltre le categorie in un'altra lista perchè la transazione potrebbe comportare spese/ricavi su più categorie. Si può ottenere il bilancio della transazione tramite il metodo getTotal().

- (Interface) **ScheduledTransaction**: interfaccia che estende l'interfaccia Transaction. Le classi che implementano questa interfaccia hanno la responsabilità di gestire le informazioni di una transazione programmata (spese/ricavi futuri). Estende l'interfaccia Transaction in quanto una scheduled transaction sarà salvata nella lista di un account in modo da poter richiamare il metodo getTotal() su di esse e contarle nel bilancio dell'account.
  - (Class) **BasicScheduledTransaction**: implementazione di base dell'interfaccia ScheduledTransaction. Oltre alle informazioni di base di una transazione abbiamo anche EndDate, che definisce la fine della transazione programmata e la frequenza.
  - (Enum) **TransactionFrequency**: definisce la frequenza di una transazione programmata (Esempio: ogni mensile/bimestrale/annuale ecc). Vi è anche la voce "NONE" tra esse che identifica una transazione futura senza frequenza, quindi avrà solo una data di fine.
- (Interface) **Scheduler**: le classi che implementano questa interfaccia hanno la responsabilità di gestire la schedulazione delle transazioni programmate.
  - (Class) **TransactionScheduler**: implementazione dell'interfaccia Scheduler. Tramite il metodo schedule si aggiungono i movimenti nelle relative Transazioni in base al tempo trascorso e alla frequenza della transazione. In caso sia il giorno uguale all'EndDate o sia passato, la transazione viene marcata come completata.
- (Interface) **User**: le classi che implementano questa interfaccia hanno la responsabilità di gestire le informazioni dell'utente che utilizzerà l'applicazione.
  - (Class) **BasicUser**: implementazione di base dell'interfaccia User.
- (Interface) **ExpenseManagerState**: le classi che implementano questa interfaccia hanno la responsabilità di gestire i dati dell'applicazione, quindi il suo stato.
  - (Class) **BasicExpenseManagerState**: implementazione di base dell'interfaccia ExpenseManagerState. Permette la creazione di conti, dell'aggiunta e cancellazione delle transazioni e la creazione e rimozione dei tag. Queste informazioni vengono salvati in delle liste. Tiene inoltre conto della valuta che l'applicazione dovrà usare tramite la variabile currency di tipo Currency (java.util.Currency) e dell'utente che usa l'applicazione.

5.2 Persistence

- (Interface) **DataManager**: interfaccia che implementaranno le classi che avranno la responsabilità di gestire il salvataggio e l'ottenimento delle

informazioni da/verso specifiche fonti (file, server...), garantendo quindi la persistenza dei dati.

  - (Class) **JsonManager**: classe che implementa l'interfaccia DataManager e si occupa del salvataggio e recupero dei dati su/da file in formato Json. Utilizza un oggetto di tipo Gson a cui vengono aggiunti dei TypeAdapter per la serializzazione/deserializzazione degli oggetti dello stato.
  - **Serializzazione/Deserializzazione**: per la serializzazione e deserializzazione sono state implementate due classi (**LocalDateSerializer** e **InterfaceSerializer**). La prima viene usata per serializzare e deserializzare una data di tipo LocalDate. La seconda invece viene utilizzata per la serializzazione e deserializzazione delle altre classi, caratterizzate dall'implementazione di determinate interfacce. Le ultime due interfacce sono state implementate con l'aiuto del sito www.stackoverflow.com in quanto soddisfavano le mie esigenze al momento, ma sarà valutata la modifica di esse per il futuro.

5.3 Controller

- (Interface) **ExpenseManager**: le classi che implementeranno quest'interfaccia hanno la responsabilità di gestire i comandi e le funzionalità dell'apllicazione richieste dall'utente. In particolare permette la creazione dei vari dati (conti, categorie, transazioni) e l'esecuzione di funzioni come l'ottenimento del bilancio totale, derivato dalla somma dei bilanci di tutti gli account o l'ottenimento di determinate transazioni in base ad un predicate. Viene parametrizzato con un tipo che estende ExpenseManagerState, il quale verrà passato nel costruttore per poter operare su esso.

  - (Class) **BasicExpenseManager**: implementazione di base dell'interfaccia ExpenseManager. Permette l'interazione tra la GUI e il Model, soddisfando le operazioni richieste. Sarà implementata ulteriormente con l'aggiunta di nuove funzionalità.

5.4 View

- **View**: le classi che implementeranno questa interfaccia avranno la responsabilità di gestire l'input/output dell'utente.

5.5 GUI

- **JavaFxView**: classe che implementa l'interfaccia View, ottiene lo stage primario e lo mostra.

- **JavaFxExpendeManager**: classe che estende la classe astratta Application. Sovrascrivendo il metodo start() l'applicazione JavaFx può essere avviata, ciò avviene anche con il metodo stop()

- **Classi di controllo della GUI**: hanno lo scopo di gestire i vari stage comunicando con il controller che si interfaccia al model
  - **StageManager**: contiene metodi per utili alla costruzione di nuovi stage.
  - **JavaFxController**: controlla lo stage principale, permettendo la visualizzazione tablellare di tutti i dati. Ne permette inoltre la rimozione e la possibilità di visualizzare maggiori dettagli su di essi una volta selezionati.
  - **JavaFxNewAccountController**: controlla lo stage per l'inserimento di un nuovo account.
  - **JavaFxNewTransactionController**: controlla lo stage per l'inserimento di un nuova transazione.
  - **JavaFxNewCategoryController**: controlla lo stage per l'inserimento di un nuova categoria.
  - **JavaFxNewMovementController**: controlla lo stage per l'inserimento di un nuovo movimento.
  - **JavaFxNewBudgetController**: controlla lo stage per l'inserimento di un nuovo budget.
  - **JavaFxNewTransferController**: controlla lo stage per l'inserimento di un nuovo trasferimento di fondi.
  - **JavaFxStatsController**: controlla lo stage per la visualizzazione di un grafico dell'andamento del bilancio totale.

6 Test

I test vengono sviluppati utilizzando JUnit Jupiter. .

.

Essi hanno la stessa struttra, dove troviamo tramite il tag @BeforeEach l'inizializzazione dei dati che serviranno per i metodi di testing. .

.

La classe DataCreator contiene metodi utili alla creazione di oggetti che saranno utilizzati nei test.

7 Conclusioni

In questo progetto vengono rispettati i principi solid (uno degli argomenti del corso). L'utilizzo del pattern MVC ne aiuta l'implementazione e soprattutto, grazie all'utilizzo di numerose interfacce e alla generalizzazione, si garantisce una discreta estendibilità dell'applicazione. L'interfaccia User al momento non è utilizzata ai fini del funzionamento dell'applicazione, ma potrà essere in futuro estesa prevedendo l'identificazione dell'utente tramite username e password in caso si preveda una sicronizzazione dei dati tra più dispositivi, necessitando perciò di un'autenticazione.

8 UML

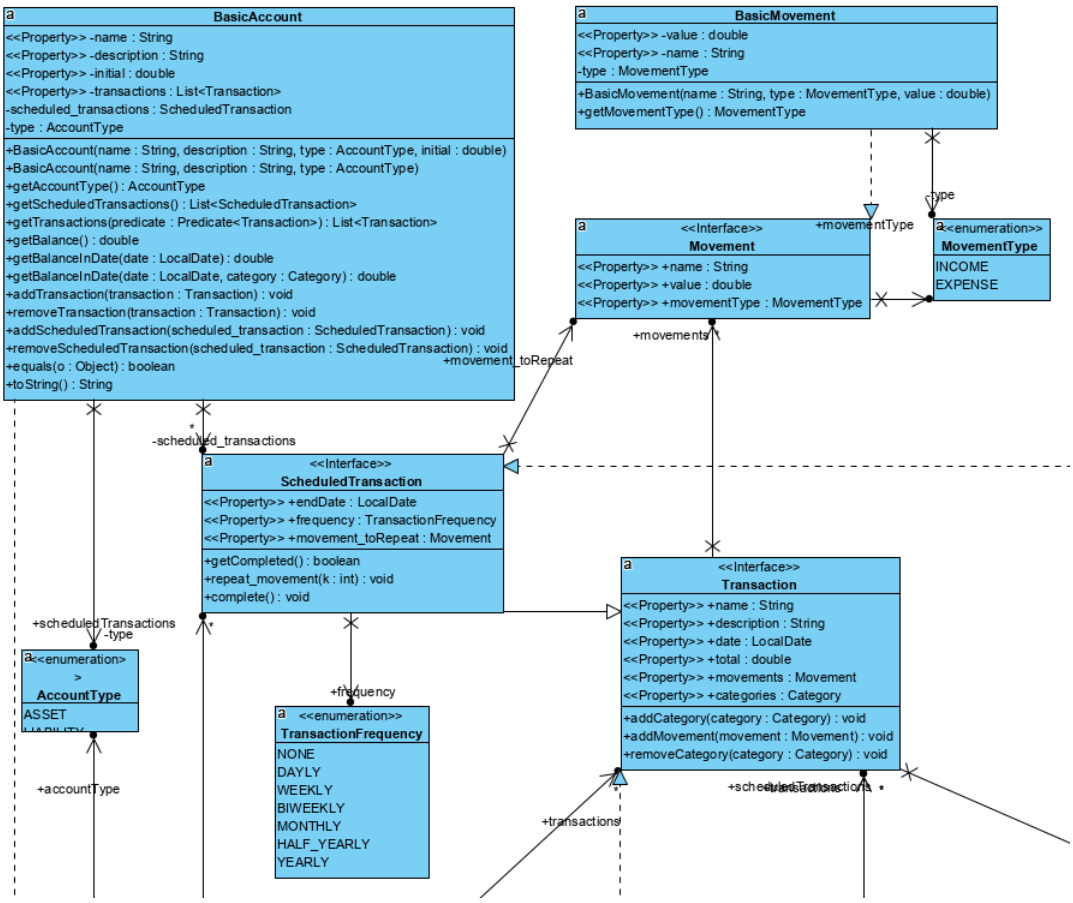

Figure 1: Model

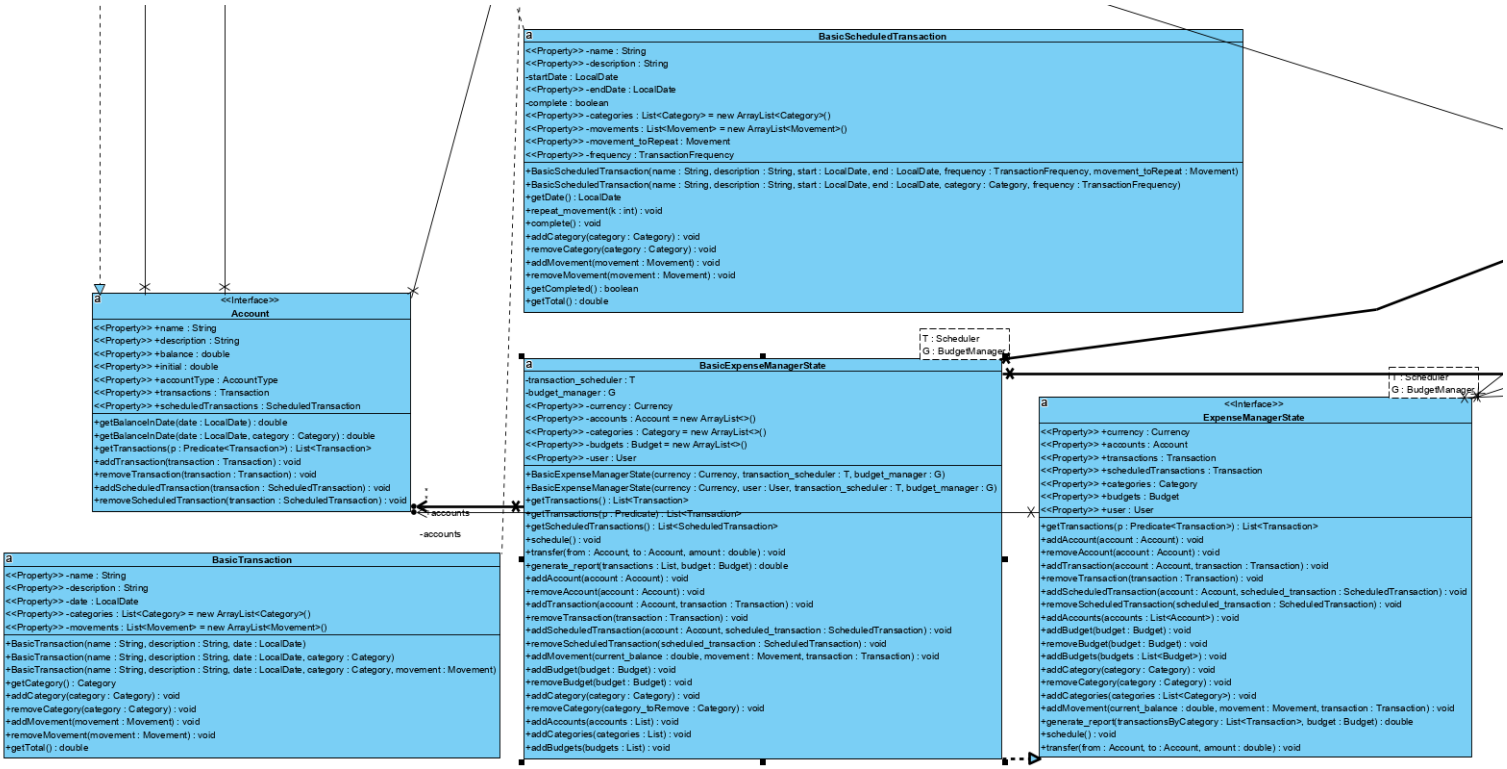

Figure 2: Model

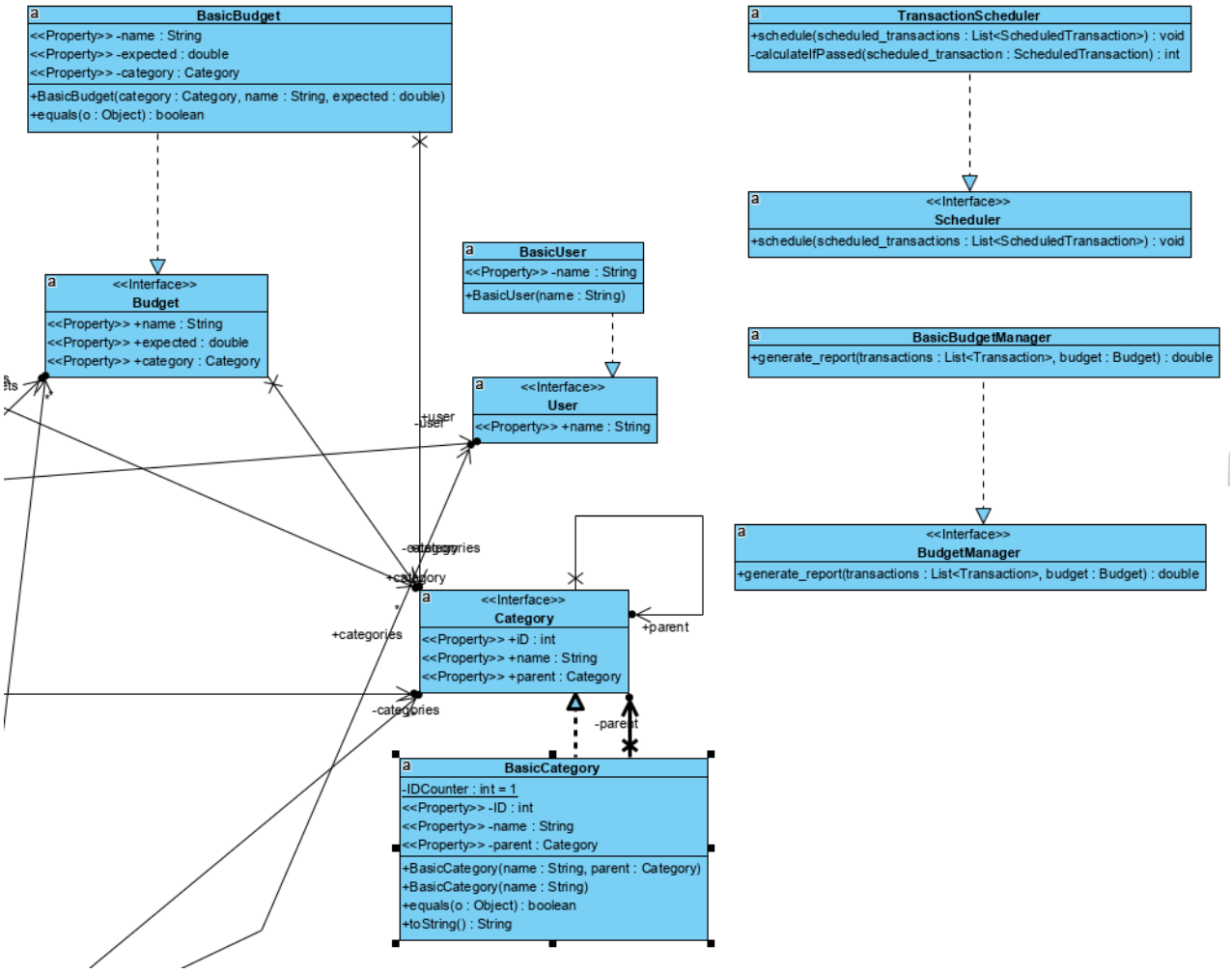

Figure 3: Model

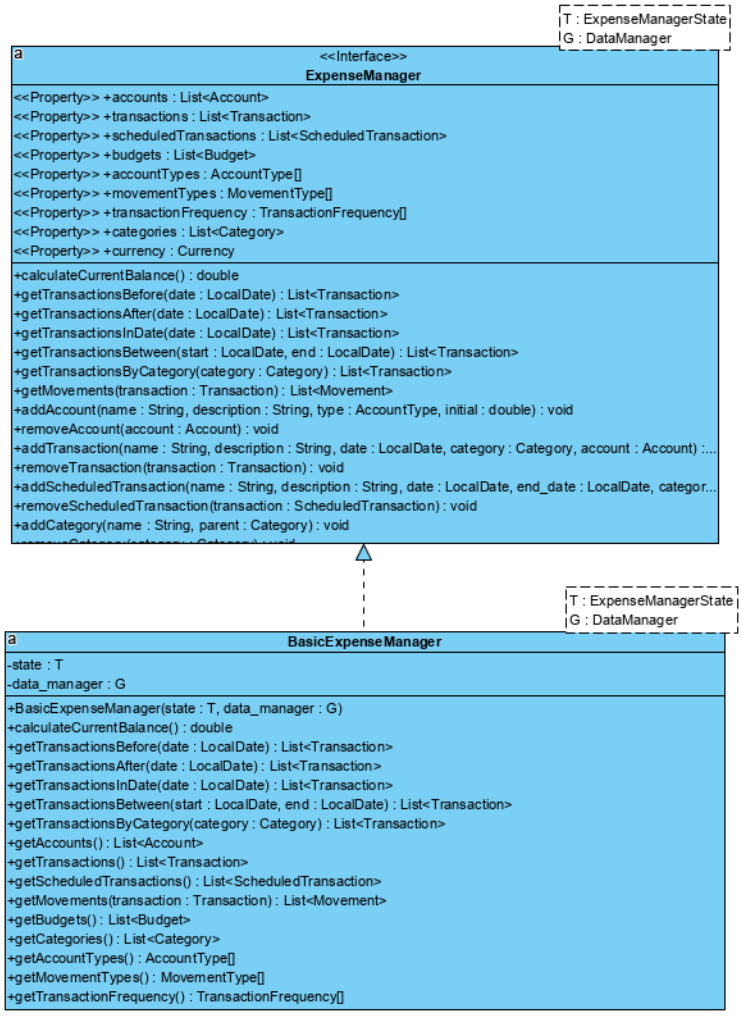

Figure 4: Controller

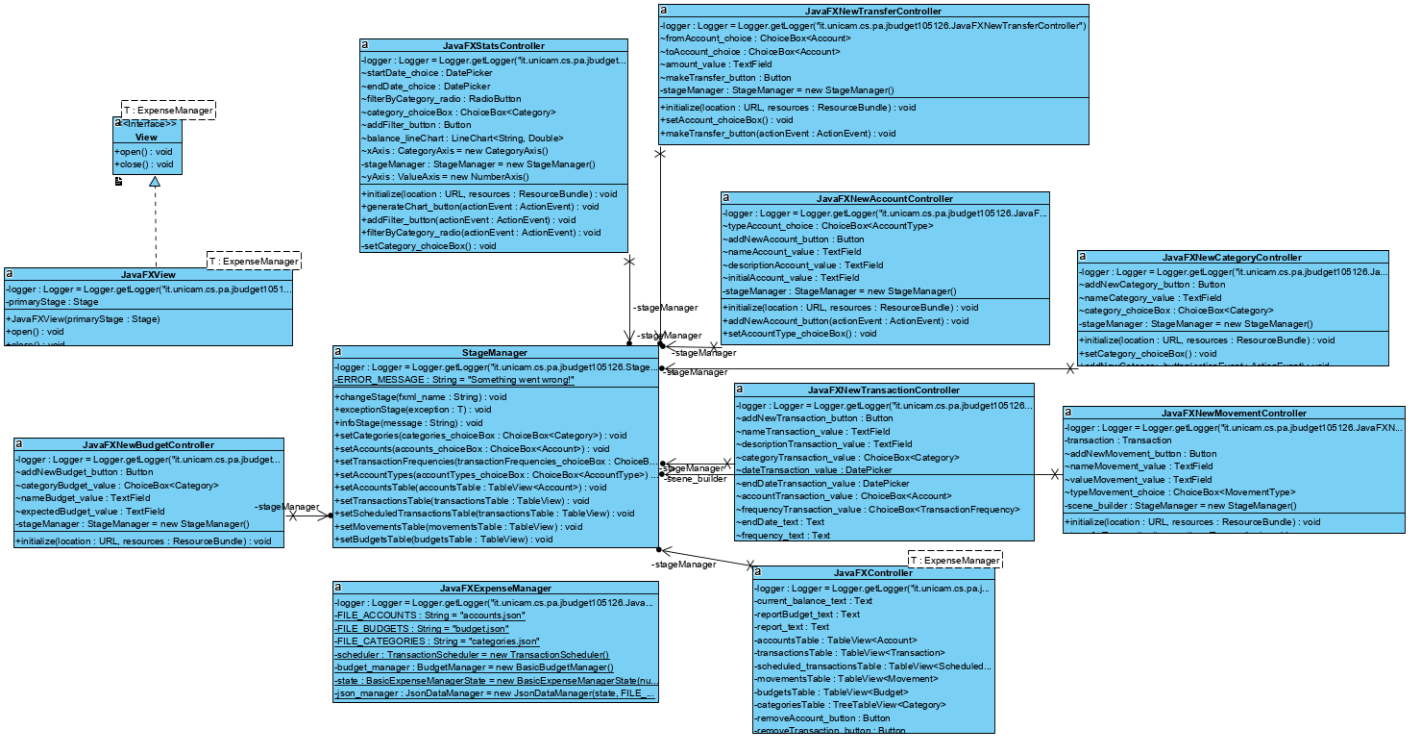

Figure 5: View