Giorgio Paoletti 105056 giorgio.paoletti@studenti.unicam.it

Applicazione per la Gestione Asset Familiare

Programmazione Avanzata Progetto – Giugno/Luglio 2020

A seguito dell'analisi svolta riguardo i sistemi di controllo per le spese familiari, sono stati individuati tre concetti principali:

- *Conto:* rappresenta un insieme di operazioni contabili (scritture) che servono per determinare e dimostrare le spese sostenute, gli introiti, il rapporto fra le entrate e le uscite, l'ammontare di debiti o crediti.
  Si individuano 2 tipologie di conti:
  - gli *asset* che rappresentano la disponibilità di denaro;
  - le *liabilities* che rappresentano invece dei *debiti* da estinguere.
- *Movimento:* rappresenta una *uscita* o una *entrata* da un conto.
- *Transazione:* rappresenta un insieme di movimenti.

Per permettere l'integrazione di nuove funzionalità si è scelto di utilizzare il design pattern *Model-view-controller* (MVC) e rispettare i principi SOLID.

Di seguito vengono illustrate le interfacce utilizzate per rappresentare il Model dell'applicazione e gestire i concetti sopra citati.

- *Movement*: l'interfaccia verrà implementata dalle classi responsabili della gestione del singolo movimento.
  Il movimento è associato a una transazione che ne gestisce la data e la lista dei tag (categoria).
  La classe ha la responsabilità di gestire la modifica delle informazioni associate al movimento: descrizione, importo, account associato e il tipo di movimento.
  L'interfaccia viene implementata dalla classe *MovementImplementation.*
  Le tipologie di movimento vengono gestite dall' enumerazione *MovementType (DEBITS, CREDITS).* La tipologia di movimento determina l'effetto di un movimento su un conto. Infatti, il saldo d'un conto di tipo *ASSET* crescerà con movimenti di tipo *CREDITS* e diminuirà con movimenti di tipo *DEBITS*. Viceversa, il saldo d'un conto di tipo *LIABILITIES* aumenterà con movimenti di tipo *DEBITS* e diminuirà con movimenti di tipo *CREDITS*. All'interno di una transazione i movimenti *DEBITS* saranno trattati sempre come negativi, quelli *CREDITS* come positivi.

- *Transaction*: l'interfaccia verrà implementata dalle classi responsabili della gestione della singola transazione.
  La transazione gestisce una lista di movimento a cui passa la lista dei tag e la data.
  Tramite un metodo sarà possibile ricavare il saldo (la variazione totale dei movimenti) della transazione.
  L'interfaccia viene implementata dalla classe *TransactionImplementation*.

- *Account*: l'interfaccia verrà implementata dalle classi responsabili della gestione del singolo conto.
  La classe ha la responsabilità di gestire la modifica delle informazioni associate al conto: descrizione, saldo iniziale, la lista dei movimenti e il tipo di conto.

Tramite un metodo sarà possibile ricavare il saldo attuale del conto. Inoltre, è possibile accedere alla lista dei movimenti e a quelli che soddisfano un determinato predicato.
L'interfaccia viene implementata dalla classe *AccountImplementation*.
Le tipologie di conto vengono gestite dall' enumerazione *AccountType* (ASSET, LIABILITIES).

- *Ledger*: questa interfaccia è implementata dalle classi che hanno la responsabilità di gestire tutti i dati dell'applicazione. È responsabile della creazione dei conti, dell'aggiunta e cancellazione delle transazioni, della creazione e cancellazione dei tag. Inoltre, mantiene la lista delle transazioni schedulate. Si occupa di schedulare le transazioni ad una certa data.
  L'interfaccia viene implementata dalla classe *LedgerImplementation*.

- *ScheduledTransaction:* indica una transazione o una serie di transazioni schedulate ad una certa data. L'interfaccia viene implementata dalla classe *ScheduledTransactionImplementation* che ha la responsabilità di generare le transazioni derivanti da un primo giorno, da un numero di transazioni, un ammontare totale e il numero di giorni tra le varie transazioni.

- *Budget*: ha la responsabilità di rappresentare e gestire un particolare budget. Ogni budget associa ad ogni tag un importo che indica l'ammontare di spesa/guadagno per il particolare tag. Ogni budget, inoltre, costruisce il predicato per selezionare i movimenti di interesse. È responsabilità delle sottoclassi definire i criteri per la selezione dei movimenti.
  L'interfaccia viene implementata dalla classe *BudgetImplementation*.

- *BudgetReport*: ha la responsabilità di mostrare gli scostamenti di spesa/guadagno rispetto ad un particolare budget. Il *BudgetReport* viene costruito da un *BudgetManager*.
  L'interfaccia viene implementata dalla classe *BudgetreportImplementation*.

- *BudgetManager*: ha la responsabilità di costruire il *BudgetReport* associato ad un *Budget* e ad un *Ledger*.
  L'interfaccia viene implementata dalla classe *BudgetManagerImplementation*.

- *Tag*: ha la responsabilità di definire una categoria di spesa/guadagno.
  L'interfaccia viene implementata dalla classe *TagImplementation*.

- *ClassRegistry*: ha la responsabilità di gestire le istanze delle classi *Transaction*, *Account*, *Tag*, *ScheduledTransaction* e *Movement*.

Si definiscono le interfacce responsabili della rappresentazione del controller dell'applicazione:

- *Controller*: questa interfaccia è implementata dalle classi che hanno la responsabilità di gestire la conversazione tra la *View* e il *Model*.
  L'interfaccia viene implementata dalla classe *ControllerImplementation*, che si occupa della gestione delle funzionalità messe a disposizione dall' applicazione. Inoltre, gestisce le istanze del Ledger, del Budget e del LoggerManager.
  Inoltre, le istanze della classe *ControllerImplementation* vengono costruite tramite il framework di *Dependecy Injection google Guice* utilizzando il Modulo *ControllerImplementationModule*.
- *Saver*: questa interfaccia è implementata dalle classi che hanno la responsabilità di gestire il salvataggio dei dati su file.

L'interfaccia viene implementata dalla classe *SaverGSon* utilizzando le *JSon google API*
Ciò permette di salvare le istanze delle classi Ledger e Budget create dalla classe ControllerImplementation salvandole su file json che sono collocati nella sottocartella resources/data del progetto.

- *Loader*: questa interfaccia è implementata dalle classi che hanno la responsabilità di gestire il caricamento dei dati da file.
  L'interfaccia viene implementata dalla classe *LoaderGSon* utilizzando le *JSon google API*
  Ciò permette di caricare le istanze delle classi Ledger e Budget da file json che sono collocati nella sottocartella resources/data del progetto.
  La serializzazione e la deserializzazione delle varie classi sono gestite tramite Classi apposite che implementano Il JSonSerializer e JsonDeserializer parametrizzate alla classe opportune.

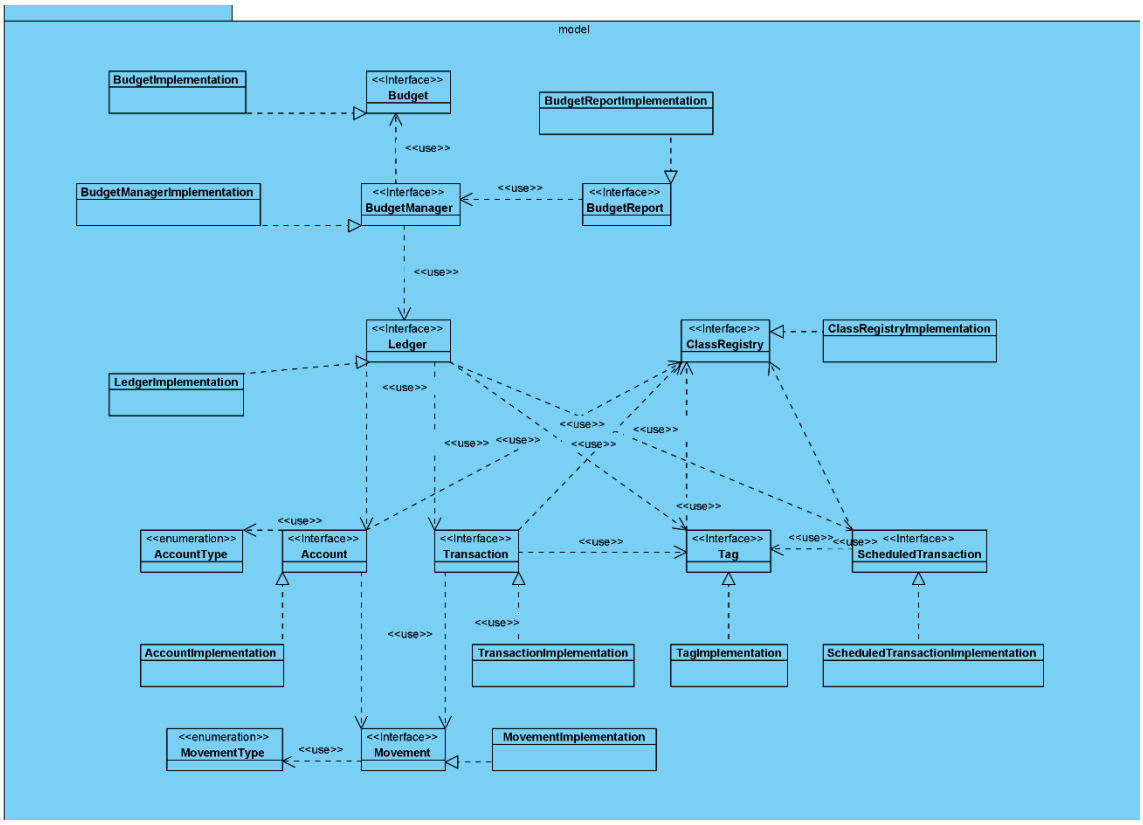
- *LoggerManager*: l'interfaccia verrà implementata dalle classi responsabili della gestione del Logger.
  Linterfaccia viene implementata dalla classe LoggerMangerImplemenattion che fa uso di un FileHandler con assegnato un SimpleFormatter.
  Ciò permette di salvare le operazioni effettuate dalla classe ControllerImplementation salvandole su file di Log che sono collocati nella sottocartella resources/log del progetto.

Si definisce la classe responsabile dell'interfacciamento con l'utente:

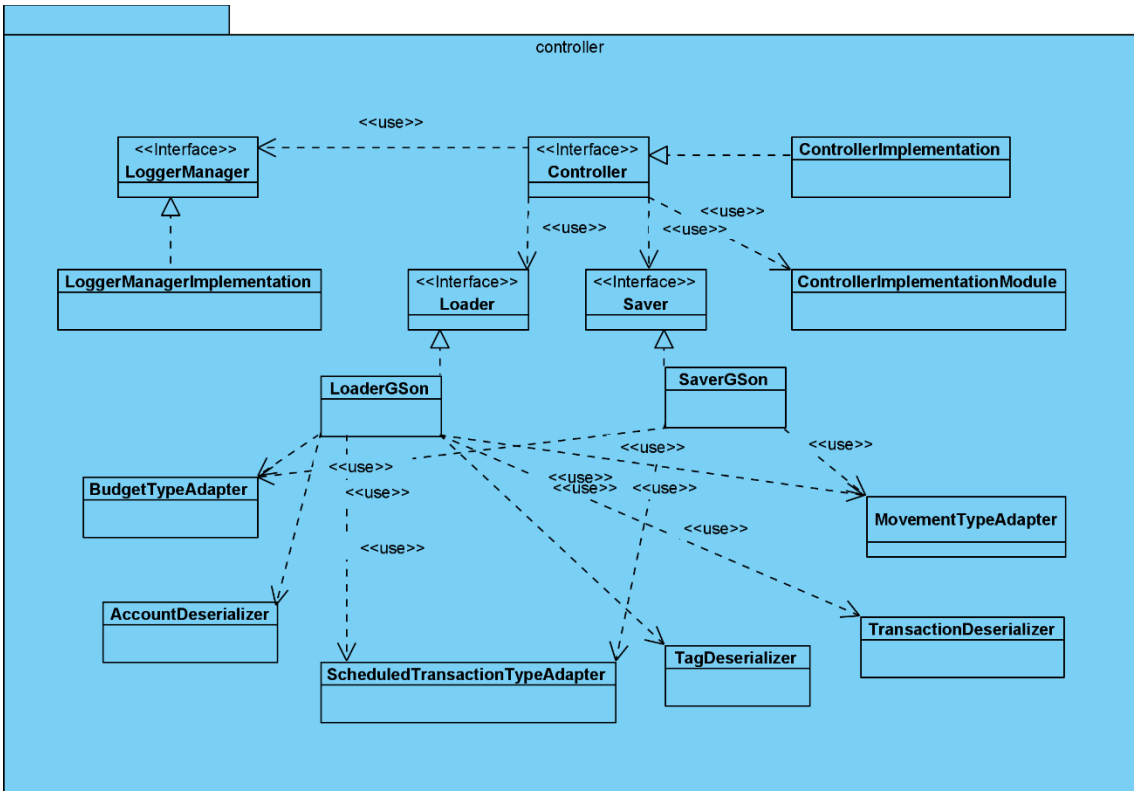
- *JavaFxViewController*: ha la responsabilità di costruire l'interfaccia grafica per l'utente finale e fa da Controller per il file *JavaFxView.fxml* costruito con l'uso del programma *SceneBuilder*
  Il file FXML è collocato nella sottocartella resources/fxml del progetto.
  La classe è stata implementata tramite l'utilizzo delle *JavaFX API*
  Inoltre, si è utilizzata la classe *CheckComboBox* della libreria *ControlsFX*

I concetti sopra citati e le relazioni che intercorrono tra le varie componenti sono illustrati nel seguente diagramma UML:

- Struttura classi Model:



- Struttura classi Controller:



Sono state implementate le seguenti Funzionalità:

- La classe *LedgerImplementation* che gestisce tramite ArrayList i seguenti componenti:
  - La lista delle Transazioni
  - La lista dei Conti
  - La lista delle Transazioni Programmate
  - La lista delle Categorie disponibili per le Transazioni

  Su tali liste si opera con i seguenti metodi:
  - Aggiungere una transazione a quelle già esistenti
  - Tramite il metodo privato *checkTransaction* viene verificato che, la transazione inserita contenga minimo un movimento al suo interno, che i movimenti siano collegati a conti esistenti e che, al minimo, sia assegnata una categoria alla transazione.
  - Aggiungere un conto a quelli già esistenti
  - Aggiungere una categoria a quelle già esistenti
  - Aggiungere una transazione programmata a quelle già esistenti
  - Schedulare le transazioni programmate
  - Rimuovere una transazione: ciò comporta la rimozione di tutti movimenti legati ad essa e la rimozione dei movimenti dai relativi conti.
  - Rimuovere un conto: ciò comporta la rimozione di tutti i movimenti a esso collegati e la rimozione di tutti movimenti dalle transazioni a cui sono legati.
    Se una Transazione rimane senza movimenti viene rimossa dalla lista delle transazioni.
  - Rimuovere una transazione programmata: ciò comporta la rimozione di tutte le transazioni schedulate e di conseguenza la rimozione di tutti i movimenti a esse collegati.

- La classe *TransactionImplemetation* che gestisce tramite una ArrayList la lista dei movimenti a cui è collegata la data della transazione e la lista delle categorie che la contraddistinguono.
  Inoltre, si è dotata la classe di un metodo che permette di ottenere il saldo totale della transazione.

- La classe *TagImplementation* che gestisce la singola categoria.

- La classe *AccountImplemetation* che gestisce tramite una ArrayList la lista dei movimenti a cui è collegata. La classe è dotata di un metodo che permette di ottenere il bilancio del conto.

- La classe *BudgetImplementation* che permette la gestione tramite una Map di una lista di Categoria collegata a un Budget aspettato.
  Sono stati sviluppati metodi per l' aggiunta e la rimozione di nuovi budget.

- La classe *ClassRegistryImplementation* permette la gestione delle istanze delle classi *TransactionImplemetation, TagImplementation, AccountImplemetation, ScheduledTransactionImplementation.*

- La Classe *ScheduledTransactionImplementation,* tramite una lista di transazioni, permette la gestione di una Transazione programmata.

- La Classe BudgetReportImplementation gestisce il report che viene generato tramite la classe *BudgetManagerImplementation* prendendo come parametri un Ledger e un Budget.

- La classe *ControllerImplementation* mette a diposizione le funzionalità proposte dall'applicazione elencate di seguito:
  - Aggiungere un Entrata o una Spesa indicando il conto e l'importo
  - Aggiungere un Trasferimento indicando i 2 Conti, l'importo trasferito e la commissione
  - Rimuovere una transazione
  - Ottenere il saldo di una transazione
  - Aggiungere un Budget indicando Categoria e importo
  - Generare un report mettendo in relazione il ledger e il budget
  - Aggiungere e rimuovere un conto
  - Ottenere il bilancio di un conto
  - Aggiungere e rimuovere (da sviluppare) una categoria di transazione
  - Aggiungere e rimuovere una transazione programmata
  - Caricare e Salvare i dati del Ledger e del Budget su file json
  - Disporre di un Log Manager per salvare le azioni eseguite sul controller su un file log

- La classe *LoggerManagerImplementation* permette la creazione di un log dato dalla combinazione di un *LogManager*, un *FileHandler* e un *SimpleFormatter* che salvano su file le operazioni effettuate sulla classe *ControllerImplementation.*

- La classe *ControllerImplementationModule* rappresenta il modulo utilizzato per istanziare la classe *ControllerImplementation* tramite il framework di *Dependency Injection* messo a disposizione da *google Guice*. Il modulo permette l'Inject dei parametri legati alle Interfacce *Ledger*, *Budget*, *LoggerManager*, *Saver* e *Loader*.

- Le classi *SaverGSon* e *LoaderGSon* che mettono a disposizione due metodi ciascuno per il salvataggio e il caricamento dei dati relativi al Ledger e del Budget.
  Le classi fanno uso della libreria GSon sviluppata da Google.
  Inoltre, sono stati sviluppati dei *TypeAdapter* relativi a ogni classe del model, così da consentire la serializzazione e la deserializzazione utilizzate dalle classi *SaverGSon* e *LoaderGSon*.

- La classe *JavaFxViewController* costituisce sia la *View* nel Modello MVC sia il controller per il file *JavaFxView.fxml*
  L'interfaccia grafica costruita con il programma *SceneBuilder* è costituita da più *TabPane*; ogni *TabPane* costituisce una funzionalità del programma e permette di:
  - *Visualizzare i dati tramite TableView.*
  - Aggiungere nuovi elementi tramite un *GridPane* che accetta vari campi a seconda dell'elemento preso in considerazione.
  - Eliminare l'elemento selezionato nella *TableView*.

Per assicurarsi del corretto funzionamento delle componenti sopra citate sono stati sviluppati dei test che mirano a simulare dei possibili scenari.

I test vengono sviluppati tramite il framework *JUnit 5*, di seguito le classi utilizzate:

- *AccountImplementationTest*:
  - Test 1 si occupa di verificare il corretto comportamento dei metodi *getInstanceByID* e *getInstance*.
  - Test 2 si occupa di verificare il corretto comportamento dei metodi addMovement in presenza di un account di tipo Assets.
  - Test 3 si occupa di verificare il corretto comportamento dei metodi addMovement in presenza di un account di tipo Liabilities.
  - Test 4 si occupa di verificare il corretto funzionamento del metodo getMovements a cui viene passato un predicato.
  - Test 5 si occupa di verificare il corretto funzionamento del metodo removeMovement.

- *LedgerImplementationTest*:
  - Test 1 si occupa di verificare il corretto funzionamento della creazione di una transazione.
  - Test 2 verifica il corretto comportamento del metodo schedule e generateTransactions.

- *MovementImplementationTest*:
  - Test 1 si occupa di verificare il corretto comportamento dei metodi getIstance e getIstanceById.
  - Test 2 si occupa di verificare le relazioni che intercorrono tra la classe Transaction e Movement.

- *TransactionImplementationTest*: Il test verifica il corretto comportamento della classe *TransactionImplementation.*

- *BudgetImplementationTest*:

  - Test 1 verifica il corretto funzionamento del metodo getPredicate dell classe BudgetImplementation.
  - Test 2 verifica il comportamento del metodo generateReport della classe BudgetManagerImplementation.

- *ScheduledTransactionImplementationTest*: Il test verifica il corretto comportamento della classe ScheduledTransaction

- *LoaderGSonTest*: Il test si occupa di verificare il corretto caricamento dell'istanza del Ledger e del Budget.

- *SaverGSonTest*: Il test si occupa di verificare il corretto salvataggio dell'istanza del Ledger e del Budget.

- *ControllerImplementationTest:*

  - Test 1 verifica il corretto funzionamento dei metodi addTransaction e addTrasferimento della classe controller.
  - Test 2 Verifica il corretto funzionato della generazione di un report.
  - Test 3 Verifica il corretto funzionamento dei meccanismi per la creazione di Transazioni Schedulate.