RELAZIONE FINALE “JBUDGET”

Vittorio Rinaldi 100763

Premessa:

L’ applicazione è pubblicata dall’autore come “Open Source senza scopo di lucro” (licenza GPL v3).

Pattern utilizzati:

* MVC (Model-View-Controller, architetturale)
* Singleton (creazionale).
* Facade per gestire un sottosistema complesso tramite un’interfaccia semplice (Ledger) nell’interazione tra l’utente ed il resto del model (strutturale)

Sviluppato utilizzando le seguenti versioni:

* Gradle 6.5
* Java 11
* JavaFX 14

È presente documentazione JavaDoc!

Note dello sviluppatore

Unicamente per questioni legate alle tempistiche di consegna, l’interfaccia utente implementata è in fase di lavorazione: con questo si intende che la filosofia scelta dallo sviluppatore è “Qualità prima della quantità”.

In altre parole, a costo di un minor ritmo, ma con ben maggiore usabilità, *si è scelto* di NON intraprendere una programmazione imprudente ed acerba, ma una che prevedesse la cura di ogni funzionalità prima della loro pubblicazione.

Attualmente, tramite interfaccia grafica è possibile gestire perfettamente la creazione di nuovi Account e Tag nell’applicazione, la loro rimozione, nonché di visualizzarne le caratteristiche.

Si sono risolti i problemi legati alla precedente consegna e la parte di interfaccia grafica accessibile all’utente in questo momento è garantita come perfettamente funzionante (anche al netto di imprevisti); si fa notare come lato back end moltissime altre funzionalità siano curate e già altrettanto pronte, necessitanti unicamente di essere collegate all’utente finale tramite GUI (si parla di: gestione dei movimenti, gestione transazioni, la schedulazione di transazioni, accantonamento di spese per Tag specifici (budget) e report consuntivi che rappresentano il saldo tra entrate ed uscite in un insieme di Transazioni). 🡪 Saranno disponibili a breve.

Anche nel lungo termine è in programma un ampliamento del programma:

* Lettura e scrittura dei dati utilizzando file Json, per garantire la sincronizzazione tra dispositivi, magari tramite cloud direttamente dall’applicazione.

I file FXML sono stati separati dalla directory del FxCcontroller per una questione di sintassi e pulizia;

sono locati nella cartella Resources.

Implementazione test-driven per tutte le funzionalità delle seguenti interfacce:

* Ledger
* Account
* Transaction
* Movement

Per iniziare…

L’applicazione può essere avviata tramite Console digitando il comando “gradle run” nella directory del progetto, oppure tramite qualsiasi IDE importando il progetto Gradle.

Package it.unicam.cs.pa.jbudget100763

Qui sono contenute tutte le interfacce e classi del Progetto.

**Main:** Classe principale, permette di avviare l’applicazione e la GUI scelta, attualmente JavaFX.

Package it.unicam.cs.pa.jbudget100763.controller

include la classe che ha il compito di ricevere i comandi dell'utente (in genere attraverso il View) e di attuarli modificando lo stato degli altri due componenti del MVC.

**Controller:** classe, riceve i comandi dell'utente tramite la View e li attua modificando lo stato del Model. Gestisce il Ledger, il Budget ed il TagBudgetReport.

Package it.unicam.cs.pa.jbudget100763.view

stabilisce i comportamenti ritenuti fondamentali per visualizzare i dati contenuti nel model ed occuparsi dell'interazione con gli utenti.

**View:** Interfaccia, ha la responsabilità di indicare le direttive principali riguardo l'interazione dell'utente con l'applicazione.

Package it.unicam.cs.pa.jbudget100763.view.javafx

include il percorso di implementazione della GUI tramite il framework JavaFX, seguendo le direttive della View.

**App:** Applicazione principale dell'implementazione tramite javaFX.

**FxController:** classe, gestisce le schermate da avviare e raccoglie tutte le interazioni dell'utente tramite la GUI, inoltrandole al controller dell'applicazione principale.

**AccountController:** classe, gestisce la finestra per la creazione dei conti.

**TagController:** classe, che gestisce la finestra per la creazione dei Tag.

Package it.unicam.cs.pa.jbudget100763.model

ha il compito di gestire i dati dell’applicazione, fornisce i metodi per accedere ad essi.

**Account:** interfaccia, è implementata dalle classi che hanno la responsabilità di gestire un conto.

**Budget:** interfaccia, ha la responsabilità di rappresentare e gestire un particolare budget, ovvero la previsione di spesa/guadagno per uno o più Tag

**Ledger:** interfaccia, è implementata dalle classi che hanno la responsabilità di gestire tutti i dati dell'applicazione.

**Movement:** interfaccia, è implementata dalle classi che hanno la responsabilità di gestire un singolo movimento.

**ScheduledTransaction:** interfaccia, indica una transazione o una serie di transazioni schedulate (previste) ad una certa data.

**Tag:** interfaccia, è implementata dalle classi che hanno la responsabilità di definire una categoria di spesa/guadagno.

**TagBudgetReport:** interfaccia, è implementata dalle classi che hanno la responsabilità di gestire un budget (bilancio) previsto per uno o più tag.

**Transaction:** interfaccia, è implementata dalle classi che hanno la responsabilità di gestire una transazione.

**AccountImpl:** classe, ha la responsabilità di gestire un conto.

* double getBalance(): aggiorna in runtime il saldo attuale partendo da quello iniziale e sommando di volta in volta tutti i movimenti avvenuti
* Set<Movement> getMovements(): I movimenti correlati a questo account hanno il riferimento nelle transazioni (e quindi nel ledger), successivamente questi vengono collegati all'account
* Set<Movement> getMovements(Predicate<Movement> condition): ritorna la lista dei movimenti di questo account che rispettano il predicato

**BudgetImpl:** classe, un budget associa ad ogni tag un importo che indica l'ammontare di spesa/guadagno previsto per quel particolare tag.

* double getBalance(Tag t): rerstituisce la cifra accantonata (prevista) per quel tag
* void setBalance(Tag t, Double expected): aggiunge un nuovo tag e relativo accantonamento previsto, al report
* Set<Tag> tags(Predicate<Transaction> condition): ritorna i tag utilizzati nelle transazioni che rispettano una certa condizione (es: avvenute in un determinato periodo di tempo)
* Predicate<Transaction> after(GregorianCalendar date): lambda expression per definire una transazione avvenuta dopo la certa data
* Predicate<Transaction> before(GregorianCalendar date): lambda expression per definire una transazione avvenuta prima la certa data

**LedgerImpl:** classe, ha la responsabilità di gestire tutti i dati dell'applicazione.

* static Ledger getInstance(): Singleton implementation
* Account addAccount(AccountType type, String name, String description, double openingBalance): Crea un nuovo account nell'applicazione
* Tag addTag(String name, String description): Crea un nuovo tag nell'applicazione
* void removeTag(Tag t): tag da rimuovere
* boolean addTransaction(GregorianCalendar date): crea una nuova transazione e la inserisce nella lista dell'applicazione
* Set<Transaction> getTransactions(Predicate<Transaction> condition): ritorna la lista di transazioni che rispettano un certo predicato
* Set<ScheduledTransaction> getScheduled(): ritorna tutte le transazioni schedulate dell’applicazione
* Set<ScheduledTransaction> searchScheduledTransaction(GregorianCalendar d): restituisce le scheduled transaction fissate a quella data
* void addScheduledTransaction(ScheduledTransaction st): Aggiunge l'istanza Sch.Tran. alla lista "contenitore" dell'applicazione presente nel ledger
* void schedule(ScheduledTransaction st): Combina (aggiunge) tutte le transazioni previste ad una data futura con una scheduled transaction
* boolean scheduleSpecificTransaction(Transaction transaction, ScheduledTransaction st): Combina manualmente una transazione con una Sched.Transaction, devono manifestarsi lo stesso momento

**MovementImpl:** implementa la responsabilità di gestire un singolo movimento

* il costruttore del movimento collega automaticamente lo stesso con la transazione associata
* getters & setters dei propri campi: MovementType, amount , date, Tag, description, Account, Transaction

**RegistryImpl<T>:** DISCLAIMER: Classe embrionale in via di sviluppo per la persistenza dei dati dell'applicazione.

**ScheduledTransactionImpl:** indica una transazione o una serie di transazioni schedulate (previste) ad una certa data.

* boolean isCompleted(): verifica se la serie di transazioni si sia manifestata o meno
* boolean addTransaction(Transaction t): se la transazione da schedulare rispetta i requisiti della scheduledTransaction allora viene inserita nella medesima

**TagImpl:** definisce una categoria di spesa/guadagno.

* Getters & setters dei propri campi: nome, descrizione

**TagBudgetReportImpl:** mostra il saldo di positivo/negativo di uno o più Tag.

* Map<Tag, Double> getTagBalance(Set<Transaction> transactions): report - saldo complessivo di ogni distinto tag trovato nelle transizioni in input
* Set<Tag> totalTags(): ritorna tutti i tag creati nell'applicazione
* Map<Tag, Double> report(Predicate<Transaction> condition): ritorna il saldo/bilancio di tutti i tag trovati nelle transazioni che rispettano il Predicato

**TransactionImpl:** ha la responsabilità di gestire una transazione.

* Set<Tag> getTags(): mostra la lista di tutti i tag distinti dei movimenti contenuti
* void addTag(Tag t): Tag da inserire a tutti i movimenti della transazione e quindi automaticamente alla transazione stessa
* double getTotalAmount(): somma algebrica del valore di tutti i movimenti contenuti

**AccountType:** enumerazione che rappresenta le tipologie di account (ASSETS, LIABILITIES, CASH)

**MovementType:** enumerazione che rappresenta la direzione di un movimento su un conto (OUTCOME, INCOME).

