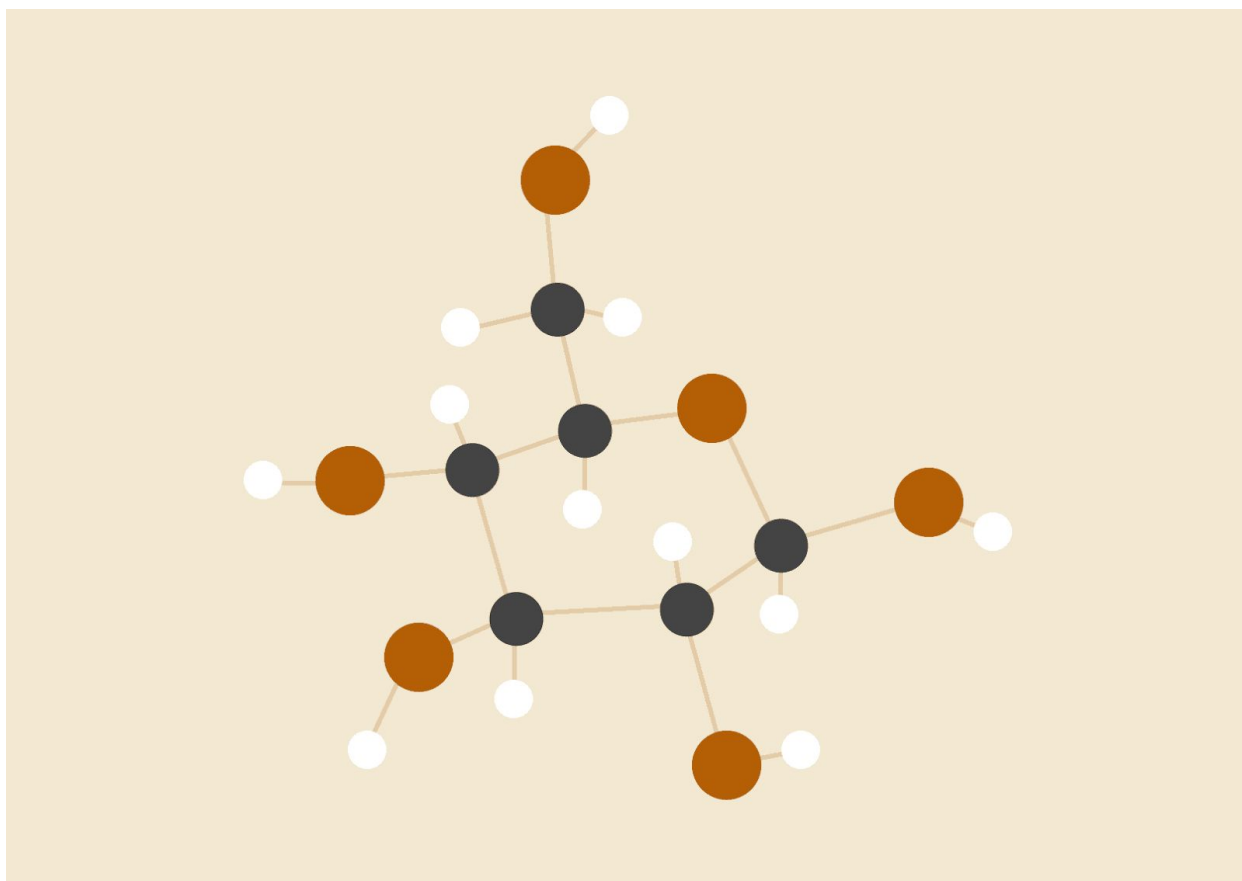


RELAZIONE PROGETTO PROGRAMMAZIONE AVANZATA

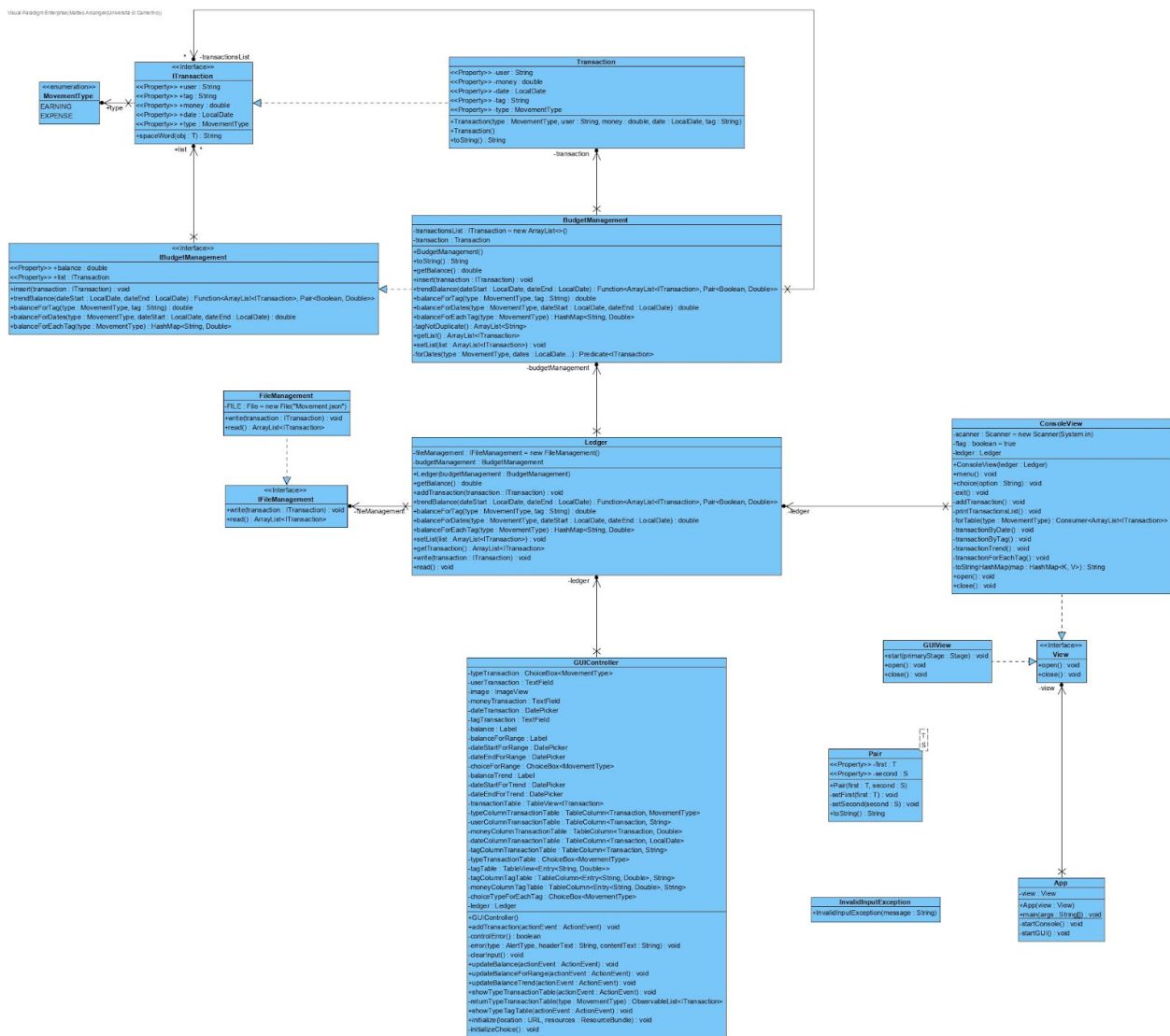


Flavio Pocari

INTRODUZIONE

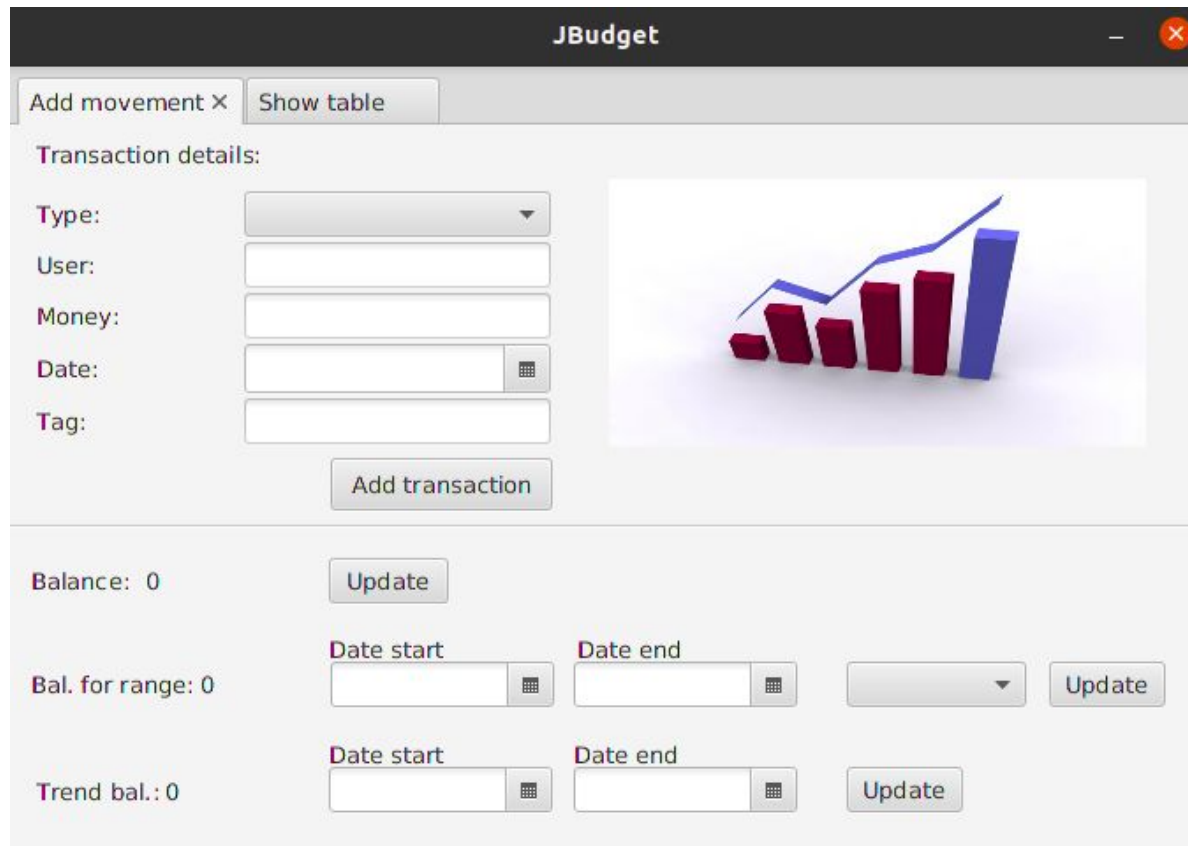
Relazione che ha lo scopo di introdurre le caratteristiche, il funzionamento all'avvio, e le mansioni che le classi/interfacce Java svolgono all'interno del progetto 'JBudget' per l'esame di PA.

Per rendere più chiara la composizione della struttura del progetto, e quindi comprendere quello che è stato realizzato, si rende disponibile un diagramma UML.



Il progetto può essere avviato in modalità GUI, la quale struttura è stata realizzata mediante SceneBuilder, oppure avviato in modalità Console.

La modalità GUI apparirà nel seguente modo:



The screenshot shows the JBudget application window. At the top, there's a title bar with 'JBudget' and standard window controls. Below the title bar, there are two tabs: 'Add movement x' (active) and 'Show table'. The 'Add movement x' tab contains a section titled 'Transaction details:' with several input fields: 'Type:' (a dropdown menu), 'User:', 'Money:', 'Date:' (with a calendar icon), and 'Tag:'. Below these fields is an 'Add transaction' button. To the right of the input fields is a 3D bar chart with a blue line graph overlay, showing an upward trend. Below the 'Transaction details' section, there are three rows of balance-related controls. The first row shows 'Balance: 0' and an 'Update' button. The second row shows 'Bal. for range: 0' and two 'Date start' and 'Date end' input fields (each with a calendar icon), followed by a dropdown menu and an 'Update' button. The third row shows 'Trend bal.: 0' and two 'Date start' and 'Date end' input fields (each with a calendar icon), followed by an 'Update' button.

È stata realizzata mediante un Tab Pane con relativi Tab, nella parte superiore si può inserire un movimento riempiendo i relativi campi e prendendo il bottone la transazione viene aggiunta alla lista e scritta sul file.

Ad ogni movimento bisogna aggiornare il bilancio mediante il pulsante 'Update'.

È possibile osservare il bilancio in un dato range di tempo selezionando la data di 'partenza' (che non può essere omessa), e la data di 'arrivo', ed il tipo per vedere le relative entrate/uscite.

Allo stesso modo funziona per trend balance, con la differenza che ritorna il bilancio per un dato range, accompagnato da 'true' o 'false' per indicarne lo stato.

Nella seconda Tab si possono osservare le tabelle, la tabella delle transazioni appare nella seguente maniera.

JBudget				
Add movement Show table x				
Movement Tag				
Type	User	Money	Date	Tag
EARNING	A	1000.0	2020-07-01	SALARIO
EARNING	B	500.0	2020-07-01	INVESTIMENTI
Order by: EARNING Update				

Mentre la tabella dei tag, ogni tag è associato alla sua spesa totale, che dipende dal tipo.

JBudget	
Add movement Show table x	
Movement Tag	
Tag	Money
SPESA	-250.0
BENZINA	-40.0
Type for each tag: EXPENSE Update	

Se si avvia il progetto in modalità Console si avrà una presentazione del tipo:

```
| [0] Exit.  
| [1] Add movement [earning/expense].  
| [2] Print movement list [earning/expense].  
| [3] Return balance.  
| [4] [Expense/Earning] by 'time'.  
| [5] [Expense/Earning] by 'tag'.  
| [6] Budget trend in the time.  
| [7] [Expense/Earning] foreach tag.  
-----  
Option:
```

Dove è possibile scegliere l'opzione desiderata, per eseguire delle operazioni come la (4), (5) e (6) bisogna seguire un determinato tipo di format dato da:

```
[1] Earning by time, [2] Expense by time.  
Format response: [1/2, TIME_START [yyyy-MM-dd], TIME_END [yyyy-MM-dd]]  
<<====<====<=<=====--> 80% EXECUTING [43s]  
EXPENSE in the range [2020-01-01, 2020-12-31]: € -290.0
```

Esempi di input: 2, 2020-01-01, 2020-12-31 oppure 1,2020-01-01,2020-12-31.

Le transazioni in forma tabellare appariranno nella seguente maniera.

Type	User	Money	Date	Tag
EARNING	A	1000.0	2020-07-01	SALARIO
EARNING	B	500.0	2020-07-01	INVESTIMENTI

Durante la registrazione di una transazione non si effettuano controlli sugli input, quindi si potrà incorrere in eccezioni di conversione da String a Double, oppure nell'inserimento manuale della data.

Le classi che costituiscono il programma sono:

InvalidInputException

Eccezione personalizzata lanciata in tutti quei casi in cui ci sia stato un problema con l'input dell'utente, ovvero quando l'opzione scelta non è associata a nessuna scelta del menu, questa eccezione viene lanciata solo se si usa la modalità Console.

MovementType

Enum che ha la responsabilità di identificare il tipo a cui è associata la transazione:

- EXPENSE, rappresenta un'uscita (spesa, utenze, affitto).
- EARNING, rappresenta un'entrata (stipendio, investimenti).

ITransaction

Interfaccia che viene implementata da *Transaction* ed ha la responsabilità di rappresentare una singola transazione.

Ha i seguenti metodi:

- `getUser()`, `setUser(String user)`, che hanno lo scopo di 'settare' l'utente che ha effettuato quella transazione e ritornarlo in futuro.
- `getTag()`, `setTag(String tag)`, 'settano' il tag per cui sono entrate/uscite di soldi, vi è la possibilità di aggiungere un solo tag per transazione con lo scopo di aggiungere più transazioni effettuate nella stessa data, o legate allo stesso concetto (es. viaggio), così da avere un report 'efficiente' di tutte le spese.
- `getMoney()`, `setMoney(double money)`, metodi che rappresentano l'ammontare della 'spesa' della transazione, che può essere entrate o uscita.
- `getDate()`, `setDate()`, metodi che tengono conto della data in cui è stata effettuata la transazione.
- `getType()`, `setType(MovementType type)`, metodi che tengono conto del tipo (entrata/uscita) della transazione.
- `spaceWord(T obj)`, metodo di default che fornisce una spaziatura per avere un corretto allineamento nella stampa della tabelle, prende in input un valore T (generico), di cui misura la lunghezza e ritorna una spaziatura data da 15 (valore predefinito) - `t.length()` = numero di spazi ' ' da aggiungere alla lunghezza della variabile.

Transaction

Sottoclasse che estende *ITransaction* ed effettua l'override dei suoi metodi, ha un unico costruttore che definisce la 'base' della transazione, prendendo come parametri formali (tipo, utente, soldi, data, tag), così se lo si istanzia si crea un oggetto di tipo '*Transaction*'.

In questa classe ho fornito una mia rappresentazione del toString.

1. `toString()`, personalizzato così da facilitare la stampa in forma tabellare della transazione.

IFileManagement

Interfaccia che ha la responsabilità di gestire la 'comunicazione' con il file attraverso due metodi:

1. `write(ITransaction transaction)`, ha la responsabilità di scrivere la singola transazione sul file, ogni volta ne viene registrata una, viene invocato il metodo, così da garantire 'persistenza', la transazione viene salvata in formato Json, per fare ciò ho usato la lib Gson.
2. `read()`, si occupa di leggere l'intero file '`Movement.*`', file dove vengono salvate le transazioni, nel mentre che il file viene letto e la transazione viene riconvertito da Json a Transaction (sempre grazie alla lib Gson), e lo si aggiunge in un ArrayList di *ITransaction*, questo metodo viene passato come parametro formale nel metodo `setList()` di *IBudgetManagement* usato ad ogni avvio del programma così da avere l'ArrayList con tutte le transazioni registrate dal primissimo avvio del programma fino ad oggi.

FileManagement

Classe che gestisce la scrittura/lettura, implementa *IFileManagement*, effettuando l'override dei suoi metodi, si nota che questi possono causare delle Exception, quali *IOException*, segnalando un'eccezione di I/O non riuscita od interrotta, dato che si occupa dell'interazione vera e propria con il file.

IBudgetManagement

Interfaccia che ha la responsabilità di gestire le operazioni legate al budget e tutte le operazioni che lo riguardano, funge da Model di *Transaction*, al suo interno si hanno i seguenti metodi:

- `getBalance()`, metodo che ritorna il bilancio, si usa l'ArrayList caricato all'avvio del programma che contiene tutte le transazioni, e si esegue la relativa somma/sottrazione di essi.
- `insert(ITransaction transaction)`, si occupa di inserire la transazione all'interno dell'ArrayList.
- `trendBalance(LocalDate start, LocalDate end)`, metodo che ritorna una `Function<ArrayList<ITransaction>, Pair<Boolean, Double>>`, ovvero accetta un ArrayList in input e ritorna un `Pair<>`, l'Arraylist viene filtrato per le rispettive date mediante un metodo private che ritorna un `Predicate<ITransaction>`, questo metodo ritorna una coppia di valori, (1) boolean che indica l'andamento, (2) il bilancio di quelle date.
- `balanceForTag(MovementType type, String tag)`, `balanceForDates(MovementType type, LocalDate start, LocalDate end)`, metodi che filtrano rispettivamente per tag o date di cui si vuole sapere la somma delle entrate/uscite.
- `balanceForEachTag(MovementType type)`, ritorna un `HashMap<String, Double>` contenente il tag (String) come chiave, e la somma (Double) delle entrate/uscite ad esso associate.
- `getList()`, `setList(ArrayList<ITransaction> list)` hanno lo scopo di ritornare una lista di `ArrayList<ITransaction>` per effettuare le varie operazioni su di essa, mentre `setList` viene usata per inizializzare la lista all'avvio del programma, gli viene passato il metodo `read()` di *IFileManagement*.

BudgetManagement

Classe che rappresenta il 'Model', definisce un insieme di operazioni contabili che servono a determinare le spese (entrate/uscite) sostenute, implementa l'interfaccia *IBudgetManagement*.

In questa classe sono presenti metodi private che semplificano le operazioni e mantengono il numero delle righe sotto le 20.

Ledger

Classe che fa da ‘Controller’ interponendosi tra il ‘Model’ e la ‘View’, ha la responsabilità di definire la logica dell’applicazione.

Pair

Classe dichiarata ‘final’ con l’unico scopo di ritornare una coppia di tipi generici ‘T’ ed ‘S’, distinti tra loro, o eventualmente uguali, viene usata come tipo di ritorno nel metodo `trendBalance` di *IBudgetManagement*.

View

Interfaccia che ha la responsabilità di garantire il corretto avvio del programma attraverso i suoi due metodi:

1. `open()`, metodo che rappresenta il punto di inizio del programma, deve essere invocato nel ‘main’.
2. `close()`, metodo che termina il programma in maniera corretta (status 0), terminazione avvenuta con successo.

ConsoleView

Classe che implementa *View*, al suo interno vi è una variabile d’istanza di *IBudgetManagement*, con il costruttore che inizializza il tutto, ed i metodi come `menù` che rappresenta le possibili scelte che l’utente può effettuare, `choice()` che prende in input la scelta dell’utente e fa partire il relativo metodo che viene chiamato a sua volta da *IBudgetManagement*, questa classe viene utilizzata nel caso di avvio del programma mediante Console.

App

Classe che contiene il main, ha un costruttore che prende in input un tipo *View*, così da inizializzare la variabile.

Al suo interno sono presenti due metodi, `startConsole()`, e `startGUI()` che fanno rispettivamente partire e garantire l'avvio del programma attraverso il metodo `open()` che chiama la Console, mentre `startGUI` fa partire la classe *GUIView* che implementa il metodo `start` di *Application*.

Può generare due eccezioni che vengono gestite in un try-catch.

GUIView

Classe che gestisce la vista della GUI, implementa l'interfaccia *View* e la classe astratta *Application* di cui bisogna fornire l'implementazione del suo metodo `start`, che rappresenta il punto di inizio.

GUIController

Classe che interagisce da Controller per la GUI, questa classe fa da controller per il file `menu.fxml`, implementa *Initializable*, interfaccia che ha un solo metodo `void` che richiede l'Override, al suo interno viene inizializzata la tabella dei movimenti, una *Image* per 'rallegrare', e tutti i *ChoicheBox* presenti, all'interno della classi si trovano i tag `@FXML` correlati alle relative variabili, in questa classe vengono eseguite tutte le operazioni che modificano i *Label* in base alle azioni che esegue l'utente.

In questa classe è presente un'istanza di *Ledger* che viene inizializzata nel costruttore, così da avere accesso ai metodi per le operazioni contabili.

JUnit Test

Sono stati effettuati dei test sulle classi *BudgetManagement* e *FileManagement*.

BudgetManagementTest

Sono stati effettuati dei test sulla maggior parte dei metodi per garantire il loro corretto funzionamento.

Nei test di `getBalance()`, `balanceForTag()`, `balanceForEachTag()` non si è fatto altro che inserire dei movimenti e verificare che il bilancio generale, bilancio per tag e bilancio per ogni tag corrispondesse.

Nei test `trendBalance()`, `balanceForDates()` sono state inserite delle transazioni con date sovrapposte, per verificare il corretto ritorno del bilancio in quel dato range.

FileManagementTest

In questa classe è stato effettuato un solo test sul metodo `read()` di *FileManagement*, provando a scrivere sul file e vedere il corretto ritorno dell'ArrayList, comparandolo con un ArrayList locale dove man mano venivano aggiunti gli stessi elementi.

Le transazioni per i test che vengono aggiunte nel file non vengono rimosse, ma nel caso di transazioni presenti questo non compromette il successo del test, istanziando un ArrayList con tutti gli elementi già presenti.