



Relazione per la **Quarta Consegna Intermedia** del progetto “**JBudget**” di Programmazione Avanzata

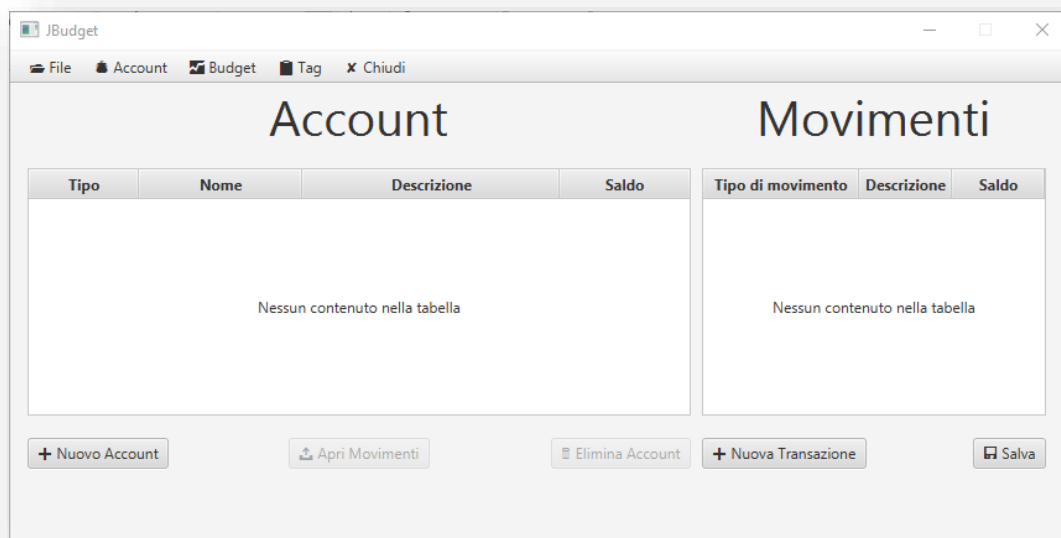
Alex Citeroni (matricola 100625)

☑ **JBudget**: applicazione open source (licenza GPLv2) sviluppata utilizzando un pattern MVC (Model-View-Controller), tramite **Gradle** (sistema open source per l'automazione dello sviluppo che introduce un domain-specific language (DSL) basato su Groovy), **Java 11** (linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica), **Gson 2.8.6** (libreria Java open source per serializzare e deserializzare gli oggetti Java su e da JSON), **JavaFX 14** e **Scene Builder**


Nella directory è inserita anche una cartella 📁 **JavaDoc**

📁 **Salvataggio e** 📁 **Caricamento dei dati**


Il salvataggio e il caricamento dei dati non vengono effettuati automaticamente, ma tramite gli appositi pulsanti “📁 Salva” e “📁 Carica”, posizionati nel menu a tendina “File” in alto a sinistra, nella Home dell'applicazione, dove è possibile scegliere in quale cartella salvare o da dove caricare i file dal proprio pc (tramite il FileChooser)



👁️ Interfaccia grafica

Tramite interfaccia grafica è possibile creare un nuovo Account, visualizzarlo e rimuoverlo, visualizzarne la lista di movimenti, creare dei Tag e dei Movimenti e visualizzarli nella tabella della creazione Transazioni. È anche possibile creare un nuovo Budget o eliminarne uno già creato e vedere il report (i file.xml, utilizzati per le finestre, sono nella cartella  resources)

✓ Avvio Applicazione JBudget

L'applicazione si può avviare tramite IntelliJ facendo click su “ Gradle”, al lato destro dello schermo, poi sulla finestra che si aprirà cliccando in ordine su:

 'jbudget100625' -> Tasks -> application -> run

Inoltre, è possibile avviarla tramite terminale, scrivendo “gradlew run” all'interno della directory del progetto



 it.unicam.cs.pa.jbudget100625

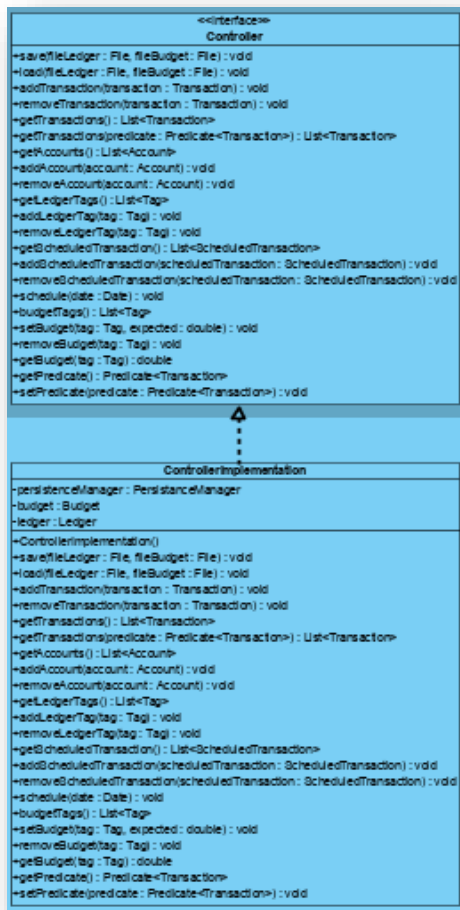
Questi sono i pacchetti contenenti le **classi** e le **interfacce** che fanno parte (per il momento) dell'app **JBudget**

Oltre a queste classi, nella cartella **test** sono contenuti tutti i test con i vari metodi implementati, usati per verificare il funzionamento dei metodi della maggior parte delle classi

MainApp
+main(args : String[]) : void
+start(primaryStage : Stage) : void

🚩 **MainApp:** classe main che permette di avviare l'applicazione (implementa l'interfaccia di JavaFX “**Application**”)

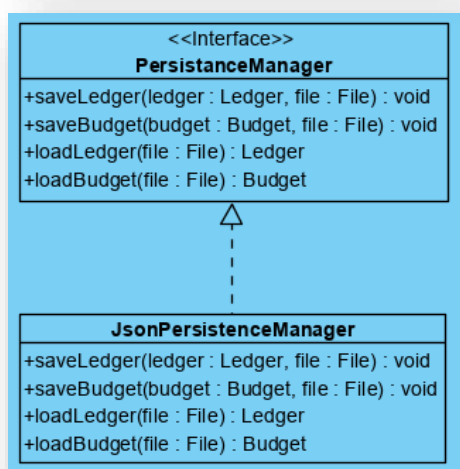
- static void main (String [] args)
 - “launch(args);”: permette il lancio dell'applicazione
- void start(Stage primaryStage): avvia l'interfaccia grafica dell'applicazione facendo riferimento ad un file “home.fxml” che a sua volta si affida ad un controller per la gestione dell'interfaccia grafica (controller contenuti nella cartella View)



📁 Controller

📖 **Controller:** interfaccia che si occupa di fornire tutti i metodi necessari al “ControllerImplementation” per controllare tutto quello che accade nell'applicazione

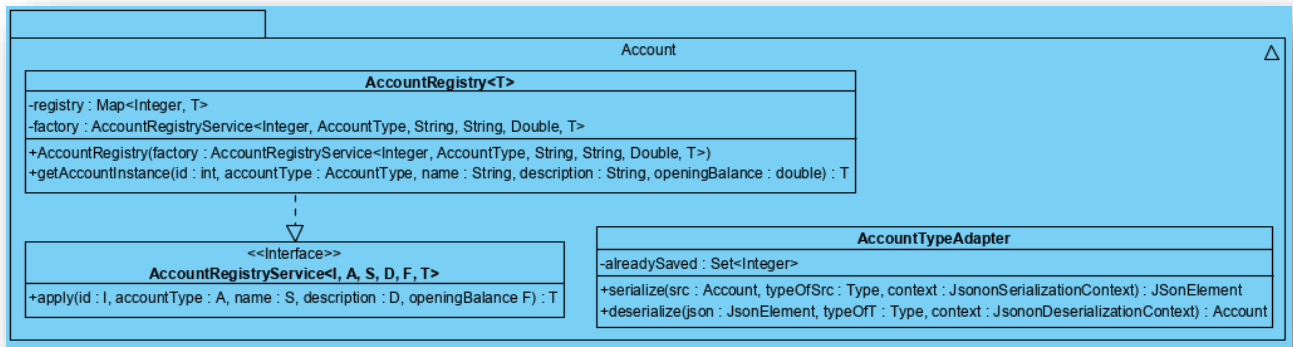
📖 **ControllerImplementation:** classe che implementa l'interfaccia “Controller” e che permette di gestire il Ledger, il Budget e il salvataggio e caricamento dei file e che quindi fa da tramite con la View per permettergli di interagire con il backend



📁 persistence

📖 **PersistenceManager:** questa interfaccia viene implementata da “JsonPersistenceManager” e fornisce i metodi per salvare e caricare file

📖 **JsonPersistenceManager:** classe che implementa “PersistenceManager” e permette di salvare e caricare i file riguardanti il “Ledger” ed il “Budget”

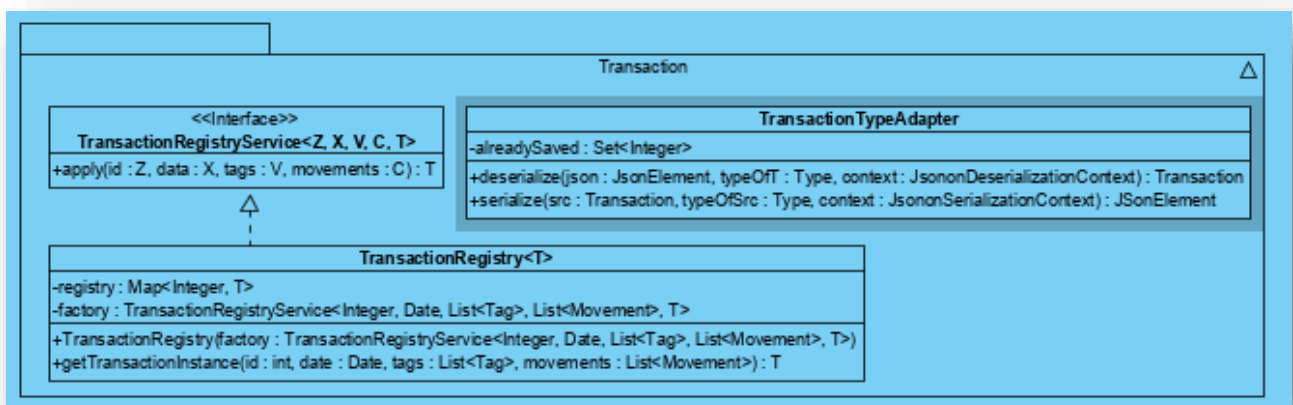


account

AccountRegistryService: interfaccia funzionale che si occupa di creare un metodo "T apply(...);" da utilizzare all'interno della classe "AccountRegistry"

AccountRegistry: classe che crea un registro dedicato agli id degli Account, permettendo di salvarli e verificarne l'unicità

AccountTypeAdapter: questa classe implementa la serializzazione e la deserializzazione rispettivamente di "JsonSerializer<...>" e "JsonDeserializer<...>" ed è utilizzata per evitare dipendenze circolari

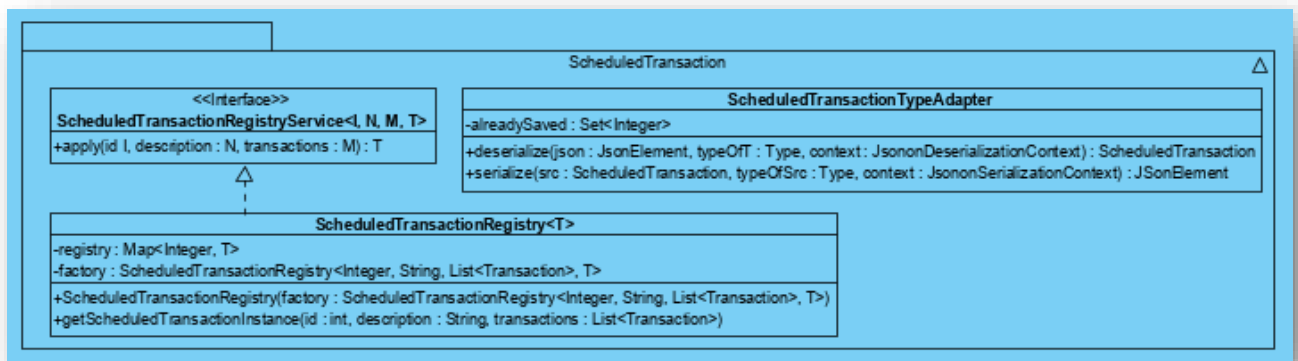


transaction

TransactionRegistryService: interfaccia funzionale che si occupa di creare un metodo "T apply(...);" da utilizzare all'interno della classe "TransactionRegistry"

TransactionRegistry: classe che crea un registro dedicato agli id delle Transaction, permettendo di salvarli e verificarne l'unicità

TransactionTypeAdapter: questa classe implementa la serializzazione e la deserializzazione rispettivamente di "JsonSerializer<...>" e "JsonDeserializer<...>" ed è utilizzata per evitare dipendenze circolari

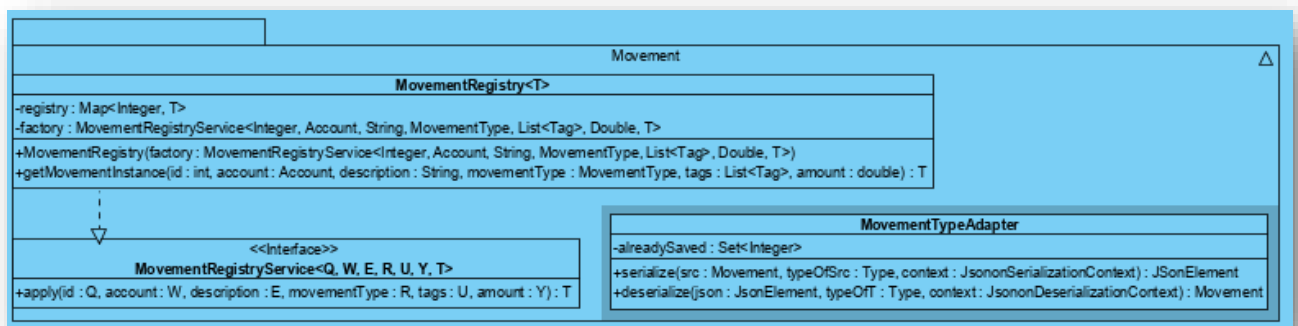


📁 scheduledTransaction

📖 **ScheduledTransactionRegistryService**: interfaccia funzionale che si occupa di creare un metodo "T apply(...);" da utilizzare all'interno della classe "ScheduledTransactionRegistry"

📖 **ScheduledTransactionRegistry**: classe che crea un registro dedicato agli id delle ScheduledTransaction, permettendo di salvarli e verificarne l'unicità

📖 **ScheduledTransactionTypeAdapter**: questa classe implementa la serializzazione e la de-serializzazione rispettivamente di "JsonSerializer<...>" e "JsonDeserializer<...>" ed è utilizzata per evitare dipendenze circolari

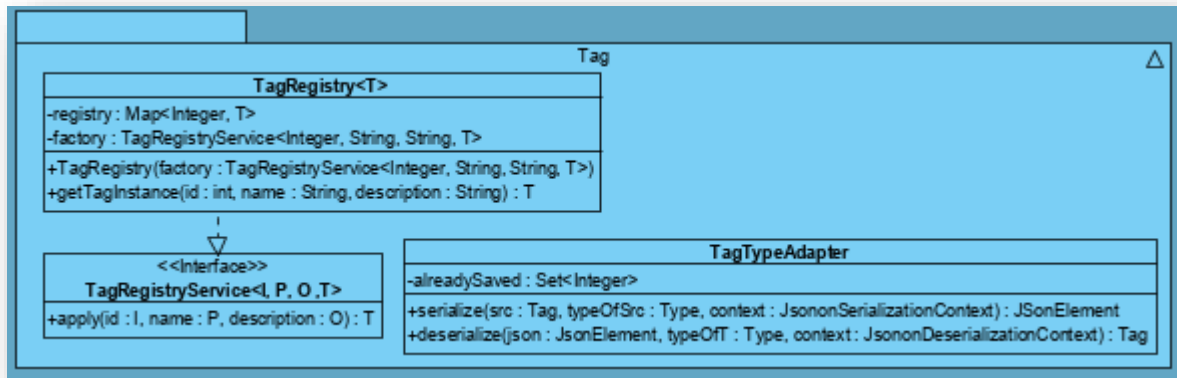


📁 movement

📖 **MovementRegistryService**: interfaccia funzionale che si occupa di creare un metodo "T apply(...);" da utilizzare all'interno della classe "MovementRegistry"

📖 **MovementRegistry**: classe che crea un registro dedicato agli id dei Movement, permettendo di salvarli e verificarne l'unicità

📖 **MovementTypeAdapter**: questa classe implementa la serializzazione e la de-serializzazione rispettivamente di "JsonSerializer<...>" e "JsonDeserializer<...>" ed è utilizzata per evitare dipendenze circolari

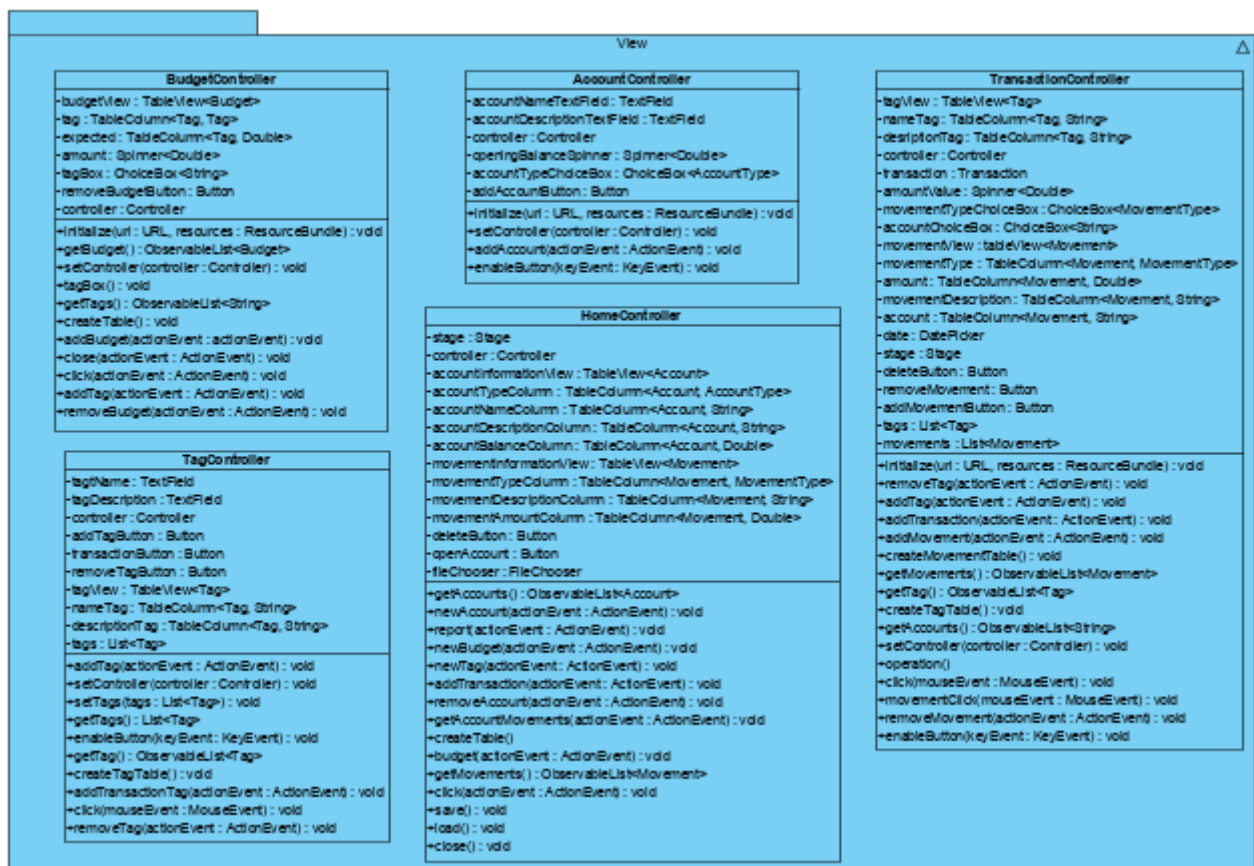


tag


TagRegistryService: interfaccia funzionale che si occupa di creare un metodo "T apply(...);" da utilizzare all'interno della classe "TagRegistry"

TagRegistry: classe che crea un registro dedicato agli id dei Tag, permettendo di salvarli e verificarne l'unicità


TagTypeAdapter: questa classe implementa la serializzazione e la deserializzazione rispettivamente di "JsonSerializer<...>" e "JsonDeserializer<...>" ed è utilizzata per evitare dipendenze circolari





View


 **HomeController:** classe che gestisce la prima finestra dell'applicazione, contenente la tabella degli Account e della lista Movimenti associati ad ogni Account

- Si possono salvare e caricare i dati su dei file.json tramite gli appositi pulsanti dal menu "File" in alto a sinistra
- È possibile creare un nuovo Account andando in alto a sinistra e cliccando "Account" e poi su "Nuovo" e così avviando una nuova finestra che viene controllata da "AccountController", oppure cliccando su "Nuovo Account"
- È possibile selezionare un Account dalla tabella ed eliminarlo tramite il bottone "Elimina Account", o visualizzarne i movimenti tramite il bottone "Visualizza Movimenti"
- È anche possibile creare ed eliminare Tag e Movimenti e visualizzarli tramite la creazione di una nuova Transazione, gestita da "TransactionController"
- Si può visualizzare, creare o rimuovere il Budget

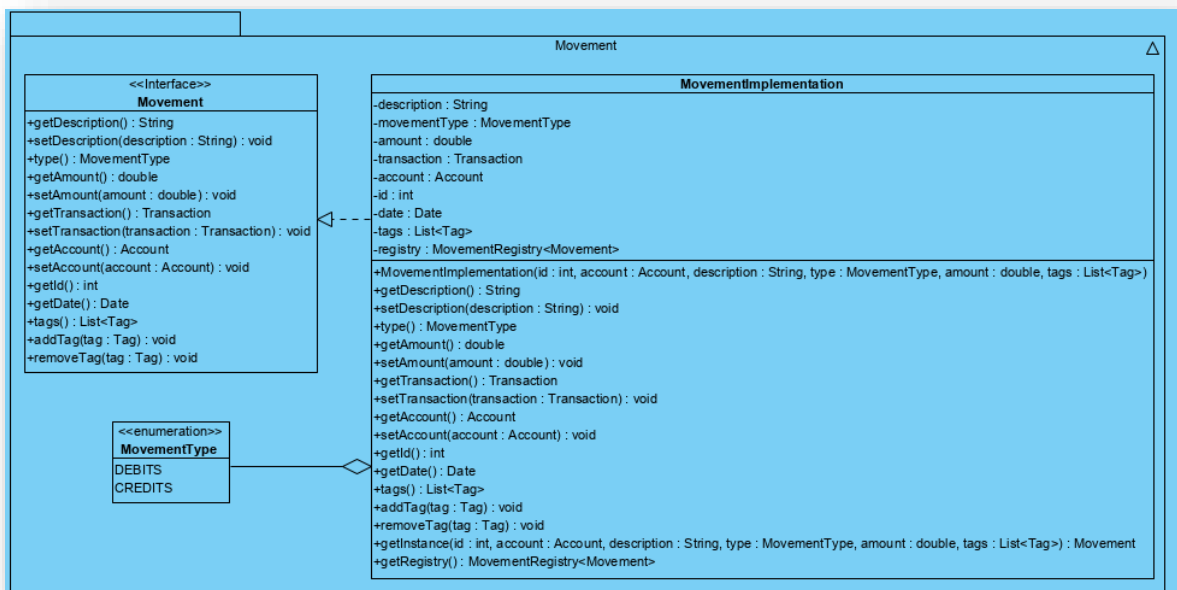
 **AccountController:** questa classe gestisce la creazione di un nuovo Account attraverso l'interfaccia grafica, infatti è possibile crearlo inserendo i dati richiesti (tipo di account, nome, descrizione e saldo d'apertura) e premendo "Aggiungi Account"

 **BudgetController:** classe che implementa l'interfaccia di JavaFX "Initializable" e che gestisce la visualizzazione, la creazione e la rimozione del Budget e del Report attraverso l'interfaccia grafica


 **TagController:** classe che gestisce la creazione e la rimozione di Tag attraverso l'interfaccia grafica, formato da un nome e da una descrizione


 **TransactionController:** classe che implementa l'interfaccia di JavaFX "Initializable" e che gestisce la creazione di Transazioni attraverso l'interfaccia grafica, compresa la scelta di una data, la creazione e la rimozione di Movimenti e di Tag

Model




movement

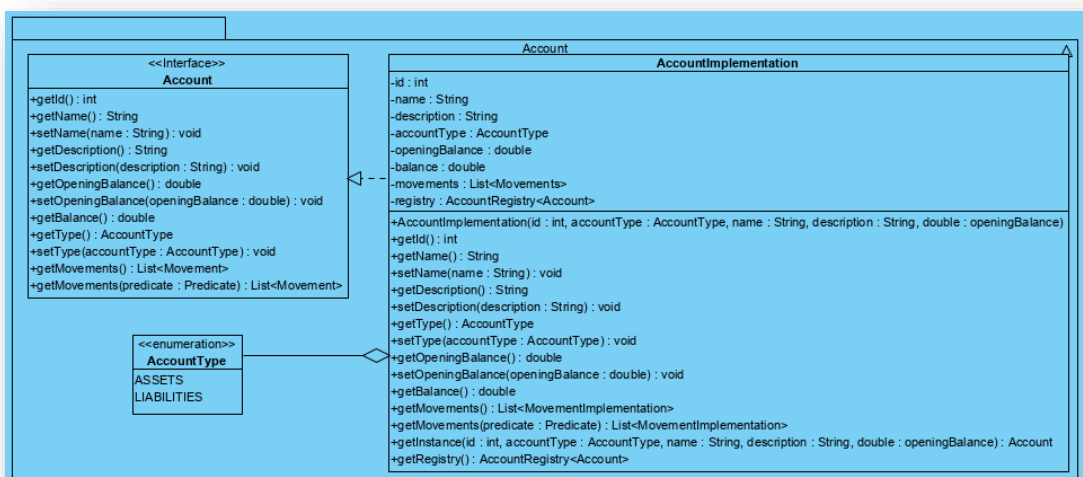
 **Movement:** interfaccia implementata dalla classe “MovementImplementation”, che gestisce un singolo movimento associato ad una transazione e ad una data

 **MovementImplementation:** classe che implementa i metodi riportati in “Movement” e che quindi gestisce un movimento contenente una lista di Tag, un account associato, una descrizione, un tipo di movimento, un’id univoco, un importo e una transazione associata, da cui deriva la data


- MovementRegistry<Movement> getRegistry(): ritorna il registro dei movimenti e se non è presente viene creato
- Movement getInstance(int id, Account account, String description, MovementType movementType, List<Tag> tags, double amount): se è presente, ritorna il movimento contenuto nel registro
- String getDescription(): ritorna la descrizione del movimento
- void setDescription(String description): permette di impostare la descrizione del movimento
- MovementType type(): ritorna il tipo di movimento
- double getAmount(): ritorna l’importo del movimento
- void setAmount(double amount): permette di modificare l’import del”account
- Transaction getTransaction(): ritorna la transazione associata al movimento
- Transaction setTransaction(): permette di impostare la transazione associata al movimento
- Account getAccount(): ritorna l’account associato al movimento
- void setAccount(Account account): permette di impostare un account
- int getId(): ritorna l’id del movimento
- Date getDate(): ritorna la data di esecuzione della transazione
- List<Tag> tags(): ritorna la lista di tag associati al movimento
- void addTag(Tag tag): permette di aggiungere un tag al movimento
- void removeTag(Tag tag): permette di rimuovere un tag dal movimento


 **MovementType:** enumerazione che rappresenta le tipologie di movimento

- **CREDITS** (entrate): all’interno di una transazione saranno trattati come positivi
- **DEBITS** (uscite): all’interno di una transazione saranno trattati come negativi



account

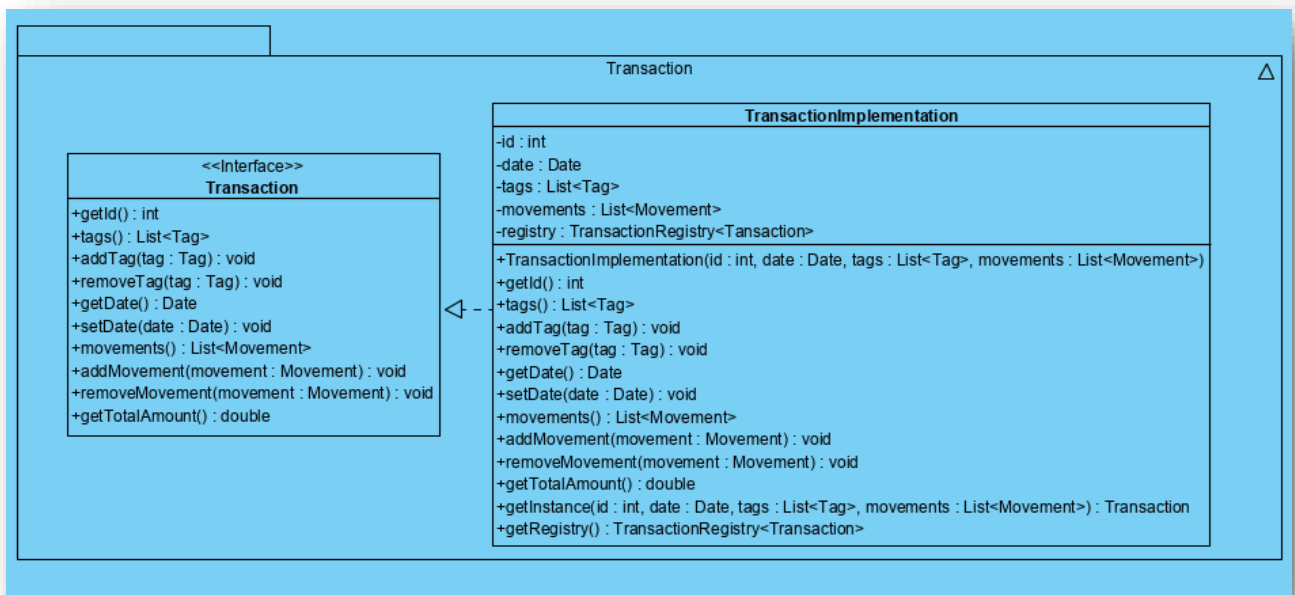
 **Account:** interfaccia implementata dalla classe “AccountImplementation”, che ha la responsabilità di gestire un conto

 **AccountImplementation:** classe che implementa i metodi contenuti in “Account” e che fornisce un modello per creare un nuovo Account composto da: un'id, un nome, una descrizione, una lista di movimenti, un tipo ed un bilancio d'apertura

- `AccountRegistry<Account> getRegistry():` ritorna il registro degli Account e se non è presente viene creato
- `Account getInstance(int id, AccountType accountType, String name, String description, double openingBalance):` se è presente, ritorna l'Account contenuto nel registro
- `String getName():` ritorna il nome dell'account
- `void setName(String name):` permette di impostare il nome dell'account
- `String getDescription():` ritorna la descrizione dell'account
- `void setDescription(String description):` permette di impostare la descrizione dell'account
- `int getId():` ritorna l'id dell'account
- `AccountType getType():` ritorna il tipo dell'account
- `void setType(AccountType accountType):` permette di impostare il tipo di account
- `double getOpeningBalance():` ritorna il saldo di apertura dell'account
- `void setOpeningBalance(double openingBalance):` permette di impostare il saldo d'apertura dell'account
- `double getBalance():` consente di ottenere il saldo attuale dell'account
- `List<Movement> getMovements():` ritorna la lista di movimenti dell'account
- `List<Movement> getMovements(Predicate<Movement> predicate):` ritorna la lista di movimenti dell'account che soddisfano un determinato predicato

 **AccountType:** enumerazione che rappresenta le tipologie di account

- **ASSETS** (risorse): crescerà con movimenti di tipo **CREDITS** e diminuirà con movimenti di tipo **DEBITS**
- **LIABILITIES** (debiti): aumenterà con movimenti di tipo **DEBITS** e diminuirà con movimenti di tipo **CREDITS**

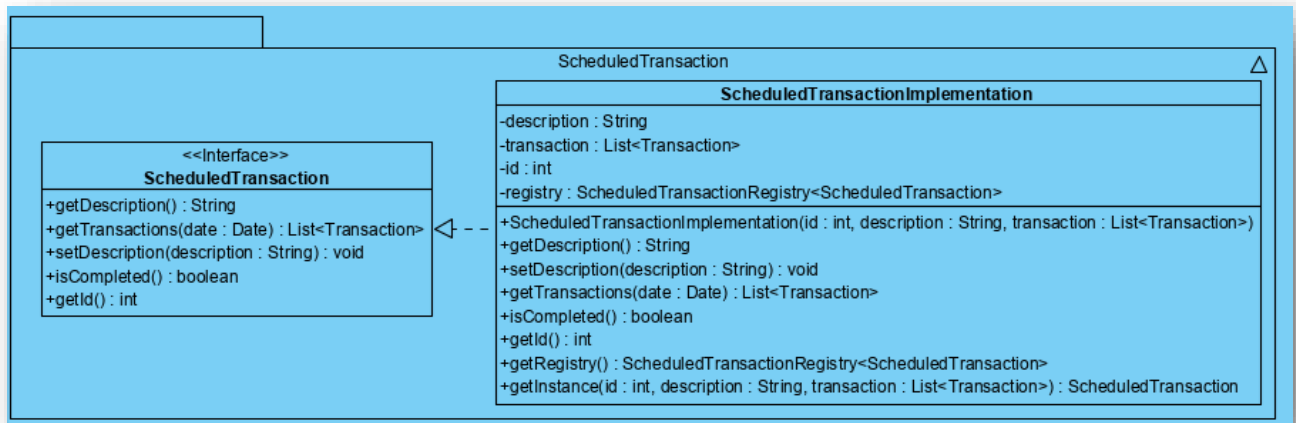


📁 transaction

📖 **Transaction:** interfaccia implementata dalla classe “TransactionImplementation”, che ha la responsabilità di gestire una transazione

📖 **TransactionImplementation:** classe che implementa i metodi dell’interfaccia “Transaction” e che fornisce un modello per creare la transazione, che comprende: un’id, una data, una lista di Tag e una lista di movimenti

- `TransactionRegistry<Transaction> getRegistry():` ritorna il registro delle transazioni e se non è presente viene creato
- `Transaction getInstance(int id, Date date, List<Tag> tags, List<Movement> movements):` se è presente, ritorna la Transazione contenuta nel registro
- `int getId():` ritorna l’id della transazione
- `List<Movement> movements():` ritorna una lista di movimenti della transazione
- `void addMovement(Movement movement):` permette di aggiungere un movimento alla transazione
- `void removeMovement(Movement movement):` permette di rimuovere un movimento dalla transazione
- `List<Tag> tags():` ritorna una lista di tag associati alla transazione
- `void addTag(Tag tag):` permette di aggiungere un tag alla transazione
- `void removeTag(Tag tag):` permette di rimuovere un tag dalla transazione
- `Date getDate():` ritorna la data di esecuzione della transazione
- `void setDate(Date date):` permette di impostare la data della transazione
- `double getTotalAmount():` permette di ottenere il saldo, cioè la variazione totale dei movimenti della transazione

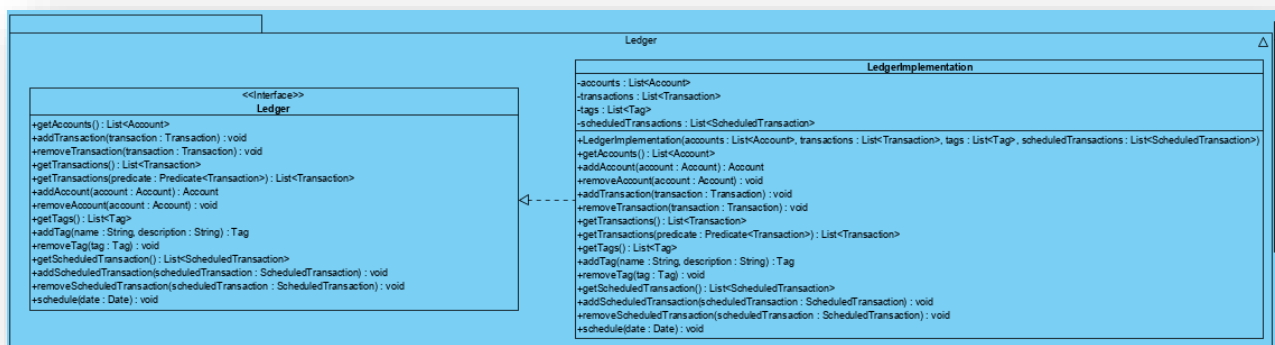


📁 scheduledTransaction


📖 **ScheduledTransaction:** interfaccia implementata dalla classe “ScheduledTransactionImplementation”, che ha la responsabilità di rappresentare una o una serie di transazioni pianificate ad una certa data


📖 **ScheduledTransactionImplementation:** classe che implementa i metodi contenuti in “ScheduledTransaction” e che fornisce un modello per creare una transazione pianificata, formata da: un'id, una descrizione ed una lista di transazioni

- `ScheduledTransactionRegistry<ScheduledTransaction> getRegistry()`: ritorna il registro delle transazioni pianificate e se non è presente viene creato
- `ScheduledTransaction getInstance(int id, String description, List<Transaction> transactions)`: se è presente, ritorna la transazione pianificata contenuta nel registro
- `int getId()`: ritorna l'id della transazione pianificata
- `String getDescription()`: ritorna la descrizione della transazione pianificata
- `void setDescription(String description)`: permette di impostare la descrizione della transazione pianificata
- `List<Transaction> getTransaction(Date date)`: ritorna una lista di transazioni pianificate in base ad una certa data
- `boolean isCompleted()`: ritorna **true** o **false** in base a se è stata completata o meno la serie di transazioni pianificate

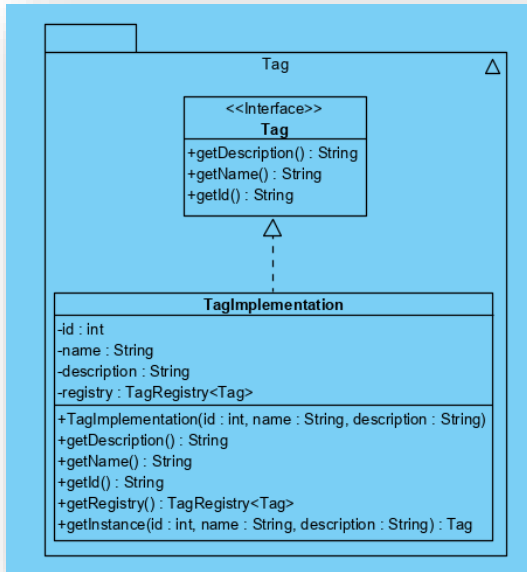


ledger

 **Ledger:** interfaccia implementata dalla classe “LedgerImplementation”, che ha la responsabilità di gestire tutti i dati di: transazioni, transazioni pianificate, tag e account

 **LedgerImplementation:** classe centrale che implementa i metodi dell'interfaccia “Ledger” e che fornisce un modello di Ledger, composto da: una lista di account, una lista di transazioni, una lista di tag ed una lista di transazioni pianificate

- `List<Account> getAccounts():` ritorna la lista degli Account del Ledger
- `void addTransaction(Transaction transaction):` permette di aggiungere una transazione alla lista delle transazioni del Ledger e a quella dell'Account associato al movimento
- `void removeTransaction(Transaction transaction):` permette di rimuovere una transazione dalla lista delle transazioni del Ledger e da quella dell'Account associato al movimento
- `List<Transaction> getTransactions(Transaction transaction):` ritorna la lista di transazioni del Ledger
- `List<Transaction> getTransactions(Predicate <Transaction> predicate):` ritorna la lista di transazioni del Ledger, in base ad un certo predicato
- `List<Tag> getTags():` ritorna la lista di tag del Ledger
- `Account addAccount(Account account):` permette di aggiungere un nuovo account alla lista del Ledger
- `void removeAccount(Account account):` permette di rimuovere un Account dalla lista del Ledger
- `Tag addTag(Tag tag):` permette di aggiungere un tag alla lista di Tag del Ledger
- `void removeTag(Tag tag):` permette di rimuovere il tag dalla lista di Tag del Ledger
- `void addScheduledTransaction(ScheduledTransaction scheduledTransaction):` permette di aggiungere una transazione pianificata alla list di ScheduledTransaction, contenuta nel Ledger
- `void removeScheduledTransaction(ScheduledTransaction scheduledTransaction):` permette di rimuovere una transazione pianificata dalla lista di ScheduledTransaction, contenuta nel Ledger
- `List<ScheduledTransaction> getScheduledTransaction():` ritorna la lista di transazioni pianificate contenuta nel Ledger
- `void schedule(Date date):` permette di pianificare una transazione per una certa data e quindi creare una transazione pianificata

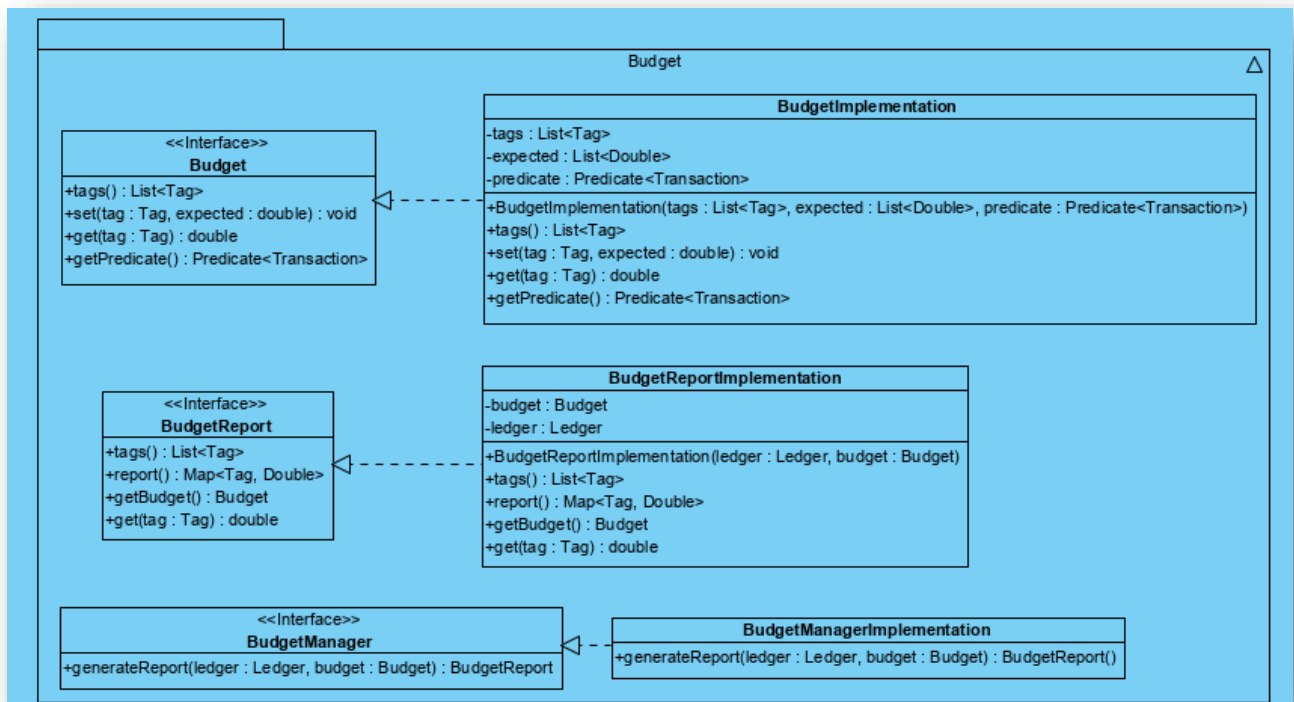


tag

Tag: interfaccia implementata da “TagImplementation”, che ha la responsabilità di definire una categoria di spesa o di guadagno

TagImplementation: classe che implementa i metodi dell'interfaccia “Tag” e che fornisce un modello composto da un nome ed una descrizione

- `TagRegistry<Tag> getRegistry()`: ritorna il registro dei tag e se non è presente viene creato
- `Tag getInstance(int id, String name, String description)`: se è presente, ritorna il tag contenuto nel registro
- `String getDescription()`: ritorna la descrizione del tag
- `String getName()`: ritorna il nome del tag
- `int getId()`: ritorna l'id del tag





budget

Budget: interfaccia implementata dalla classe “BudgetImplementation”, che ha la responsabilità di rappresentare e gestire un particolare budget


BudgetImplementation: classe che implementa i metodi della classe “Budget” ed ha la responsabilità di associare ad ogni tag un importo che indica l'ammontare di spesa o di guadagno per il particolare tag e di costruire il predicato per selezionare i movimenti di interesse


- `List<Tag> tags()`: ritorna una lista di tag
- `void set(Tag tag, double expected)`: permette di impostare il budget attraverso il tag e il valore atteso
- `double get(Tag tag)`: ritorna il valore atteso in base allo specifico tag
- `Predicate<Transaction> getPredicate()`: ritorna il predicato
- `void setPredicate(Predicate<Transaction> predicate)`: permette di impostare il predicato

 **BudgetReport**: interfaccia implementata da “BudgetReportImplementation”, che ha la responsabilità di mostrare gli scostamenti di spesa o di guadagno rispetto ad un particolare budget

 **BudgetReportImplementation**: classe che implementa i metodi dell'interfaccia “BudgetReport” e che definisce i criteri per la selezione dei movimenti

- `List<Tag> tags()`: ritorna una lista di tag
- `Map<Tag, Double> report()`: ritorna un report sottoforma di mappa, che associa ai tag i valori
- `Budget getBudget()`: ritorna il budget
- `double get(Tag tag)`: ritorna il valore atteso in base allo specifico tag

 **BudgetManager**: interfaccia implementata da “BudgetManagerImplementation”, che ha la responsabilità di costruire il BudgetReport associato ad un Budget e ad un Ledger

 **BudgetManagerImplementation**: classe che implementa l'interfaccia “BudgetManager” e ha un solo metodo che permette di generare un report

- `BudgetReport generateReport(Ledger ledger, Budget budget)`: permette di generare il report del budget



Possibili sviluppi futuri

Purtroppo, in questa consegna il tempo a disposizione era molto limitato e non sono stato in grado fare tutto quello che avevo in mente...

Queste sono alcune delle idee che avrei voluto implementare:

- ❖ Grafici e statistiche riguardanti i dati presenti nell'app
- ❖ Utilizzo dell'applicazione tramite web e mobile, tramite l'utilizzo di un sito
- ❖ Inserimento di profili utenti, con la possibilità per ogni utente di gestire il proprio budget ovunque (tramite l'utilizzo di database, per l'eventuale grande mole di dati)
- ❖ Impostare un metodo di pagamento e pagare direttamente attraverso l'app per transazioni pianificate