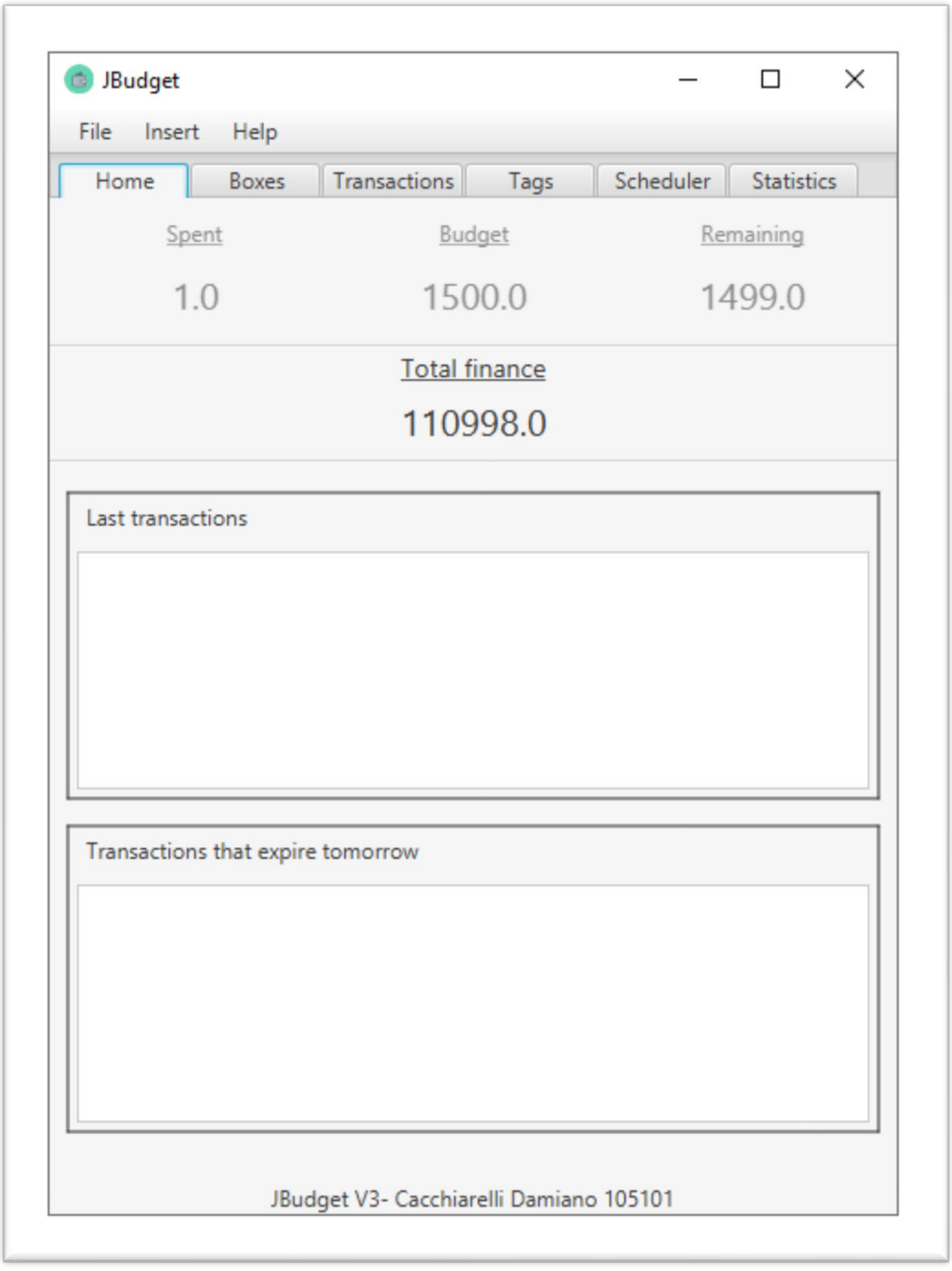
JBudget

Damiano Cacchiarelli 105101

Programmazione Avanzata

Corso di Laurea in Informatica

Anno 2019/2020



Introduzione

JBudget - Damiano Cacchiarelli 105101

Per la realizzazione di un sistema di gestione delle finanze familiari sono stati individuati due concetti fondamentali:

- *Conto* : rappresenta un gestore in cui si tiene traccia di movimenti contabili. Possiede un proprio bilancio e permette di ricevere o inviare denaro. Possono essere di vari tipi ognuno con compiti diversi;
- *Transazione* : Rappresenta un singolo movimento di denaro che piò essere di uscita o entrata da un *conto*.

È stato utilizzato il design Pattern Model-View-Controller (MVC) per la struttura base del progetto. Come si intuisce facilmente dal nome, tale pattern è costituito da 3 strati:

- *Model* : è il componente centrale del pattern che gestisce i dati dell'applicazione;

- *View* : rappresenta la visualizzazione dei dati contenuti nel model e permette all'utente di interagire con l'applicazione;

- *Controller* : elabora gli input provenienti dalla *view* per convertirli in comandi/dati per il *model*.

Di seguito sono riportate le *Interfacce* e le *Classi* implementate con la relativa descrizione delle responsabilità.

Descrizione delle responsabilità

JBudget - Damiano Cacchiarelli 105101

Lista delle interfacce:

Modello

- Bank;
- CashBox;
- Transaction;
- Category;
- CategoryMannager;
- IdentificationManager;
- Identifier;

Controllo

- Controller;
- StatisticOfBank;
- SchedulerTransaction;
- MasterScheduler;
- Synchronizer;

Vista

- View

Lista delle classi:

Modello

- FamilyBank;
- Box;
- Wallet;
- LoadBox;
- SaveBox;
- DebitBox;
- BasicTransaction;
- Tag;
- TagManager;
- IntegerIdManager;
- IntegerId;

Controllo

- FamilyBankController;
- BasicStatisticOfBank
- MasterSchedulerBasicTransaction;
- TemporalScheduler;
- LocalSynchronizer;
- InterfaceAdapterGson;

Vista

- ViewJavaFXJBudget;

Category e Tag

Le classi che implementeranno l'interfaccia *Category* avranno la responsabilità di rappresentare una categoria di cui un elemento può far parte.

L'implementazione di questa interfaccia è *Tag*: questa classe permetterà, oltre a determinare quella che sarà la categoria di una determinata transazione, di definire una sua gerarchia basata su diversi livelli di priorità. L'unica condizione imposta sulla priorità è che essa non può essere rappresentata da un numero minore di zero.

La classe sovrascrive il metodo *equals()* e il metodo *HashCode()*: due tag vengono considerati uguali se hanno lo stesso nome.

```
public interface Category { }
```

CategoryManager e TagManager

Le classi che implementeranno l'interfaccia *CategoryManager* avranno la responsabilità di gestire degli oggetti di tipo Category. Dovranno implementare metodi per aggiungere, rimuovere i vari elementi all'interno dell'insieme.

```
public interface CategoryManager { }
```

L'implementazione di questa interfaccia è *TagManager:* essa gestisce l'insieme dei tag tramite un HashSet, quindi non sarà possibile aggiungere due tag uguali. Un *TagManager* permetterà inoltre di accedere e ottenere gli i tag partendo dal nome del tag stesso;

Transaction e BasicTransaction

Le classi che implementeranno *Transaction* avranno la responsabilità di implementare quelle che sono le caratteristiche di una transazione.

```
public interface Transaction { }
```

La mia implementazione dell'interfaccia è *BasicTransaction*: una BasicTransaction accetta sia valori positivi che negativi (può essere sia un'entrata che una spesa); Inoltre ogni transizione potrà essere "attiva" o "non attiva", ovvero una transazione è attiva se non è ancora stata contabilizzata, non attiva altrimenti. Per cambiare il suo stato la classe mette a disposizione il metodo *toExecute() : void*. Una transazione inoltre sarà caratterizzata da una data, da una descrizione, da un tipo, definito dall'enumerazione *TypeTransaction*, da uno o più *Tag* e da un oggetto *Identifier*. La classe sovrascrive il metodo *equals()* e il metodo *HashCode()*: due transazioni vengono considerati uguali se hanno la stessa descrizione, la stessa data, lo stesso importo, la stessa lista di *Tag* e lo stesso *Identifier*.

Identifier e IntegerId

Le classi che implementeranno *Identifier* avranno la responsabilità di creare un codice identificativo. Le classi inoltre dovranno implementare un metodo per ottenere il codice e un metodo per ottenere la sua rappresentazione sotto forma di stringa. È accettato qualsiasi tipo generico <T>.

La mia implementazione dell'interfaccia è *IntegerId* che crea un codice di tipo Integer. La classe sovrascrive il metodo *equals()* e *HashCode()* e definisce che due istanze della classe sono uguali se hanno lo stesso valore.

```
public interface Identifier<T> { }
```

IdentificationManager e IntegerIdManager

Le classi che implementeranno *IdentificationManager* avranno la responsabilità di gestire e distribuire dei *Identifier*. Inoltre dovranno permettere di poter verificare se un determinato codice è stato già distribuito.

```
public interface IdentificationManager<T extends Identifier<?>> {  }
```

L'implementazione *IntegerIdManager* permette di gestire degli *Identifier* di tipo *IntegerId*. Mette a disposizione un meccanismo che permette di ottenere un *IntegerId* che potrà essere sia un nuovo codice sia uno restituito in precedenza.

CashBox e Box, LoadBox, Wallet, SaveBox, DebitBox

Le classi che implementeranno *CashBox* avranno la responsabilità di implementare quelli che poi saranno dei "contenitori di soldi". Lo scopo di queti box sarà quello di tenere traccia delle varie transazioni che possono essere aggiunte al loro interno.

```
public interface CashBox {  }
```

La mia implementazione dell'interfaccia prevede la classe astratta *Box*, le sue classi figlie *Wallet* e la classe astratta *LoadBox* che a sua volta è padre delle classi *SaveBox* e *DebitBox*. Ogni box ha la possibilità di gestire una lista di Transazioni, le quali potranno essere aggiunte o eliminate, memorizzando il bilancio, i soldi spesi e quelli guadagnati. In più, le classi figlie di *LoadBox* avranno la possibilità di definire una certa soglia di denaro da raggiungere. In particolare un *Savebox* definisce un limite massimo del box (*goal*) che dovrà essere raggiunto e potrà quindi essere usato per risparmiare una determinata quantità di soldi (come se fosse un vero e proprio salvadanaio). Un *DebitBox* invece, definisce un debito che dovrà essere saldato.

Bank e FamilyBank

Le classi che implementeranno *Bank* avranno la responsabilità di implementare un gestore di *CashBox*. Lo scopo di una bank sarà quello di tenere traccia di più box.

```
public interface Bank {  }
```

La mia implementazione dell'interfaccia è la classe *FamilyBank*. Una FamilyBank si occupa di gestire una lista di CashBox memorizzando un bilancio, i soldi spesi e quelli guadagnati; inoltre sarà possibile impostare un determinato budget che rappresenta la massima quantità di denaro che l'utente che l'utente vuole cercare di spendere ogni mese; tale budget non è vincolante: sarà infatti possibile spendere più di quanto previsto, ovviamente rispettando i vincoli dei *CashBox*.

Controller e FamilyBankController

Le classi che implementeranno *Controller* avranno la responsabilità di eseguire determinate operazioni sulle classi del MODEL.

L'implementazione *FamilyBankController* definisce delle operazioni "base" come aggiungere e rimuovere tag, transazioni e box oppure operazioni come la ricerca di transazioni con determinate caratteristiche. Per altri tipi di operazioni si appoggia alle altre classi del CONTROLLER come *BasicStatisticOfBank* o *TimerScheduler*.

```
public interface Controller {  }
```

StatisticOfBank e BasicStatisticOfBank

Le classi che implementeranno *StatisticOdBank* avranno la responsabilità di eseguire delle operazioni di carattere statistico sulle classi del MODEL. La mia implementazione, *BasicStatisticOfBank* mette a disposizione alcune operazioni di questo tipo: permette di conoscere, ad esempio la spesa totale di un determinato tag o il numero di transazioni eseguite in un determinato giorno. Può essere utilizzato qualsiasi tipo generico <T> che estende l'interfaccia *Bank*.

```
public interface StatisticsOfBank<T extends Bank> {  }
```

SchedulerTransaction e TemporalScheduler

Le classi che implementeranno *SchedulerTransaction* avranno la responsabilità di generare delle transazioni programmate.

In particolare viene richiesto di implementare un metodo che permetta di generare una lista di transazioni partendo da una di esempio e seguendo un determinato schema.

Viene inoltre chiesto di implementare un metodo che permetta di generare due liste in cui una delle due ha transazioni con importi opposti all'altra lista in modo tale che sia rappresentato uno spostamento di soldi tra due *CashBox*.

Ogni *SchedulerTransaction* dovrà essere caratterizzato da un tipo che ne descriva il funzionamento.

```
public interface SchedulerTransaction<T extends Transaction> {  }
```

L'implementazione *TemporalScheduler*, in particolare ha la funzionalità di generare transazioni uguali e ripeterle una volta ogni periodo di tempo definito da una *BiFunction<Double, LocalDate, LocalDate>*. Il *TemporalScheduler* è definito dall'enumerazione *TEMPORAL* e mette a disposizione una descrizione delle sue funzionalità.

MasterScheduler e MasterSchedulerBasicTransaction

Le classi che implementeranno *MasterScheduler* avranno la responsabilità di gestire tutti gli *Scheduler*. Dovranno implementare funzionalità che permetteranno di aggiungere e rimuovere *Scheduler* e di restituire lo *Scheduler* corrispondente al TypeScheduler tipo richiesto.

Ogni *Scheduler* che il manager gestisce dovrà avere tipo diverso.

La mia implementazione dell'interfaccia è *MasterSchedulerBasicTransaction*.

```
public interface MasterScheduler<T extends Transaction> {  }
```

Synchronizer e LocalSynchronizer

Le classi che implementeranno *Synchronizer* avranno la responsabilità di caricare, salvare, e sincronizzare. Devono essere definiti i tre parametri <B>, <C>, <I> che dovranno rispettare queste dipendenze: <B **extends** Bank>, <C **extends** CategoryManager>, <I **extends** IdentificationManager<?>>.

La mia implementazione è *LocalSynchronizer* che permette di salvare e caricare lo stato dell'applicazione per mezzo di tre file .json che rappresentato tre istanze delle classi : *FamilyBank*, *CategoryManager* e *IntegerIdManager*. Viene utilizzata la libreria *Gson* che permette di convertire oggetti Java nella loro rappresentazione JSON, ma anche convertire una stringa JSON in un oggetto JAVA equivalente. *Per* la sincronizzazione dei dati tra più dispositivi non mette a disposizione un metodo automatico ma si dovrà procedere manualmente salvando e ricaricando i dati ogni qual volta avviene una modifica dello stato.

View e ViewJavaFXJBudget

Le classi che implementeranno *View* avranno la responsabilità di ricevere dati in input dall'utente e di passarli al CONTROLLER e di visualizzare lo stato del MODEL. La mia implementazione *ViewJavaFXJBudget* utilizza JavaFX e dei file XML per interagire con l'utente. In particolare abbiamo:

- *JBudget.fxml*, che rappresenta la vista principale;
- *AboutMenu.fxml*, che permette di visualizzare informazioni dell'applicazione;
- *AddWallet.fxml*, che permette all'utente di inserire i dati per creare un nuovo Wallet;
- *AddSaveBox.fxml*, che permette all'utente di inserire i dati per creare un SaveBox;
- *AddDebitBox.fxml*, che permette all'utente di inserire i dati per creare un DebitBox;
- *AddTransaction.fxml*, che permette all'utente di inserire i dati per una transazione;
- *AddTag.fxml*, che permette all'utente di inserire i dati per creare un nuovo Wallet;
- *SetBudger.fxml*, che permette all'utente di inserire i dati per impostare un budget;
- *TemporalScheduler.fxml,* che permette all'utente di usufruire delle funzionalità dello scheduler.

Ad ognuna di queste "viste" è associato un determinato controller che gestisce i loro comportamenti; Abbiamo rispettivamente: *JavaFXJBudget*, *JavaFXAboutMenu.fxml*, *JavaFXAddWallet*, *JavaFXAddSaveBox*, *JavaFXAddDebitBox*, *JavaFXAddTransaction*, *JavaFXAddTag*, *JavaFXSetBudgetController* e *JavaFXTimerSchedulerController*. Tutti questi controller sono figli della classe *JavaFXStageController*. Altre "viste" e altri controller potranno essere aggiunti per permettere all'utente di eseguire un numero maggiore di interazioni con l'interfaccia.

Enumerazioni ed Eccezioni

Sono implementate le seguenti enumerazioni:

- *TypeTransaction* : descrive il tipo di transazione. Sono stati definiti due tipi di transazione: *DEBIT* che descrive una transazione con importo negativo; *CREDIT* che descrive una transazione con importo positivo.
- *TypeCashBox* : descrive il tipo di *CashBox*. Sono stati definiti tre tipi di box: *WALLET*, *SAVEBOX*, *DEBITBOX*.
- *TypeScheduler* : descrive il tipo di *Scheduler*. È stato definito un tipo di scheduler: *TEMPORAL*.

È stata implementata la seguente eccezione:

- *ExistingElementException*, che viene lanciata quando un elemento è già presente.

Funzionalità implementate

JBudget - Damiano Cacchiarelli 105101

Sono state implementate le seguenti funzionalità :

- Possibilità di creare ed eliminare Tag, Transazioni, Wallet, SaveBox e DebitBox;
- Di un Tag è possibile inserire una priorità compresa tra 1 e 5 anche se la classe non definisce alcun limite, se non quello per il quale il numero non dovrà essere minore di zero;
- Di una transazione è possibile definire valori sia positivi che negativi ed è possibile inserire transazioni sia passate che future: queste verranno eseguite in modo autonomo il giorno della data inserita dall'utente. Ad una transazione è possibile assegnare uno o più tag;
- Non è possibile modificare né i tag né le transazioni una volta creati (Operazioni che possono essere facilmente implementabili);
- È possibile per l'utente cercare transazioni modificando i tre parametri *CashBox*, *Tag*, *Data*. È possibile inoltre conoscere il numero di transazioni effettuate nei sei giorni precedenti, la quantità di denaro spesa nei sei mesi precedenti e la percentuale di denaro speso per le diverse categorie in relazione alla spesa totale.
- Un *Wallet* rappresenta un asset da cui è possibile prelevare o accreditare del denaro tramite l'aggiunta di transazioni. Non è possibile prelevare più denaro di quello che si possiede.
- Un *SaveBox* ha le stesse possibilità di un *Wallet* ma in più ha la possibilità di definire un obbiettivo, al quale raggiungimento il Box sarà pieno e non sarà più possibile aggiungere o rimuovere transazioni;
- Un *DebitBox* permette invece di definire un debito e di tenere traccia del suo stato, quindi sapere per esempio quanti soldi sono rimasti per saldarlo. Non mette a disposizione alcun metodo che permetta, definita la quantità di denaro da restituire, di calcolare in modo autonomo le varie rate e quindi di creare in modo autonomo transazioni. È possibile tuttavia per l'utente utilizzare uno *Scheduler* che permette di generare una lista di transazioni.
- È disponibile per l'utente un solo tipo di *Scheduler* che offre il seguente servizio: data una transazione di esempio vengono create delle transazioni ognuna ad una distanza di giorni dalla precedente, partendo dalla data della transazione di riferimento, tutte con lo stesso identico valore (Per ora è possibile scegliere di ripetere la transazione ogni giorno, una volta alla settimana o una volta ogni mese). È possibile tuttavia implementare scheduler con servizi più complessi e di diverso genere. Un possibile scheduler potrebbe essere quello che dato un importo di un prestito, la durata in mesi e il tasso di interesse calcoli in modo automatico le rate e aggiunge le relative transazioni.
- Lo *Scheduler* può essere utilizzato su tutti i tipi di Box con le relative limitazioni.

- Il CONTROLLER mette a disposizione dell'utente la possibilità di spostare una determinata quantità di soldi da un Box ad un altro. Questo è possibile definendo il box da dove prendere i soldi, il box di destinazione e l'importo; il CONTROLLER in automatico genererà due transazioni una che rappresenterà un movimento in uscita che andrà inserito nel primo box e l'altra che invece rappresenta un movimento in entrata e verrà inserito nel secondo. Per ora questa operazione di "*Shift*" può essere eseguita solo tra due *Wallet* ma potrebbe essere facilmente estesa alle altre tipologie di Box.

Sono state implementate le seguenti funzionalità statistiche:

- Determinare quanto si è speso in un particolare mese;
- Determinare quanto si è speso per un tag;
- Determinare quanto si è speso in un particolare giorno;
- Determinare il numero di transazioni eseguite in un giorno;

Queste funzionalità sono state implementate nella vista con qualche limitazione. Per esempio non è possibile visualizzare il numero di transazioni effettuate in un giorno se da questa data sono trascorsi più di sette giorni; oppure non è possibile visualizzare il denaro speso in un mese se sono trascorsi più di sei mesi da quel periodo.

Test sviluppati

JBudget - Damiano Cacchiarelli 105101

Sono stati sviluppati i seguenti test:

- *BasicTransactionTest* : effettua test sulla classe *BasicTransaction*. In particolare verifica che: non sia possibile creare transazioni con valori *null* e che i metodi per l'ottenimento delle informazioni siano implementati correttamente.
- *WalletTestTest* : effettua test sulla classe *Wallet*. In particolare: verifica che non sia possibile creare Wallet con valori negativi o *null*; verifica che non sia possibile inserire transazioni senza *Tag* o transazioni con un valore tale da portare il Wallet ad un bilancio negativo; verifica che non sia possibile rimuovere una transazione se il bilancio non lo permette; verifica che non sia possibile aggiungere due transazioni uguali; verifica il metodo *checkTransaction* che ha il compito di stabilire se quella transazione può essere eseguita in quel box; verifica il metodo *UpdateBalance* che ha il compito di aggiornare lo stato del box;

- *SaveBoxTest* : effettua test sulla classe *SaveBox*. In particolare: verifica che non sia possibile creare SaveBox con valori *null* oppure con goal negativo o minore del bilancio; verifica che non sia possibile aggiungere o rimuovere Transazioni dopo che si è raggiunto il traguardo; verifica che non sia possibile sforare quello che è il limite definito dal *goal*; verifica il metodo *checkTransaction* che ha il compito di stabilire se quella transazione può essere eseguita, ovvero verifica che dopo aver sommato il valore della transazione con il bilancio del box esso sia comunque compreso tra zero e il *goal*.
- *DebitBoxTest* : effettua test sulla classe *DebitBox*. In particolare: verifica che non sia possibile creare DebitBox con valori positivi o con valori *null*; verifica che non sia possibile aggiungere transazioni di tipo *DEBITO*.
- *FamilyBankTest* : effettua test sulla classe *FamilyBank*. In particolare : verifica che non sia possibile creare una bank con un budget minore di zero; verifica che non sia possibile aggiungere box uguali; verifica che non sia possibile rimuovere box che non sono presenti nella bank; verifica il metodo *UpdateBalance* che ha il compito di aggiornare il bilancio dell'intera bank;
- *BasicStatisticOfBankTest* : effettua test sulla classe *BasicStatisticOfBank*. In generale vengono verificati se i vari metodi restituiscono valori coerenti con lo stato della bank.
- *FamilyBankControllerTest* : effettua test sulla classe *FamilyBankController*. In particolare viene verificato che l'operazione di *shift* venga effettuata correttamente; viene verificato se il metodo per la ricerca delle transazioni sia implementato in modo corretto; verifica se l'operazione di schedulazione viene effettuata correttamente.

Estendibilità del codice

Il codice mette a disposizione meccanismi per essere esteso in modo semplice.

Per quanto riguarda le classi del model è possibile personalizzare diversi elementi. Ad esempio :

- *Codice identificativo* : è possibile definire un codice identificativo più complesso dell'attuale *IntegerId*. La nuova classe che descriverà il codice dovrà implementare l'interfaccia *Identifier* che descrive il contratto che tutti i codici identificativi devono rispettare;
- *Box per le transazioni* : è possibile estendere le attuali classi o implementarne di nuove per definire box con funzionalità diverse. Queste classi dovranno implementare l'interfaccia *CashBox* che descrive il contratto che tutti i box devono rispettare.

Per le quanto riguarda le classi del controller, ad esempio è possibile estendere :

- *Scheduler* : è possibile definire nuovi Scheduler che implementano funzionalità diverse. Ne è stato dato un esempio precedentemente. Per fare questo viene richiesto che il nuovo Scheduler implementi l'interfaccia *SchedulerTransaction* che descrive il contratto che tutti gli Scheduler devono rispettare e che abbia un *tag* diverso da tutti gli altri Scheduler. Una volta creato basterà inserirlo all'interno del gestore di Scheduler;
- *Synchronizer*: è stato implementato un sincronizzatore che permette di salvare localmente i dati dell'applicazione; naturalmente è possibile definirne uno nuovo, magari che implementa un servizio client-server e permette di condividere i dati su più dispositivi. Per fare ciò la nuova classe deve implementare l'interfaccia *Synchronizer* che descrive il contratto che tutti sincronizzatori devono rispettare.

Per quanto riguarda le classi della vista, è stata implementata un vista grafica, ma è possibile aggiungere ulteriori interfacce e personalizzare le loro funzionalità : la nuova classe deve implementare l'interfaccia *View* che definisce il contratto che tutte le interfacce devono rispettare.

Diagramma UML

JBudget -

Damiano Cacchiarelli 105101

- Emumerazioni
- Modello
- Controllo
- Vista
- App