11/28/2017

Steven Barry Lauren Higgins Ioan Luca Cameron Lamberth Chadha Degachi

T06-Foliotracker

Final Design

Table of Contents

[1. Overall Design rational 2](#_Toc499662447)

[2. Changes to Initial design 2](#_Toc499662448)

[3. Back-End API Design 2](#_Toc499662449)

[4. Relationships between Front-End and Back-End: 3](#_Toc499662450)

[5. Method Specifications 4](#_Toc499662451)

[6. Assertions used 6](#_Toc499662452)

[7. Rationale of Junit Tests used 7](#_Toc499662453)

[8. Class Diagram 8](#_Toc499662454)

# Overall Design rational

Our completed system implements the MVC (Model-View-Controller) design pattern, to keep the application decoupled and communicate through the controller class as an intermediary between the view and the model. We have also implemented a folio tracker driver class to initialise the system through the main method, which is outside of the three main packages. The view package of the system consists of the code to handle the GUI (front-end) and updates the user with the most up to date price of the company and ticker symbol entered using the quote server utility class. When the system is executed the GUI representation presents the user with a drop down menu containing a list of options available to let the user create a new stock folio, as well as load an existing folio or save the one currently active. The view classes also implement the observer interface from the Java API to follow the observer design pattern also.

Our controller package consists of classes that react and handle user events using action listeners, they communicate with the back-end API through Java interfaces and update the altered data within the GUI. The classes play the most important role within the system to provide a level of abstraction between the model and the view, and keeps the data hidden and decoupled.

# Changes to Initial design

Upon our initial design we have carried out substantial refactoring of the system. This includes altering the listener classes which were originally included in the view package. They have now been moved to the controller package, to provide a means of communication between the front and back end, sending communication from the graphical interface through to the API to add stock, delete stock, edit, buy and sell, then output that information to the interface.

We have also completed the communication and associations between the different packages which were initially incomplete, providing a full system and completed end product satisfactory to the user requirements and functional behaviours.

# Back-End API Design

Our back-end design consists of three main classes and three interfaces providing abstraction to each class to the controller through the interface. These interfaces and classes are:

* IFolioTracker & FolioTracker
* IFolio & Folio
* IStock & Stock
* As well as additional Exception classes which extend the Java API classes, to implement system specific exceptions giving the user details of the error gone wrong
* The model also contains the quote server class in the web package used to reference the URL and get accurate information of stock share prices based on the ticker symbol entered by the user to identify the company shares are to be bought from, this works in real time with the opening and closing of the stock market to auto-refresh the details of the price per share and we also calculate the user’s profits or losses on each folio

# Relationships between Front-End and Back-End:

From front-end to back-end communicating by utilising the controller.

Folio Driver: contains the main method and initiates the Folio Tracker View class which presents the user with the GUI.

The Folio Tracker View: is the high level interface showing the user the options available to interact with the stock tracker. This GUI interacts with a number of the controller classes including the create folio listener, load listener and save listener which call a number of the back-end API methods such as create folio, get folios and save the folio details to disk.

Folio View: contains the main text fields of the folio data such as the ticker symbol, number of shares and the share name. Communicates with the add stock listener and delete stock listener which call backend API methods such as getStockName(), getNumberOfShares() and deleteFolio().

Right Click Row: communicates with buy action listener, sell action listener and delete action listener classes to call backend API methods: getStockByTicker(), and buy() method from the folio class as well as getShares(), sell() and deleteFolio().

# Method Specifications

**IFolioTracker Interface:**

CreateFolio:

/\*\*

\* @modifies this

\* @effects folios' = folios + new Folio()

\*/

GetFolios:

/\*\*

\* @returns set of folios

\*/

DeleteFolio:

/\*\*

\* @modifies this

\* @effects folios' = folios - folio

\*/

Refresh:

/\*\*

\* @modifies this

\* @effects refreshes all stocks in all folios such that stock price is updated

\*/

**IFolio Interface:**

CreateStock:

/\*\*

\* @requires

\* @modifies this

\* @effects stocks' = stocks + new Stock();

\* @throws InvalidNameException, NegativeSharesException, NoSuchTickerException, WebsiteDataException, NegativeSharesException

DeleteStock:

@modifies this

@effects stocks' = stocks – stock

getStocks:

@return set of stocks

getValue:

@return folio value

getName:

@return folio name

getStockByTicker:

@return stock where stock ticker equals ticker

**IStock Interface:**

getTicker:

@return ticker

getName:

@return name

getShares:

@return shares

getPricePerShare:

@return price per share

getHoldingValue:

@return total stock value

Buy:

@require

@modifies this

@effects this.shares' = this.shares + shares

totalcost' = totalcost + shares \* pricePerShare

sell:

@require

@modifies this

@effects this.shares' = this.shares - shares

totalcost' = totalcost - shares \* pricePerShare

lossProfit:

@return loss/profit

setName:

@modifies this

# Assertions used

Folio Class:

In the refresh method of the folio class an assertion has been added to identify if the stocks have the same number of stocks after the refresh has occurred, if it hasn’t there has been an unexpected error within the system.

Stock Class:

Refresh Method: in the try catch block of the method an assertion has been added to assert the value of the share price is greater than 0 i.e. is a positive value, if the value is negative the user is alerted with an assertion error as the share price may be negative which should not happen in our implementation of the system.

# Rationale of Junit Tests used

Folio Test class:

This JUnit class implements several Junit tests according to different test conditions, testing the methods of the Folio class. The first tests use a newly created folio with the Microsoft ticker symbol and tests it against the value of the set increased by 1 to determine if the set has increased after the new stock data has been added. The next test uses assert true to validate the same data as the first test, however this one adds another stock to the folio and the folio is checked for equality to see if it contains all of the old stock data and the new added stock. This class also tests our created exception classes, which throws WebsiteDataException first by checking that inputting a negative share value throws the exception as well as an empty share name and an invalid ticker symbol entered which is not a company valid in the stock market. This class also tests consistency, reflexivity, symmetry and not equal null on certain values of objects created.

Folio Tracker Test class:

This test class involves a set up method, instantiating three references to the folio tracker class. The

create folio test involves creating a copy set that references one of the objects created, and adds a

folio to the set of folios then compares the copy set is equal to the expected value which is zero, as

well as the size of the ft1 object which should now have a size of one after adding the test folio to

the set. We also test adding a duplicated folio to the set of folios, and check that the code throws

the expected exception not allowing the duplicate to be added. An empty folio is also

attempted to be added to test the correct exception is thrown and prevent the system from adding

empty folio names. Unit test methods are included to test the deletion code of the folio and assert

that the folio is correctly deleted from the folio tracker when deleted from the set. The refresh

mechanism of the application is tested to provide an insight into the functionality of refreshing

whilst throwing the applicable exceptions.

StockTest Class:

Has several stock objects set up to test the system with legitimate ticker symbols, stock names and shares to be bought. Tested the methods to buy shares correctly from the stock class, as well as trying to buy a negative number of shares which will throw the expected exception. Tested the name of the stock matches the expected value input by the user, and that the holding value meets the expected value set by the user who can then by and sell shares. We also test reflexive, symmetric, consistent, transitivity, the hash code, equals method and the ticker symbol tests the user inputting an invalid ticker in the stock market.

# Class Diagram

Diagram of top level source code packages linked showing the model view and controller showing dependencies:

