**CS308 Folio-Tracker**

**Chloe Forsyth xeb12185@uni.strath.ac.uk**

**Tom Maxwell gvb12182@uni.strath.ac.uk**

**Ryan Gibson yfb13226@uni.strath.ac.uk**

**Amanda Renny gpb12164@uni.strath.ac.uk**

**Design rational:**

We decided that we would use a development method known as the Rational Unified process as this centers largely around iterative programming and revolves around four key stages; inception, elaboration, construction and transition. The inception stage does not really apply here as we were given a project brief as the task and as the program will not be ran live by anyone; nor does the transition phase apply as the program will not be ran live and there is no “real” customer. However the methodology as a whole worked well with our own style. The elaboration phase was useful as it allowed us to break the program down into constituent parts and work out what we needed to do for each of them. The construction phase was again key to us as this is where the actual programming took place. After something had been programmed we could go back and see what needed changed and how it fitted in with the rest of the program as a whole.

**API Design inc specs:**

Tracker.java: Handles the creation, deletion and viewing of portfolios. Manages the refresh rate of updates to the model.

Contained methods:

public Portfolio createPortfolio(String Name)

Modifies: This

Effects: Creates a new portfolio in the model and returns it

public Portfolio loadPortfolio(File inputFile) throws FailedToLoadFileException

Modifies: This

public boolean deletePortfolio(Object o)

Modifes: This

Effects: Deletes the given portfolio

public List<Portfolio> getPortfolios()

Effects: returns a list of portfolios

Public void setRefreshRate(long RefreshRate) throws IllegalRefreshRateException

Effects: Sets the model refresh rate to the given refresh rate

public void addObserver(Observer o)

Portfolio.java: Interface of a Portfolio object. Handles all Portfolio-specific activities and data, including the addition and deletion of stocks and getting stock related details.

Contained methods:

public Stock newStock(String ticker) throws NoSuchTickerException

Modifies: this

Effects: Creates a new Stock object

public boolean deleteStock(Object o);

Modifies: this

Effects: Deletes given Stock from the Portfolio. Returns True if successful, otherwise returns False.

public List<Stock> getStocks();

Effects: Returns this.stocks

public double getTotalValue();

Effects: Returns this.totalValue

public double getNetGain();

Effects: Returns this.netGain

public String getName();

Effects: returns this.name

public void savePortfolio(File outputFile)  throws FailedToSaveFileException

Effects: saves the portfolio to the given file

public boolean isSaved()

Effects: returns true if the portfolio was saved, false if it wasn't

Stock.java: Interface of a Stock object. Handles all information required of a publically traded stock. Can add and remove shares, and get stock-specific details.

Contained methods:

public boolean addShares(int noShares) throws

CantPurchaseMoreThanAvailableException

Requires: must be a positive number of shares

Modifies: this

Effects: Adds shares to this.shares

public boolean removeShares(int noShares) throws

CantRemoveMoreThanOwnedException

Requires: must be a positive number of shares

Modifies: this

Effects: Removes shares from this.shares

public String getTicker();

Effects: Returns this.ticker

public String getName();

Effects: Returns this.name

public double getTotalSpent();

Effects: Returns this.totalSpent

public String getExchange();

Effects: Returns this.exchange

public double getCurrentValue();

Effects: Returns this.currentValue

public double getHoldingValue();

Effects: Returns this.holdingValue

public int getNumberOfShares();

Effects: Returns this.noShares

public double getNetGain();

Effects: Returns this.netGain

public double getClosingPrice();

Effects: Returns this.closingPrice

public double getOpeningPrice();

Effects: Returns this.openingPrice

public double getDailyChange();

Effects: Returns this.dailyChange

public double getDailyMax();

Effects: Returns this.dailyMax

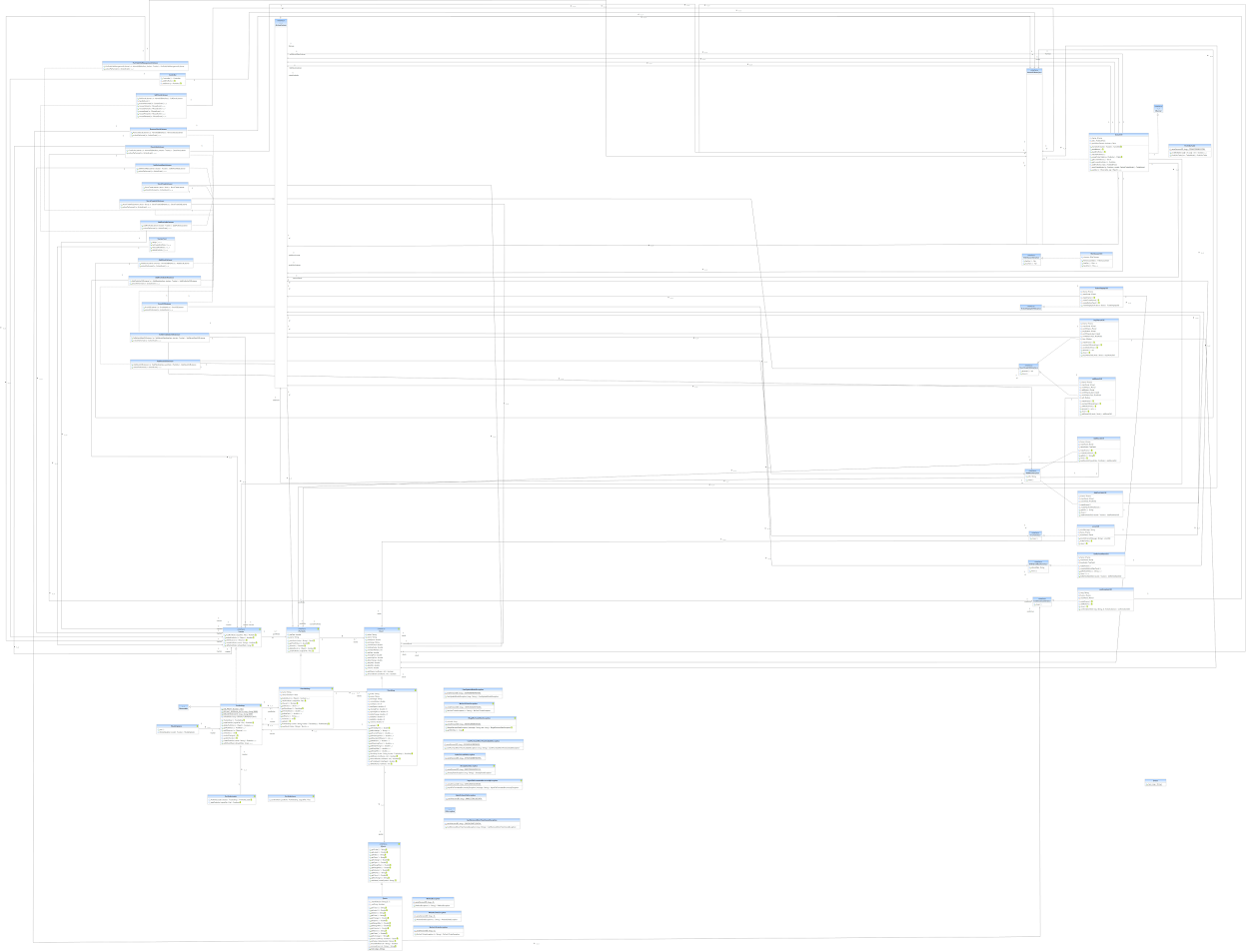
public double getDailyMin();

Effects: Returns this.dailyMin

public double getVolume();

Effects: Returns this.volume

**Class Diagram:**



**Assertions and JUnit Tests:**

As we were lost a group member the day before this was due we have a limited number of assertions and Junit tests. Assertions have been added within the code of the model and are checking in many places that parameters are not null inputs to allow for smooth running of the code and to catch any errors. Here we are assuming that the inputs for ticker and tracker should never be null as the methods they are used within require these as input to function in the correct manner.   
  
assert ticker!=null;  
assert tracker!=null;  
  
They have also been placed in multiple areas of the model to check values against each other to ensure correctness such as;  
  
assert (dailyMax > (dailyMin)): "the daily max is lower than the daily min!";  
  
which does a basic sanity check to make sure that the obvious is correct, being that the daily maximum is always greater than the daily minimum. In other areas assertions have been used to check that values are equal to themselves for example;  
  
assert(totalSpent == (totalSpent)): "total spent does not equal itself";  
  
which should always be true as totalSpent should always be equal to itself and the error message shouldn't occur during the running of the code.

For the Junit test we only tested the Model Package as this is all we had time for. We tested a number of different methods from each class to ensure that all data was being passed correctly and the requires and effects clauses of each specification were being met.

/\*\*

\* Test to check that shares are added correctly

\*

\* **@throws** CantPurchaseMoreThanAvailableException

\*/

@Test

**public** **void** testAddShares() **throws** CantPurchaseMoreThanAvailableException {

stock1.addShares(100);

assertEquals(100, stock1.getNumberOfShares());

stock1.addShares(1744);

assertEquals(1844, stock1.getNumberOfShares());

stock1.addShares(0);

assertEquals(1844, stock1.getNumberOfShares());

stock1.addShares(-100);

// negative numbers are allowed in addShares for some reason

assertEquals(1744, stock1.getNumberOfShares());

}

We also tested to ensure that some methods work continuously through updates to the information being passed into the program.

/\*\*

\* Test to check that totalValue works as expected.

\* Also tests if it works when a stock is removed

\* or shares are sold

\* **@throws** NoSuchTickerException

\* **@throws** IOException

\* **@throws** AlreadyExistsException

\* **@throws** CantPurchaseMoreThanAvailableException

\* **@throws** CantRemoveMoreThanOwnedException

\*/

@Test

**public** **void** testTotalValue() **throws** NoSuchTickerException, IOException,

AlreadyExistsException, CantPurchaseMoreThanAvailableException, CantRemoveMoreThanOwnedException {

portfolio1.newStock("AAPL");

portfolio1.newStock("ms");

List<Stock> stocks = portfolio1.getStocks();

stocks.get(0).addShares(1000);

stocks.get(1).addShares(1000);

//values of 1 share

**double** stock1Value = stock1.getCurrentValue();

**double** stock2Value = stock2.getCurrentValue();

**double** expectedTotalValue = (stock1Value \* 1000) + (stock2Value \* 1000);

assertEquals(portfolio1.getTotalValue(), expectedTotalValue, ***DELTA***);

// test when shares are removed

stocks.get(1).removeShares(10);

expectedTotalValue = (stock1Value \* 1000) + (stock2Value \* 990);

assertEquals(portfolio1.getTotalValue(), expectedTotalValue, ***DELTA***);

//test when a stock is removed

portfolio1.deleteStock(stocks.get(1));

expectedTotalValue = (stock1Value \* 1000);

assertEquals(portfolio1.getTotalValue(), expectedTotalValue, ***DELTA***);

}

Our tests also made sure that the correct exceptions were being thrown so that the user receives the correct message.

/\*\*

\* Test NoSuchTickerException is thrown when user enters incorrect ticker

\*

\* **@throws** NoSuchTickerException

\* **@throws** IOException

\* **@throws** AlreadyExistsException

\*/

@Test(expected = NoSuchTickerException.**class**)

**public** **void** testNoSuchTickerException() **throws** NoSuchTickerException,

IOException, AlreadyExistsException {

portfolio1.newStock("aaaaaa");

}