***Testing***

The decision was made to use unit tests for this project as all the core functionality is performed by concrete java methods. This makes unit testing the most appropriate approach, as opposed to integration testing. This means that all the concrete classes in the Model package, StockModel and FolioModel, were tested. Junit 5.4 was used as it was the most recent version of Junit available. The tests were designed to focus on each functional aspect of each class as well as ensuring that any implemented validation works as intended. Our build of the folio tracker passes all the tests described below. This document is broken down by class and then class methods. The tests used for each method are described under their respective subheadings.

**FolioModel Tests**

**newStock()**

The tests for the adding a new stock involve adding an instance of the StockModel to the empty ArrayList. Then we call newStock() on the folio object with the same parameters as before. We then assert that all the StockModel’s values in the folio object array match the StockModel’s values in the dummy array.

We also create another test with bad inputs and assert the returned value is null.

**buyStock()**

This method is tested by adding a StockModel object, with initial share count of 1, to the array in the test FolioModel object. Then we buy another share of the same stock and assert that the number of shares stored for that stock is 2.

We also assert that the method returns null on bad inputs.

**deleteStock()**

This method is tested by first adding a stock to and array, then asserting that the deleteStock() method returns true.

We also assert that the method returns false on bad inputs, such as calling it on an empty array and with the incorrect ticker symbol

**save() and load()**

The save feature is tested by ensuring that calling the save method returns true. The load method is tested by ensuring that calling the load method on the file created in the save test does not throw an exception. The validation of the load method is tested by ensuring that attempting to load from a non-existent file returns null.

**Sorting**

This feature is tested by adding 2 stocks to the test folio stock list an ensuring that the correct one is in index 0 of the stock list depending on which criteria is being used while sorting.

**StockModel Tests**

**Object Construction**

The construction of the objects is tested by calling refresh on each of the 2 stockModel test objects. Good arguments are tested by asserting that refresh does not return null. Bad arguments are tested by asserting refresh() does return null.

**refresh()**

This method is hard to test as we can never know what the value of the stock we are pulling is, and the errors thrown are beyond our control. The best we can do is test that it returns a number and not null.

**sellShares()**

This feature is tested by ensuring that selling one stock of the good test stock returns true. The validation for this feature is tested by asserting that attempting to sell more shares than exists of the good test stock returns false. It also tested by asserting that the number of shares remains the same.

**buyShares()**

This feature is tested by saving the value of a stock before buying more stock the checking that the value after the sale is higher than beforehand. It is also asserted that the number of stocks increase to the correct amount

**setValue()**

This was tested by giving the method a value and asserting it returns the correct result