**Design Explanations**

**Introduction**

For the given stage 2 assignment two for FIT 3077, several design pattern and principles have been implemented as demonstrated in the unit lectures. The purpose behind them is to adhere to correct software engineering principles. The following report outlines the design patterns and principles used in our code and why we have chosen to use them.

**Overview**

This application is called “Stock Quote Monitor Application”, user needs to input a stock symbol of the stock that he/she wants to monitor, choose the type of the monitor (can either be numerical or graphical, default to numerical at the start), and also choose the live API (StockQuoteService) or the API that gives the data from the past since 22nd of September, 2010. Then, by simply clicking the add button, the information will show on the area below. If a user wants to remove a monitor, he/she can just select the monitor by clicking the title, then click the delete button.

This application is started by initialising an **AppDisplay** in the **Main** class. The **AppDisplay** class is in charge of the display of this application. It takes user’s input to parse it to **InputHandler** class and get the monitor panel generated from the **Monitor** class to show on the screen. The **InputHandler** class takes the user’s input and choose the right API and monitor type to use by calling/initialising them. It also has a **StockQuote** type to store the information got from other classes.

We have two types of API by calling different functions in the API classes, so we decided to make an interface **StockQuoteServiceInterface**, with a function **getQuote**(), and implements them in 2 classes of calling those 2 API classes respectively. And because those 2 subclasses are the classes to get the information from the API, so we decided to put a thread in each of the **StockQuoteService** and **TimeLapseService** classes to control how long we need to fetch the data from the API and update the new information in the app. According to the assignment specification, we have set the thread for **StockQuoteService** to sleep every 1 minute and 5 seconds for **TimeLapseService**. We implemented the thread in each class by extend **Thread** class and override the **run()** method. Then in each class’s constructor, we call **start()**.

**StockQuote** class has a constructor to take the formatted information from the **StockQuoteServiceInterface**. It can be used as a type to store the **StockQuote** information. **StockQuote** class has a **StockInfo** class, **StockInfo** is a class has only all the attributes of the arguments in **StockQuote** constructor but no methods, so we are using this class to store the information and also using this class to get data conveniently. For example, if we store the information in the **StockQuote** attributes which was what we did in stage 1, then we would have to make corresponding methods to call each attribute instead of just simply using (**StockInfo**).(attribute). We have used **StockInfo** a lot in Monitor class, only because **StockInfo** is much easier to use and call all the attributes.

**Monitor** is an abstract class. The reason we did this is because we have 2 types of monitor and we want to render the data in one kind of data type to the front end and display it, so we are rendering the monitor data as a **JPanel** type to the **AppDisplay** class so it doesn’t need to adjust for different types of data. In the **TextMonitor** we have implemented **generatedMonitor**() method from the superclass by adding content to the attribute **textArea ,** then add the whole **textArea** to the **jPanel** and render to the frontend. We have also implemented the **updateMonitor**() method from the superclass by renewing the **textArea** and also the **stockInfo.** In the **GraphicMonitor** we have done pretty much the same thing, but we call another class **GenerateGraph** to generate the graph because we can’t extend 2 classes in the same time and we need to here for **Monitor** and **ApplicationFrame**. We update the graphic monitor by adding new data we got from the API to the line chart.

**Monitor** class and **StockQuote** class are **Observer** and **Observable** relationship, we made this decision because **Monitor** gets data from **StockQuote** and **Monitor** needs to know when **StockQuote** information gets updated.

**Design patterns**

**Observable and observer**

An observable/observer pattern was implemented in our design and code. The **StockQuote** class implemented the **Observable** class and the **Monitor** class would implement the observer class. This was chosen to be implemented because it would be necessary for the monitor class to know when a stockquote has changes its values . By implementing this pattern, the **Monitor** class is informed when a stockquote has changed its value and when informed it can make necessary changes to satisfy the new stockQuote values. For the **Observable** and **Observer** classes, we have implemented our customized one instead of using the java inbuilt classes, because it’s easier to parse the information for us.

**Dependency inversion principle**

The dependency inversion principle was implemented in the design and code for Stage two part two. This was implemented through the use of interface classes to create abstractions between high level and low level classes. In particular, the **StockQuoteinterface** and **Monitorinterface** classes. These two classes are interfaces and have classes which implement them. The reasoning behind such a design is that it will: Reduce coupling between the classes that implement the interface classes.

**Design decisions**

**Redesign of the the class diagram**

It was found that the old design of stage two part a was not very open to the idea of extensibility. Extensible in the sense of adding different kind of monitors(i.e a graphic monitor). There were also problems with classes being rigid and not robust(The **monitor tabl**e,**stockMonitor** classes from stage 1), these classes prevented extensibility of adding different types of monitors. Hence a new design was needed. This was fixed with implementing an abstract class of **Monitor** where a monitor could then be text or graphic , this redesign then allowed for different monitors in the program and ultimately satisfied the stage 2 requirements.

**Addition of a controller class(InputHandler)**

In stage one of the assignment, a GUI class called **Displaymonitor** was used that communicated directly with the backend classes(i.e stockquote services). It was later apparent that this was of bad design because it should not directly communicate with backend classes. To amend this problem, a controller class was added called **inputHandler** where the GUI window now only communicates with the controller class and the controller class interacts with the back end classes. Essentially acting as a medium between front end backend classes.

**Advantages**

The advantages of our design are it’s very extendable for more kinds of monitor or more stock APIs, and also if we have more user input, it’ll be very easy to handle too by just slightly adding the information to the **InputHandler** class and adding the corresponding methods to perform the actions.

Furthermore, another advantage of the design is that it is capable of handling multiple stock service types. This is allowed through the use of a **StockQuoteService** interface class which allows for different StockQuoteService types. This therefore allows for good extensibility for new StockQuote services.

**Limitations**

We have imported both of the **StockQuoteService** and **StockQuoteTimeLapseService** using **wsimport** command from Terminal, due to some bugs and warnings of **wsimport**, we get warnings every time we call the service.