Design Rationale

The primary design pattern that influenced the structure of the program was the Model View Controller (MVC). MVC was chosen largely due to the requirement to be able to have multiple views observing multiple stocks that are updated from a database[1]. In comparison to the extended UI approach, the MVC pattern allows for the implementation of this without violating the Acyclic Dependencies Principle (ADP)[2]. The use of a model package breaks the cycle between controller and view, and has both depend on model. Additionally, the active model[1] variant of the MVC pattern was used. The inclusion of an observer pattern into the MVC in this way, allows the model to be updated independently of the controller.

In the MVC pattern, model doesn't depend on view. Therefore, new views can be added to the system without having any effect on the model. This was advantageous as it allowed for the implementation of a text view as specified in part 1, whilst also preparing for the addition of other view types in part 2.

The observer pattern plays a key role in the management of passing information to and updating the observers. When a change occurs to the values in stock (whenever the timer is triggered), all displays were needed to be updated. The observer pattern is able to provide this functionality without introducing any tight coupling. It does this in a similar way to how cycles can be broken to adhere to the ADP through the use of interfaces. By making use of the observer interface update() to notify its observers, subject is able to remain only loosely coupled with its observers. This is also helped made possible by the use of dependency injection of and observer into subject, again avoiding tight coupling.

Dependency injection is also used elsewhere in the program in order to adhere to the Dependency Inversion Principle[3] (DIP). For example, the constructor of Stock is parsed a reference to ServerAbstract which it then stores as an attribute, as opposed to creating a new object. This helps to reduce the level of dependency of Stock on a server. If a new server is created by the stock, then that stock instance becomes responsible for the management of that server, however, where dependency injection is utilised, stock only interacts with the interface that the server it is parsed provides. Furthermore, due to the Liskov Substitution Principle[4] and the abstraction provided by the serverAbstract class, stock doesn't need to know the type of server that it will be operating with until the moment it is created. This means that the type of stock used doesn't need to be hardcoded into the Stock class, further increasing its maintainability in the future.

The use of an abstract server[5] design pattern is also used to support the DIP. Having Stock contain a reference to a concrete implementation of a server would violate the DIP. The idea that a stock will rely on a server for information is a very stable one and key to the operation of the program. There is also nothing in the system that dictates what type of server might be suppling the information in the future, so long as the data is provided in the same format, meaning that in the future it may be desirable to implement different types of servers. Maintaining the level of abstraction provided by the abstract server design means if

the server type is changed, these changes can be restricted to that server and don't flow into other classes that make use of it.

References

1. FIT3077 slides – The Model-View-Controller Architectural Pattern
2. FIT3077 slides – Principles of Object-Oriented Design 3
3. FIT3077 slides – Principles of Object-Oriented Design 2
4. FIT3077 slides – Principles of Object-Oriented Analysis and Design
5. FIT3077 slides – Design Principles and Design Patterns