Design Rationale

Architectural Pattern

The architectural pattern 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 getting updates from a webservice. In comparison to the extended UI approach, the MVC pattern allows for the implementation of this without violating the Acyclic Dependencies Principle (ADP). The use of a model package breaks the cycle between controller and view and has both the view and model depend on the controller. Additionally, the active model 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 architecture, the 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, such as a graph view.

Design Patterns & Design Rationale

The observer pattern plays a key role in the management of notifying the views(observers) that stock(subject) has received updated information – and that they (the views) should update themselves to reflect this. We used this design pattern because when a change occurs to the values in stock (whenever the timer is triggered), all displays are 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 the observer into subject, again avoiding tight coupling. Therefore, we can freely add any number of observers that display the data in different ways and do not need to change any code in our concrete subject(stock).

Dependency injection is also used elsewhere in the program in order to adhere to the Dependency Inversion Principle. For example, the constructor of Stock is passed 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 passed provides. Furthermore, due to the abstraction provided by the serverAbstract class, stock doesn't need to know the type of server that it will be communicating with. This means that the type of server used doesn't need to be hardcoded into the Stock class, further increasing its maintainability in the future.

The use of an abstract server 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 specific server for retrieving its information mean stock would become very stable, which reduces the ease in which we could update stock. By having an AbstractServer we can be sure that stock will receive information in the form that it can handle. As stock can only communicate with the servers through an abstract class. Maintaining the level of abstraction provided by the abstract server design means if a server changes its implementation, these changes can be restricted to that specific server and will not affect stock, or any other class. As they communicate through a common, stable interface.