Part 6: Final Report

Team 10:

Douglas Allen
Heesuk Jang
David Zhuzhunashvili

December 12, 2017

Summary:

Stock Score provides investment information to users. Users enter a stock ticker and a date range to generate a stock score which indicates if the user should buy, hold or sell a stock. Users can save a score as a favorite or share the score with other users.

Business Requirements are NONE.

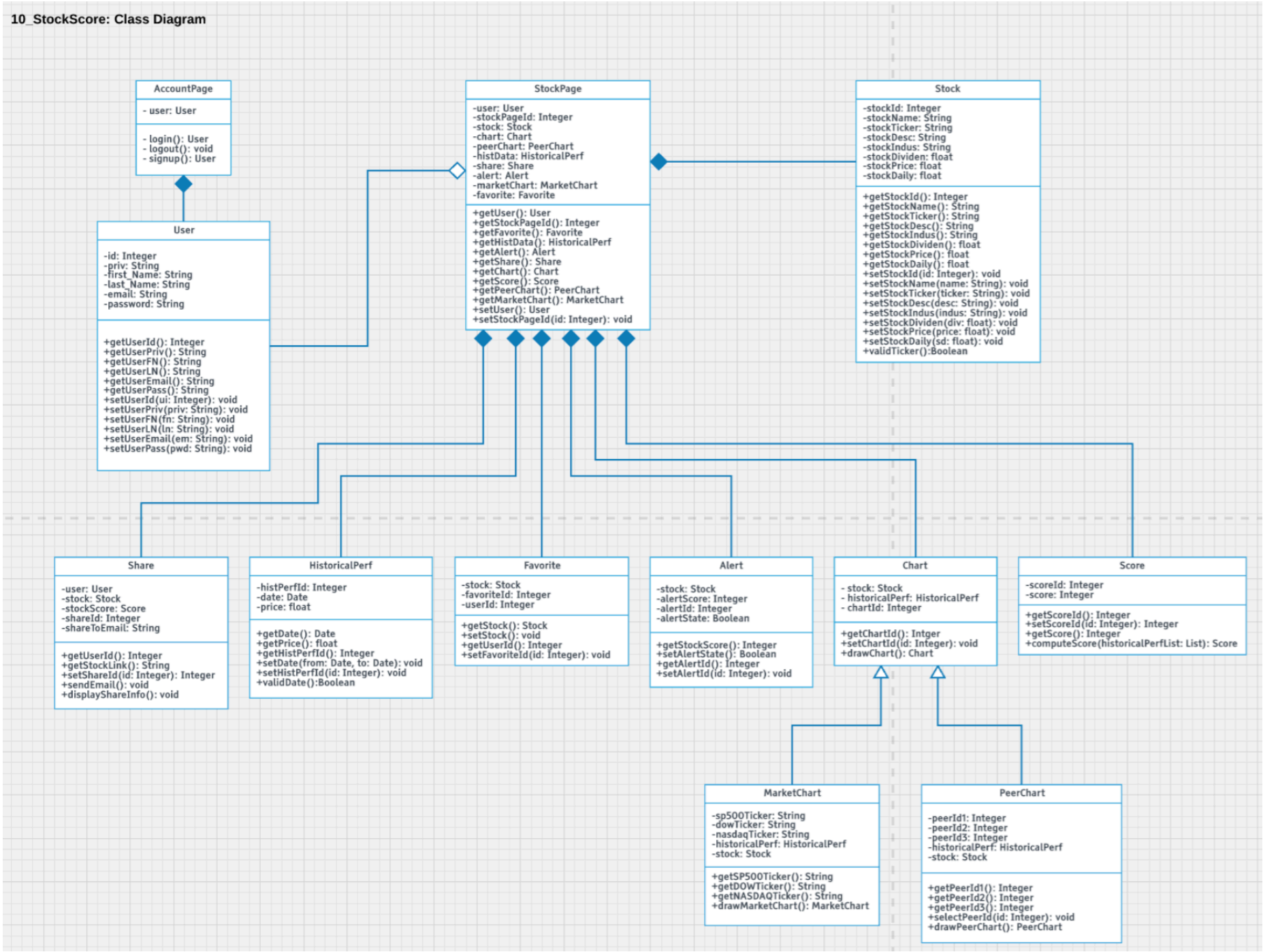
Implemented Features

User Requirements	
ID	**Requirement Title**
UR-03	See the historical price information for a stock.
UR-04	See a stock recommendation score for a stock.
UR-06	Compare the historical price information of my stock with its competitors.
UR-07	Compare the historical price information of my stock with its market benchmarks.
UR-08	Store a stock as a favorite.
UR-011	Share a stock and its recommendation score with others.
Non-Functional Requirements	
ID	**Requirement Title**
NFR-01	Respond within 3 seconds on average to an user's request
NFR-02	Have a capacity of handling 10 simultaneous requests at a time
NFR-03	Display correctly on screen resolution down to 1024 x 768
NFR-04	Display correctly on the latest version of Chrome and Firefox

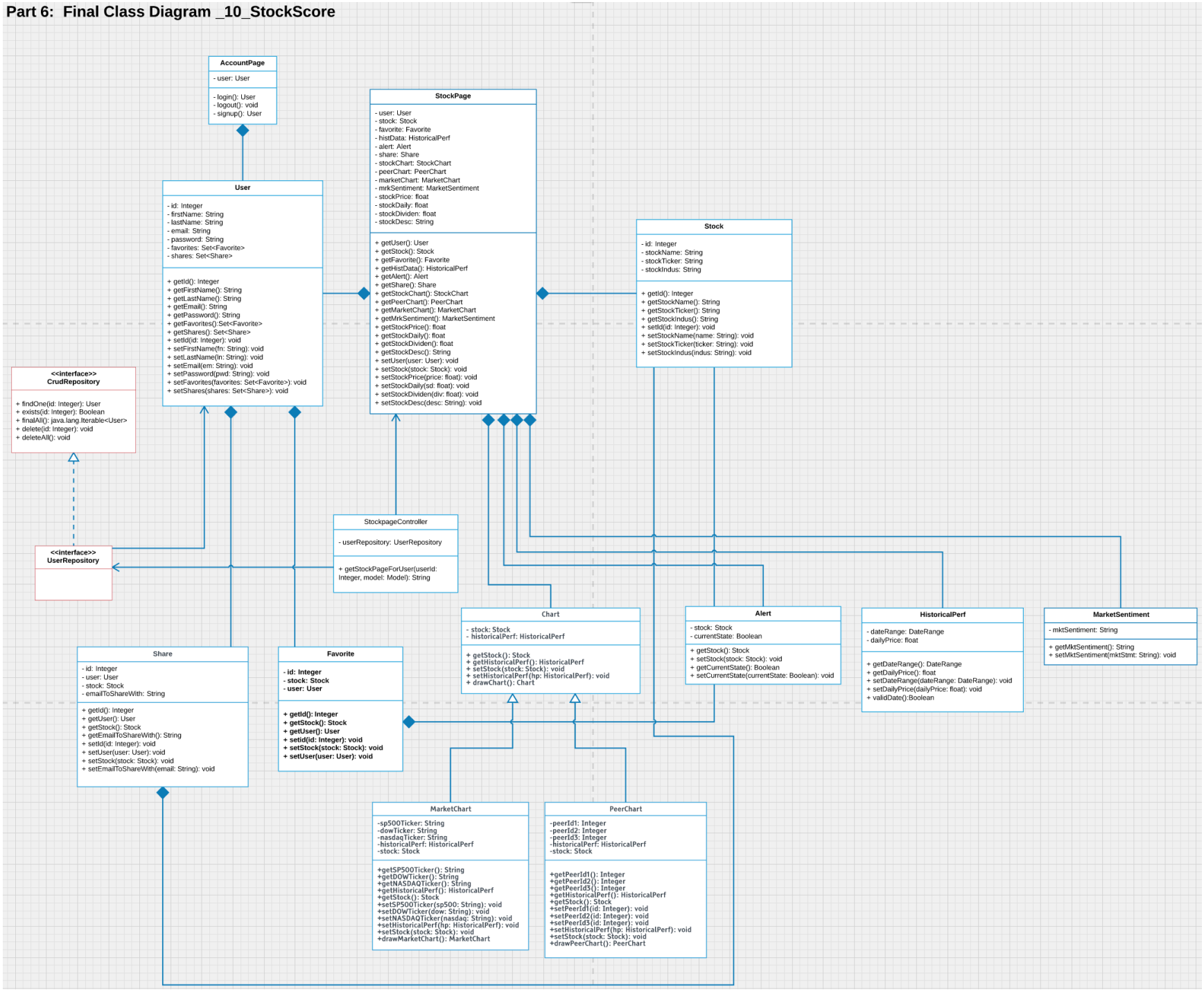
Not Implemented Features

User Requirements	
ID	**Requirement Title**
UR-01	See current price and industry classification of a stock.
UR-02	See company overview of a stock.
UR-05	See dividend payout ratio of a stock.
UR-09	Set alert based on the state of the recommendation score.
UR-010	Notify user when the state of the recommendation score changes.
Non-Functional Requirements	
ID	**Requirement Title**
NFR-05	Authenticate to access StockScore account.
NFR-06	Password security using SSL (https) and encrypted in database.

Old Class Diagram from Part 2:

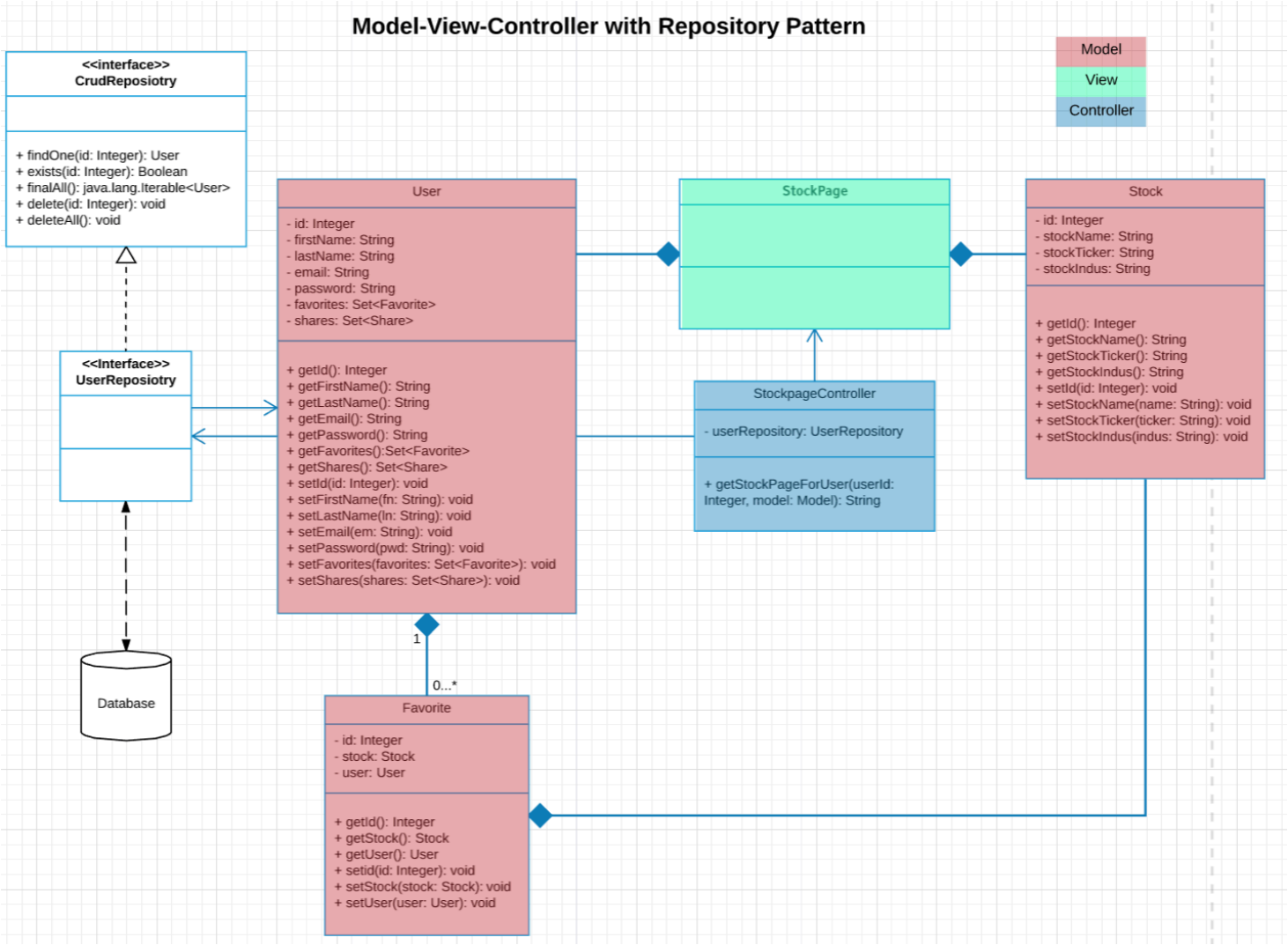


Final Class Diagram:



Design Patterns We Applied:

We used the MVC architectural pattern throughout our project. We used Spring Boot to implement all of our functionality. We used Spring Boot for two main reasons: it is based on the MVC architectural pattern, and it removes most of the boilerplate code that one would need to copy and paste to make new functionalities work. To demonstrate how we implemented the MVC pattern into our project, we will be using the user requirement of Storing a Favorite Stock. Initially, Favorite was a part of the Stockpage class, but then we added it to the User class instead. In our new class diagram, if someone makes a request for a user, the controller (StockPageController) grabs the user ID and searches for it in the database by calling “findOne(userId)” on UserRepository, - the repository is similar to a DAO, but while a DAO is an abstraction of data persistence, the Repository is an abstraction of a collection of objects - the UserRepository then searches for the User and initializes a User object using the information it found in the database for that user. Once the user object is created within the controller, the controller can either store new favorite stocks inside that user using the method “setFavorites” or it can grab the user’s current favorites using the method “getFavorites” and display it to the view (StockPage).



Design Patterns We Did Not Apply But We Planned To:

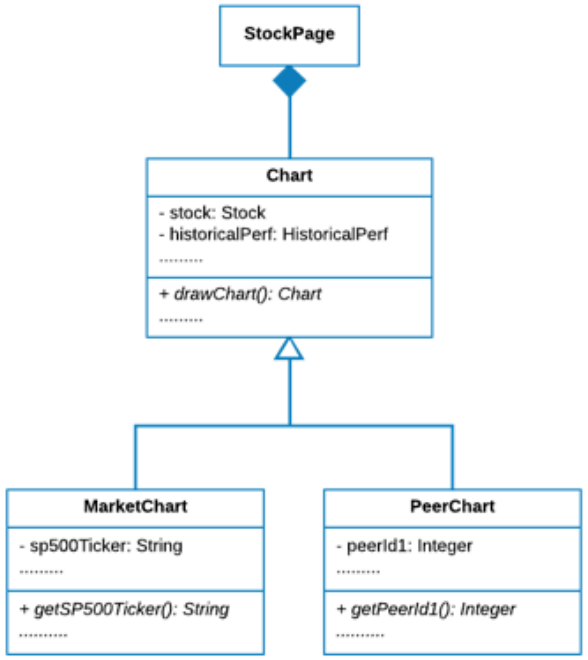
Due to the use of **MVC with Repository pattern**, we did not apply the following design patterns. If we had used them, this would show how they would work.

1. Factory Method Pattern

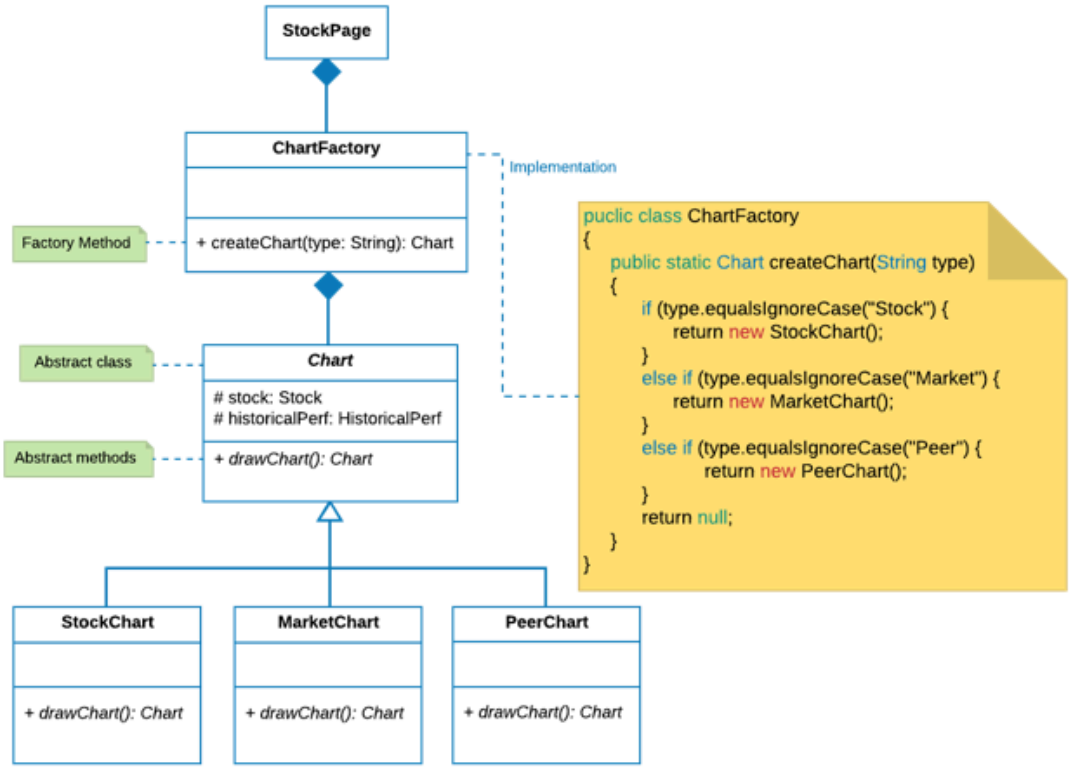
Since there is an inheritance hierarchy among the base class **Chart** and its subclasses **MarketChart** and **PeerChart**, we could add polymorphic creation capability by defining a static factory method **createChart()** in the factory class, **ChartFactory**. The factory method returns three different types of Chart: StockChart, MarketChart, and PeerChart. Each subclass of **Chart** overrides the abstract method, ***drawChart()*** to draw its specific chart. In this way, StockPage (as the client) only needs to call the generic function of *createChart()* in the factory class when new charts need to be added without affecting StockPage or ChartFactory. For more detailed implementation of the changes, please look into the following diagrams.

```
Public class StockPage
{
    public static void main (String[] args)
    {
            Chart sp500Chart = ChartFactory.createChart("Market");
    }
}
```

[Original Class Diagram]



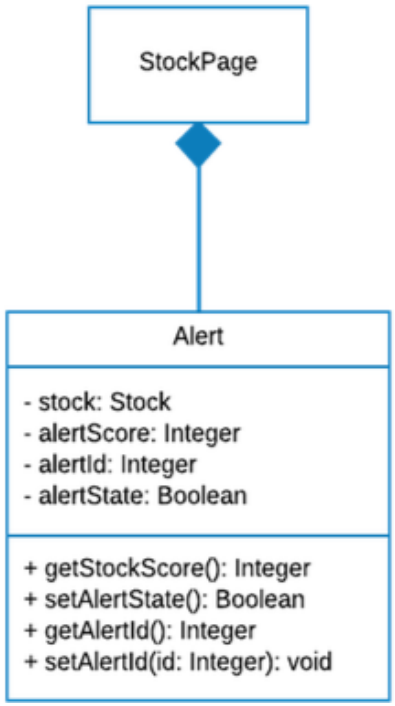
[After Applying Factory Pattern]



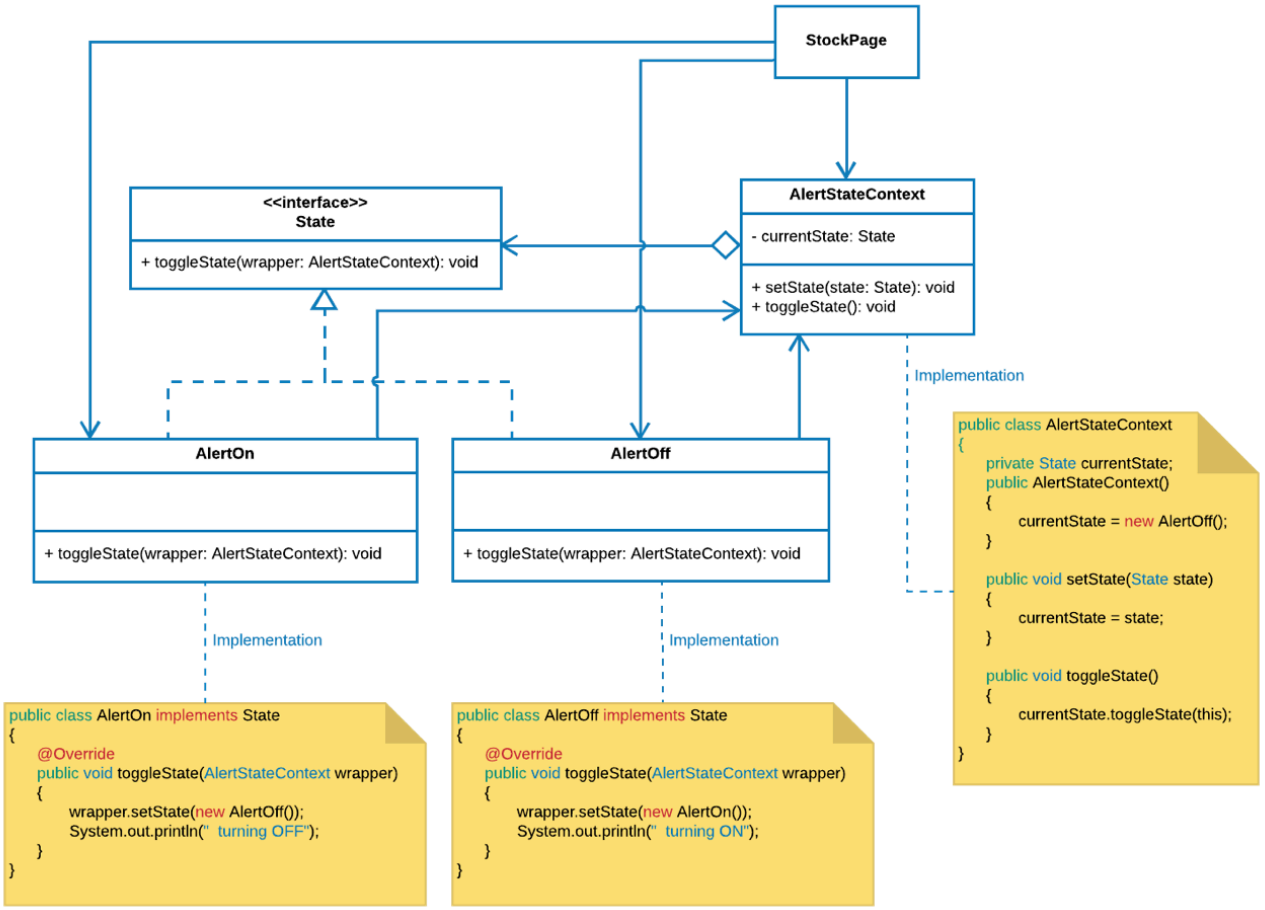
2. State Pattern

To allow the object **Alert** to alter its behavior when its internal state changes from **OFF** and **ON** and vice versa, we applied State Pattern to create two new classes, **AlertOn** and **AlertOff** of the context object, **AlertStateContext** and to extract the state-related behaviors into these classes. **AlertStateContext** contains an instance of a concrete State subclass, **AlertOff**, that defines the *currentState* and the **AlertStateContext** will delegate the execution to each state object, **AlertOn** and **AlertOff**. The class **AlertStateContext** defines the interface ***State*** of interests to **StockPage** and ***State*** encapsulates *toggleState(wrapper: AlertStateContext)* behavior associated with a particular state of the **AlertStateContext**. Both **AlertOn** and **AlertOff** subclasses implement a behavior associated with On or Off state of the **AlertStateContext** by overriding *toggleState(wrapper: AlertStateContext)* method, which follows the common interface ***State***. For more detailed implementation of the changes, please look into the following diagrams.

[Original Class Diagram]



[New Class Diagram After Applying the State Pattern]



Retrospective: What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

Through the process of analysis and design we have learned many things, both coding related and non-coding related. We learned that using the MVC pattern is very useful, it structures the code very well and allows us to create very modular and scalable (in terms of functionality) projects. However, we also learned that dividing the work up, MVC's separations of concerns, into model, view, and controller makes the final process of merging all the separate codes a great challenge. Yet, we still believe that the project would be a lot more disorganized without the MVC pattern making it much harder for us to implement (payoffs outweigh the costs). It would have been a lot more efficient if everyone implemented their own functionalities, with model, view, and controller, rather than having each person implement a layer of the MVC pattern. In this way, the merging process could have been much easier in our situation.

One specific thing we really liked about using MVC with Spring Boot was the Thymeleaf template which is a pure html template which made integrating our website designs into our code extremely easy.

The refactoring process was also another very useful portion of our design process. Not only did we end up removing many functionalities we no longer needed, but we also applied new design patterns which made the actual code development much easier and structured.

Overall, we gained a lot of experience with the MVC pattern and got a deeper understanding of how it works and why it is so useful from doing this project. Using Spring Boot made the learning process for MVC much easier since it makes implementing difficult functionalities very easy.

We also gained experience in using the agile development process during this project. We met once a week and discussed what we had done and what we would finish before the next sprint, which would start next week at the same time. Using this approach made our progress much faster since we always knew what everyone was doing. Nevertheless, it would have been much more efficient if we had at least two meetings per week so we could game more frequent updates on everyone's work.