Team 01

1. Dante Pasionek → Dpasi314
2. Preston Kelley → pkelley
3. Walker Schmidt → walkyd12

Project

PaperTrader - An alternative to traditional stock market simulators
Creating a user-friendly trading environment to provide real-time information to gain valuable trading suggestions

Project Summary

A python based stock market simulator, web application that allows users to trade in real time with fake currency. Traders are able track their investments and find optimal trading strategies.

Index

Previous Class Diagram

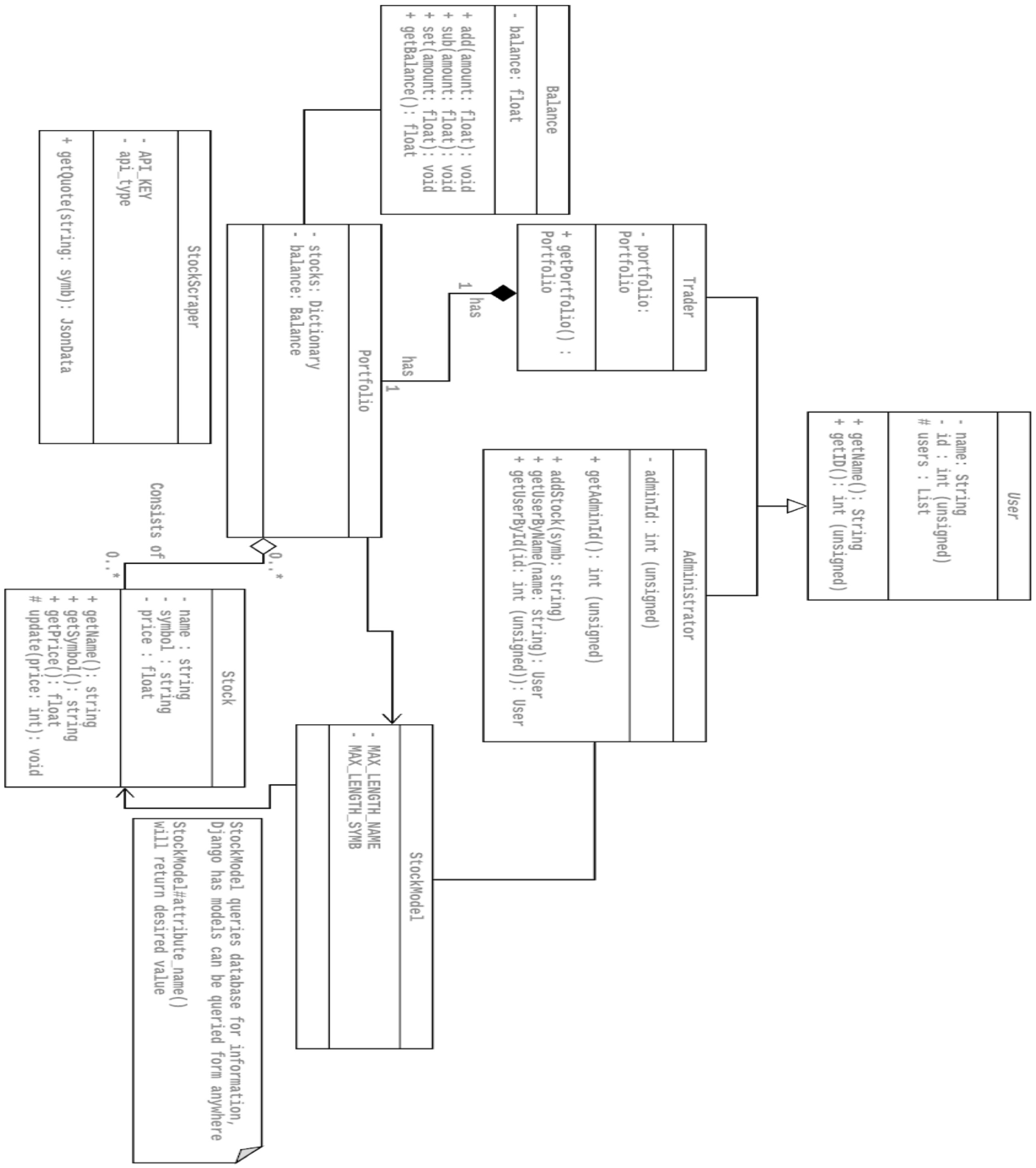

Balance
- balance: float
+ add(amount: float): void
+ sub(amount: float): void
+ set(amount: float): void
+ getBalance(): float
StockScraper
- API_KEY
- api_type
+ getQuote(string: symb): JsonData
Portfolio
- stocks: Dictionary
- balance: Balance
Trader
- portfolio: Portfolio
+ getPortfolio() : Portfolio
has
1
has
1
User
- name: String
- id : int (unsigned)
# users : List
+ getName(): String
+ getID(): int (unsigned)
Administrator
- adminId: int (unsigned)
+ getAdminId(): int (unsigned)
+ addStock(symb: string)
+ getUserByName(name: string): User
+ getUserById(id: int (unsigned)): User
Consists of
0..*
0..*
Stock
- name : string
- symbol : string
- price : float
+ getName(): string
+ getSymbol(): string
+ getPrice(): float
# update(price: int): void
StockModel
- MAX_LENGTH_NAME
- MAX_LENGTH_SYMB
StockModel queries database for information,
Django has models can be queried form anywhere
StockModel#attribute_name()
will return desired value

Figure 1: Previous class diagram

Completed Class Diagram

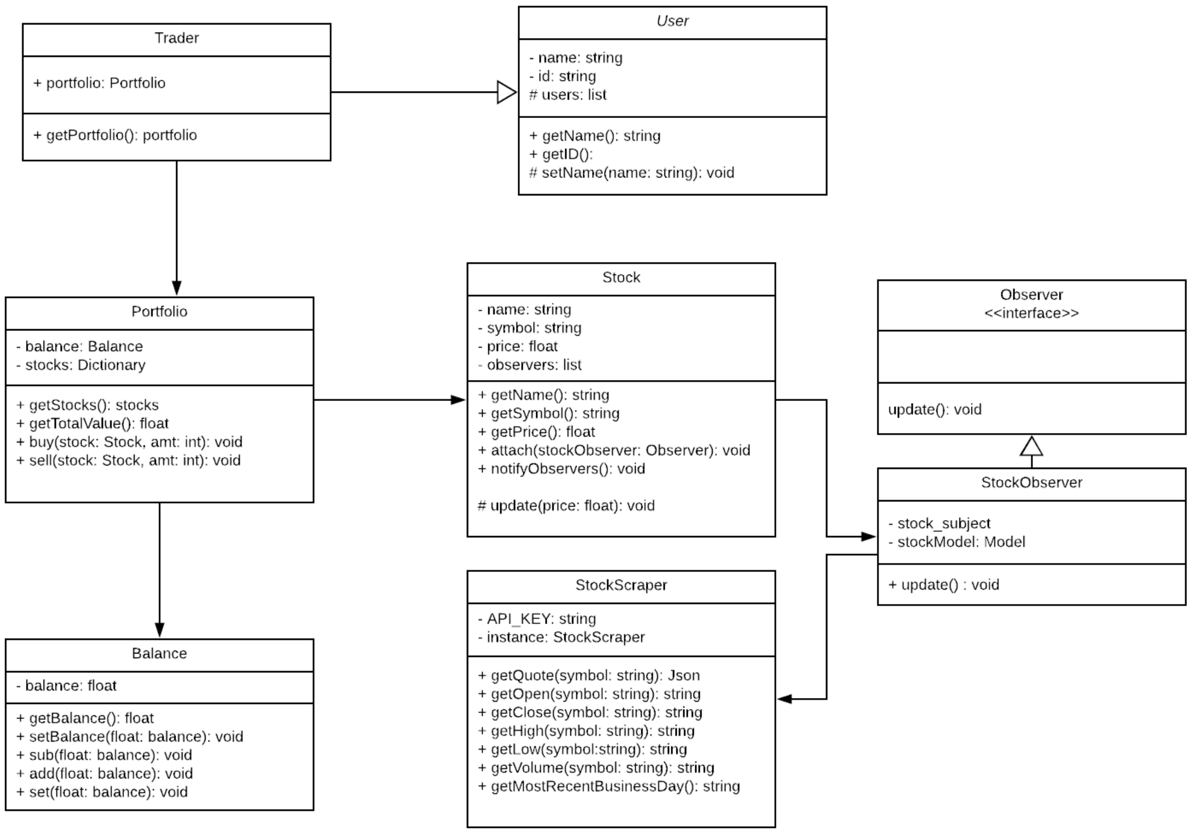

Trader
+ portfolio: Portfolio
+ getPortfolio(): portfolio
User
- name: string
- id: string
# users: list
+ getName(): string
+ getID():
# setName(name: string): void
Portfolio
- balance: Balance
- stocks: Dictionary
+ getStocks(): stocks
+ getTotalValue(): float
+ buy(stock: Stock, amt: int): void
+ sell(stock: Stock, amt: int): void
Stock
- name: string
- symbol: string
- price: float
- observers: list
+ getName(): string
+ getSymbol(): string
+ getPrice(): float
+ attach(stockObserver: Observer): void
+ notifyObservers(): void
# update(price: float): void
Observer
<<interface>>
update(): void
StockObserver
- stock_subject
- stockModel: Model
+ update() : void
StockScraper
- API_KEY: string
- instance: StockScraper
+ getQuote(symbol: string): Json
+ getOpen(symbol: string): string
+ getClose(symbol: string): string
+ getHigh(symbol: string): string
+ getLow(symbol:string): string
+ getVolume(symbol: string): string
+ getMostRecentBusinessDay(): string
Balance
- balance: float
+ getBalance(): float
+ setBalance(float: balance): void
+ sub(float: balance): void
+ add(float: balance): void
+ set(float: balance): void

Figure 2: Completed class diagram

Summary

It’s important to note that while our class diagram may look less meaty than the previous one, however Django provides a lot of functionality. Django models provide painless SQLite3 database access. Additionally, Django provides a default admin structure which already exists within to create the functionality with which the project attempted to replicate, because of this, the Admin class was omitted from this class diagram. Similarly, some time was spent filling in the scaffolding of framework, primarily focused on designing the Django models (Detailed below, Figure 3). Designing the admin module through Django required the StockModel, other classes such as Stock, StockObserver and Portfolio classes all make use of the StockModel. The Stock class was modified to not only house the information from the model, but also uses the Observer design pattern to simultaneously save new information to the database while also updating a portfolio with an updated price, opening, closing, high, low or volume value.

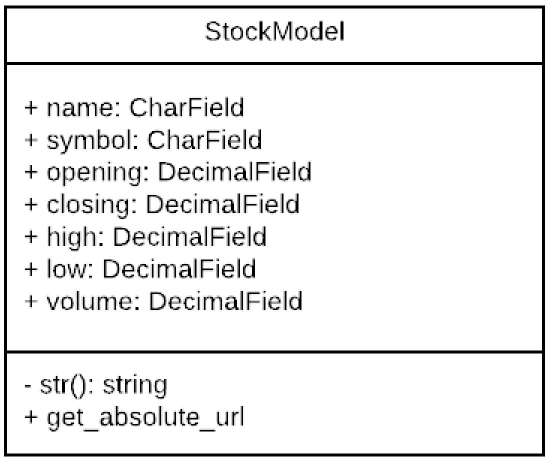
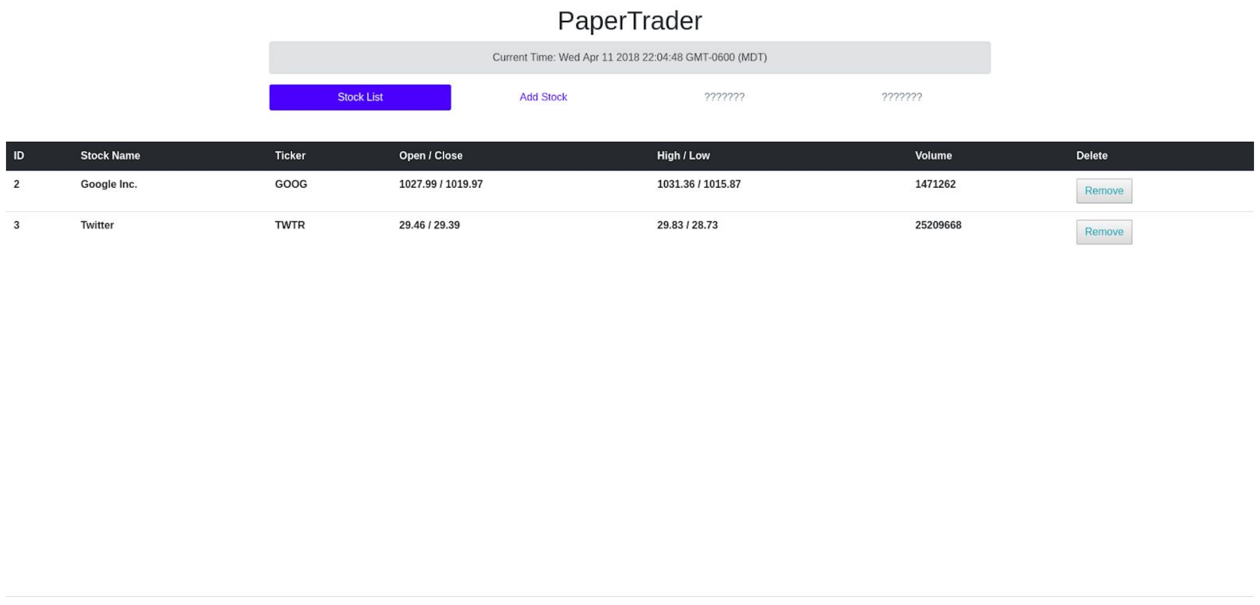


Figure 3: StockModel

During development time, many tests were created to help assure that the product would be functional. These tests, also integrated into Django and thus transparent in the class diagram, are for every major focus point of this project. For example, the Stock, Balance, Portfolio, StockObserver and StockScraper all have tests to ensure proper functionality as the project nears completion. It is necessary for all these classes to interact with one another in specific ways, and allow us to guarantee that specificity. Finally, time was also spent on creating and designing the views (Figures 4, 5, 6) to interact with the system. The three primary views, Stock list, create stock, delete stock are created right now.



ID	Stock Name	Ticker	Open / Close	High / Low	Volume	Delete
2	Google Inc.	GOOG	1027.99 / 1019.97	1031.36 / 1015.87	1471262	Remove
3	Twitter	TWTR	29.46 / 29.39	29.83 / 28.73	25209668	Remove

Figure 4: Stock list

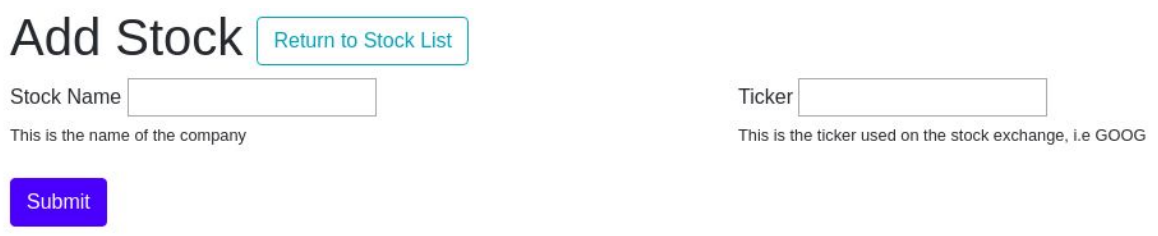
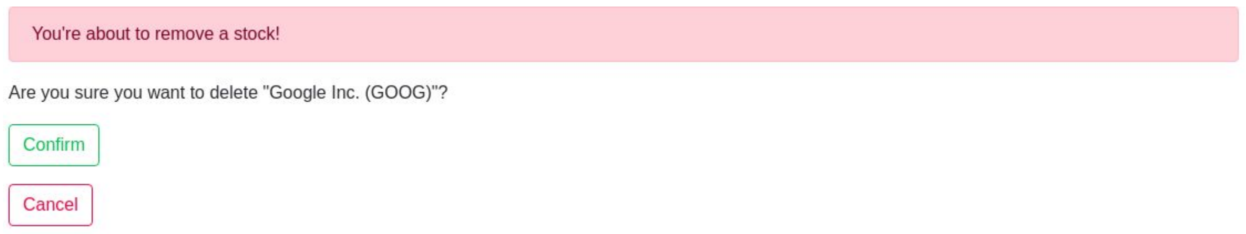

Add Stock
Return to Stock List
Stock Name
This is the name of the company
Ticker
This is the ticker used on the stock exchange, i.e GOOG
Submit

FIgure 5: Add stock page

We see the two main fields a user must fill out. The Stock name is mostly for Stock object aesthetic, it is a string stored and presented to the user on the stock list, and as of now the API doesn't provide this information. For this reason, the stock name is allowed to be anything desired. Conversely, the ticker symbol is required. The ticker symbol is what is distinguishes the company on the stock exchange and also is what the API uses to collect information. Hitting the submit button will call the StockScraper to gather the information. This also created an object in the StockModel (which is saved in a SQLite3 database) and therefore accessible by Stock object and can be referenced throughout the system.


You're about to remove a stock!

Are you sure you want to delete "Google Inc. (GOOG)"?

Figure 6: Remove a stock

As required by Django, a confirmation screen is required before deleting an object when using their default removing form. The primary key (unique number) is passed into the delete_stock Django model where the framework handles the deletion.

Breakdown of Work

I. Stock Model
   A. Design
      1. Dante
      2. Walker
      3. Preston
   B. Implementation
      1. Dante
      2. Walker
II. Portfolio
   A. Design
      1. Dante
      2. Walker
      3. Preson
   B. Implementation
      1. Preston
III. User
   A. Design
      1. Dante
      2. Walker
      3. Preston
   B. Implementation
      1. Walker
IV. Balance
   A. Design
      1. Dante
      2. Walker
      3. Preston
   B. Implementation
      1. Dante
V. Stock
   A. Design
      1. Dante
      2. Walker
      3. Preston
   B. Implementation

            1. Dante
            2. Walker

VI. Stock Observer
    A. Design
        1. Dante
        2. Walker
        3. Preston
    B. Implementation
        1. Dante

VII. StockScraper
    A. Design
        1. Dante
        2. Walker
        3. Preston
    B. Implementation
        1. Dante

VIII. Models
    A. Design
        1. Dante
        2. Walker
        3. Preston
    B. Implementation
        1. Dante
        2. Walker
        3. Preston

IX. Views
    A. Design
        1. Dante
    B. Implementation
        1. Dante

GitHub Graph

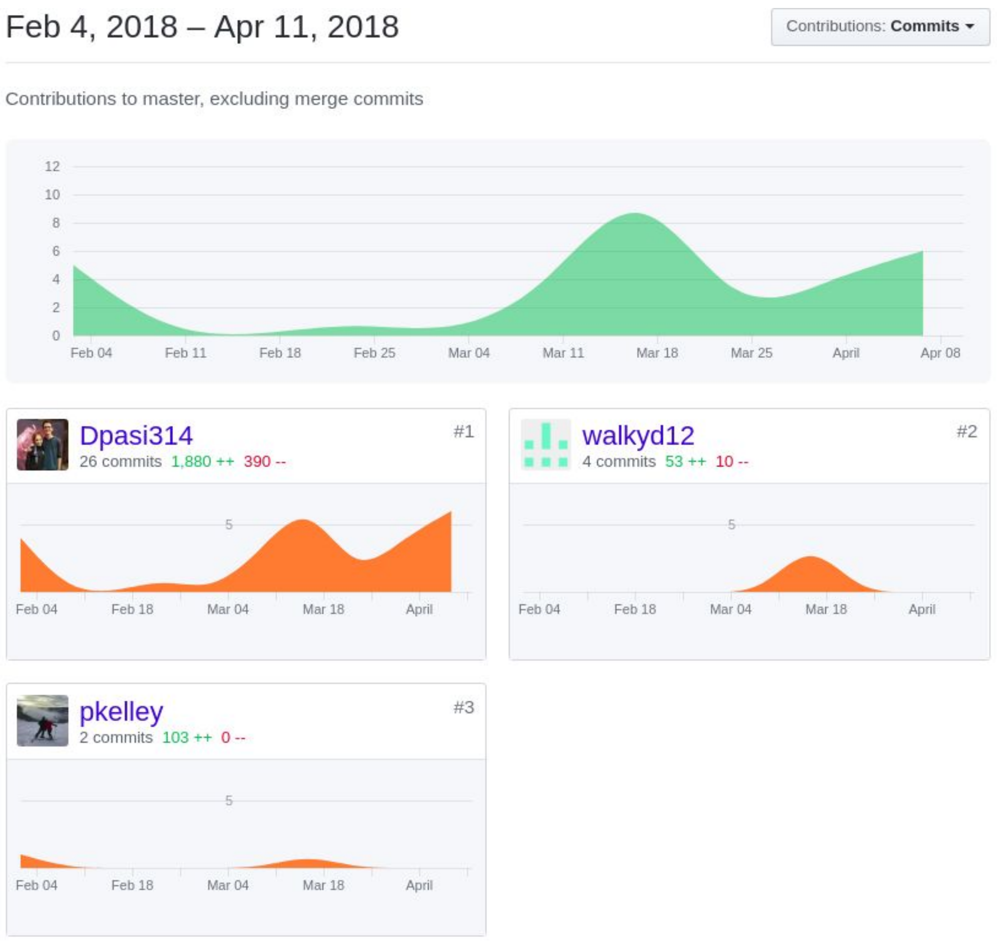


Figure 7: GitHub Graph\

Unfortunately our project had a small oversight and one of our team members has been contributing with an account not initially associated with the project as a contributor.

Proof seen here: with 3 contributors but 4 authors. It's unknown how this occurred but will be fixed in the next sprint.

Design Pattern

This project is making use of the Observer pattern. Since variables of stocks are frequently changing, it's important that we keep them updated. This means refreshing the objects with the latest information as it comes in. Using the design pattern allows use an Observer to update the price when notified. For example, if we need to update the price at the end of the day, we can simply call the observer to update this information and save it to the database. Typically, Observer is implemented using an interface and a class that implements the update method.

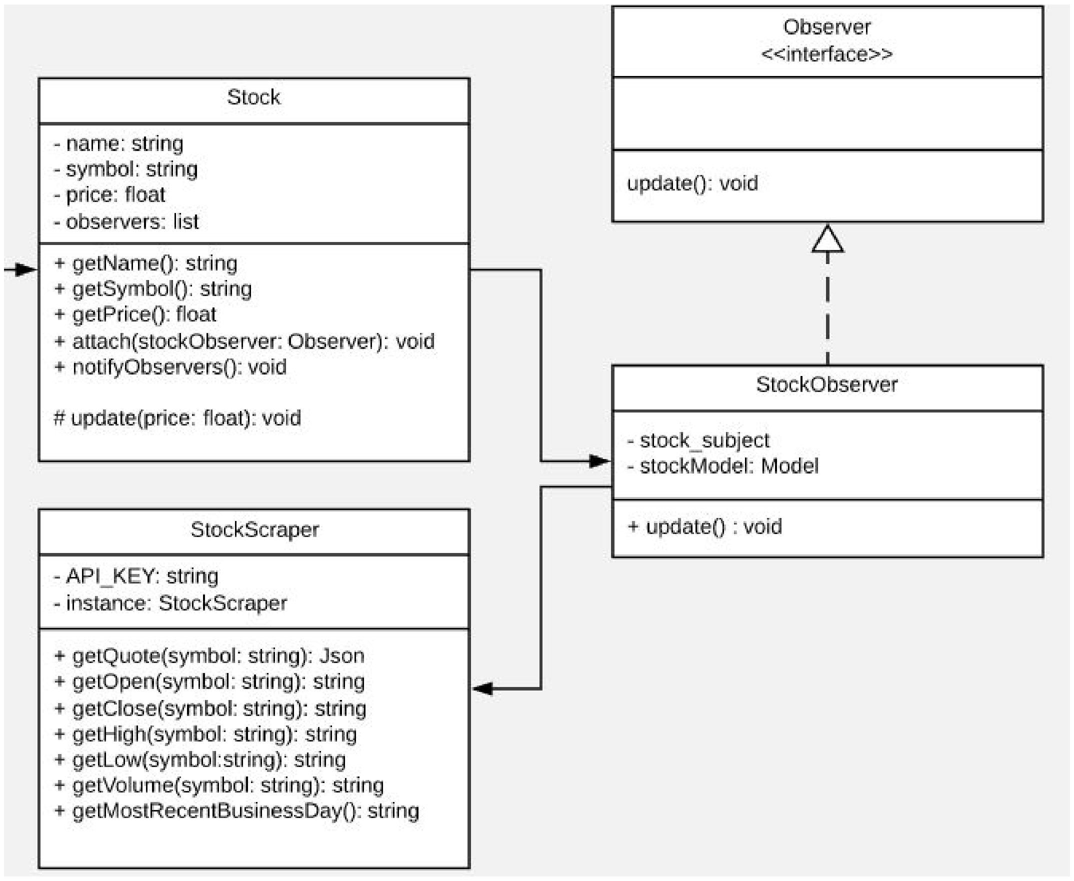

Observer
<<interface>>
update(): void
Stock
- name: string
- symbol: string
- price: float
- observers: list
+ getName(): string
+ getSymbol(): string
+ getPrice(): float
+ attach(stockObserver: Observer): void
+ notifyObservers(): void
# update(price: float): void
StockObserver
- stock_subject
- stockModel: Model
+ update() : void
StockScraper
- API_KEY: string
- instance: StockScraper
+ getQuote(symbol: string): Json
+ getOpen(symbol: string): string
+ getClose(symbol: string): string
+ getHigh(symbol: string): string
+ getLow(symbol:string): string
+ getVolume(symbol: string): string
+ getMostRecentBusinessDay(): string

Figure 8: Observer design pattern

We can see that the Stock observer implements the *update* method defined in the Observer interface. The update method calls the Stock scraper to gather any updated information. From there, the observer takes the new information and saves it to the database. It also updates the current stock object (the subject, calling the update on its observer) with the updated information.

Final Iteration

The final iteration includes many housekeeping tasks to polish the final product. Notably, the StockScraper will need to implement a constructor to take in the symbol and gather all the data then. Currently every method is making an API call. The *StockHandler* class should be refactored to *StockScraper* to reflect the true name of the class. A few more views for Portfolio, Buy and Sell will need to be created. The Portfolio class already has a working buy/sell functionality but currently User's can't see that information. This is the largest, and final milestone to hit for this project. Some small design tweaks are also in order, however won't affect the overall product. Finally making use of the prediction and strategy information is needed as a final finishing touch.

The next few weeks will be dedicated to a wide variety of tasks, hopefully intertwined, to make a fully functioning project.