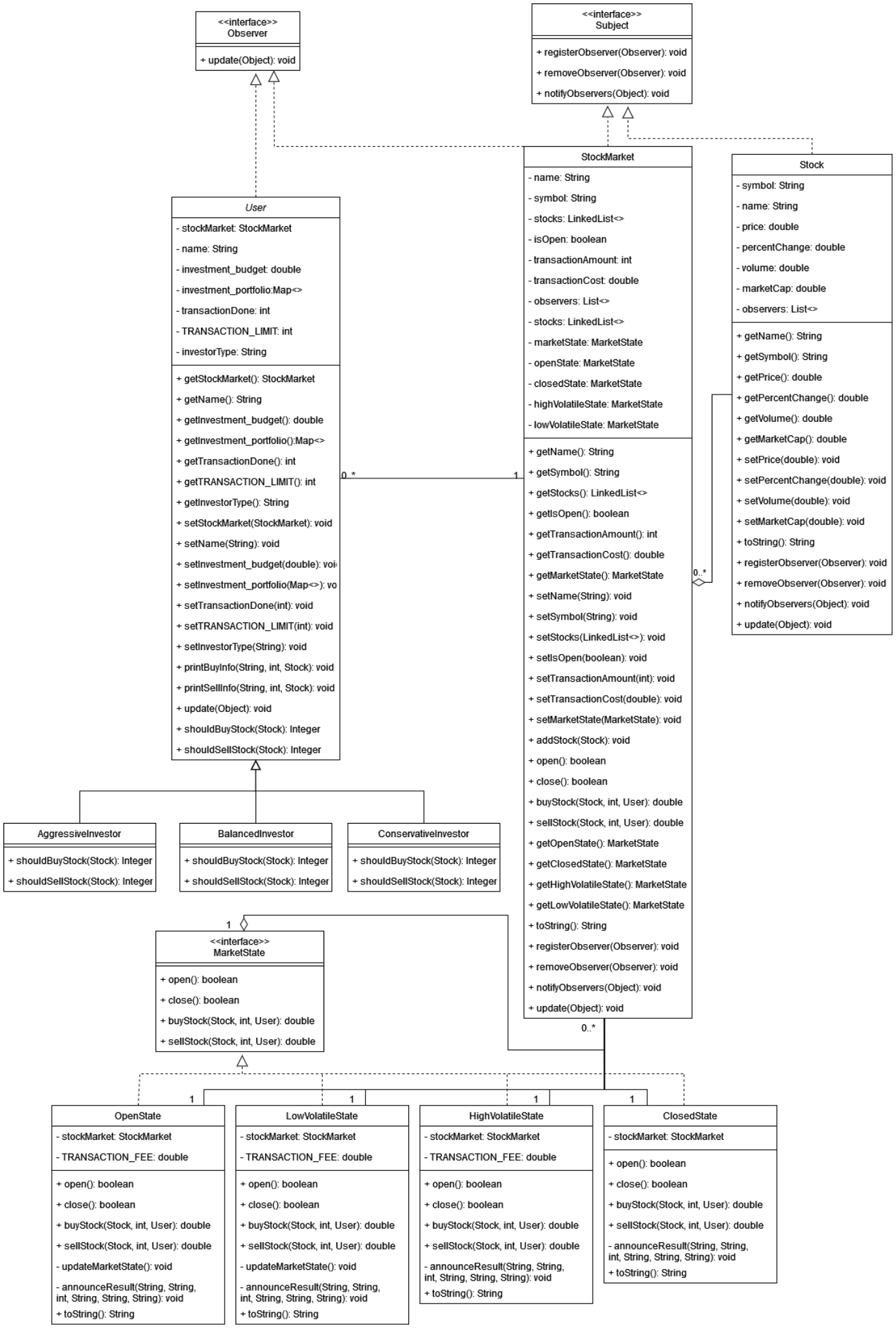
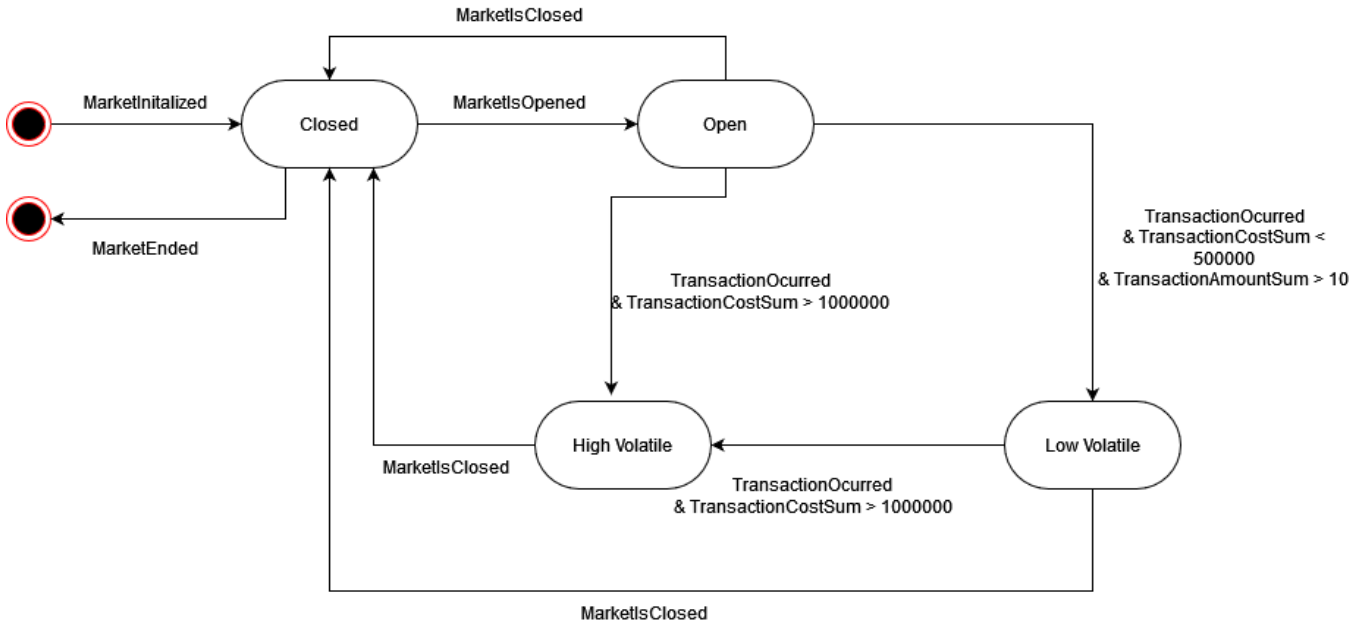
CS 319 - Spring 2023
Homework - 1

First & Last Name: Arda İynem
Department: Computer Science
Section: 01

Class Diagram



State Diagram



Design Patterns

State Design Pattern

- **Objects**
  - MarketState is the state interface

  - ClosedState is the state in which the stock market is closed and no transaction operations can be done
  - OpenState is the state in which the stock market is open and operates with 1.5% Transaction fee
  - LowVolatileState is the state in which the stock market is open operates with 0.5% Transaction fee
  - HighVolatileState is the state in which the stock market is open operates with 3.0% Transaction fee

  - StockMarket is the class utilizing the state classes and interfaces above to imitate a High Level State Machine software design

- **How**
  - StockMarket has a MarketState variable (marketState) to store the current state of the market.
  - When market is first initialized the marketState is initialized to ClosedState as in a real scenario

  - StockMarket has 4 other MarketState variables to represent states of the market (openState, closedState, lowVolatileState, highVolatileState)
  - StockMarket has a getter and setter to access and manipulate the current state (getMarketState(), setMarketState(MarketState))

  - Since each state implements MarketState they have:

    open(): Opens the state market if it is not already

close(): Closethe state market if it is not already

buyStock(): Buys a stock if possible and calls updateState() function in open and low volatile states

sellStock(): Sells a stock if possible and calls updateState() function in open and low volatile states

These transaction and market opening/closing methods are responsible for the state changing of the StockMarket as they also exist in the StockMarket object. Since in open and low volatile states it is possible to change their state to other than closed state they also have an updateState() function.

- In conclusion, certain functions of StockMarket behave according to its state in which these functions are defined differently for each state.

- **Why**
  - For the sake of better readability and writability the code implementers, developers and users need to understand the code clearly. As other design patterns aim to produce a clear, easy to understand and an organized code; State Design Pattern also aims these criteria.
  - Eventually the resulting code is clearer and way more organized besides the code's similarity to the real life scenario which increases understandability.
  - Moreover many nested decision statements for different cases are avoided with this pattern. The better encapsulation helped to simplify the complex and crowded structure

Observer Design Pattern

- **Objects**
  - Observer is the observer interface with update(Object) method.
  - Subject is the subject interface with registerObserver(Observer) removeObserver(Observer) notifyObservers(Object) methods.

  - Observer implementers (StockMarket, User) override the update function to define what happens when it is notified upon a change in Subject
  - Subject implementers (StockMarket, Stock) have an ArrayList<Observer> to store its observers which will be notified upon a change. They also override the registerObserver, removeObserver, and notifyObservers functions to add, remove, and call update functions of the observers respectively.

  - Stock implements the Subject interface since StockMarkets are the observers of Stocks and need to be alerted to the changes of stocks.
    NOTE: This may not be one of the objectives of the homework yet in real life stocks may be traded in multiple markets so this solution is scalable compared to one to many relationship between stock markets and and stocks respectively.

  - StockMarket implements both the Subject interface and the Observer interface since StockMarkets are the observers of Stocks and the Users are the observers of the StockMarkets. Namely StockMarkets are listening the Stocks while the Users are listening the StockMarkets

    - User implements the Observer interface since Users are the observers of StockMarkets. Namely Users are listening the StockMarkets in case any change happens in any stock belongs to the StockMarket

- **How**
    - When a Stock is created and added (with addStock(Stock) of StockMarket) to a StockMarket registerObserver(Object) method of the Stock is called to add the StockMarket to the observer list of the Stock.
    - When a User is created, in its constructor registerObserver(Object) method of the Stock is called to add the User to the observer list of the StockMarket as the StockMarket is one of the constructor parameters of User.

    - When the notifyObservers function of the Stock is called by itself, the update function of the StockMarket is called in notifyObservers. Since the update function of StockMarket is implemented to call notifyObservers function of itself then update function of each User belonging to that StockMarket is called.
    - In conclusion this chained call mechanism means that whenever a change happens in a Stock each user of each StockMarket having that Stock is alerted.

- **Why**
    - For the sake of better readability and writability the code implementers, developers and users need to understand the code clearly. As other design patterns aim to produce a clear, easy to understand and an organized code; Observer Design Pattern also aims these criteria.

- Eventually the resulting code is clearer and way more organized besides the code's similarity to the real life scenario which increases understandability.
- Moreover a better abstraction mechanism is provided with this design pattern since access to different objects is minimized to a few functions and coupling is significantly lessened. Also the unnecessary checking of the listeners is avoided by a one-to-many notification system.