611 Final Project Design Document

Team Members:
- Justin Sayah (jsayah@bu.edu U67474020)
- Junru He (hjryy@bu.edu U44804866)
- Haiwei Sun   (hsun1239@bu.edu U66364483)
- Shangzhou Yin (syin10@bu.edu U63027471)

Github Repository Link: https://github.com/justin-sayah/611FinalProject

Implemented Classes

1. **Interfaces:**
    - Account: it holds four methods containing ***void deposit(), void withdraw(), double getBalance(), int getAccountNumber()***
    - Tradeable: it holds two methods containing ***double getPrice(), void changePrice(double price)***

2. **Classes:**
    - **Customer:** this class extends from the abstract class Person and is used to instantiate the a customer object with ***parameters(i.e. Id, firstName, lastName, username, password, dob, ssn).*** The static methods connect with the database class PeopleDao and can be used to retrieve customer objects based on the personId or username and password.

    - **SystemDriver - concise main to run program**

    - **StockTradingSystem - has one routine to draw login screen, entry point for whole system**

    - **Manager:** this class extends from the abstract class Person and is used to instantiate a manager object with ***parameters(i.e. Id, firstName, lastName, username, password, dob, ssn).*** The ***static methods(i.e. static void blockAccount() and static void unblockAccount())*** connect with the database class TradingAccount to block and unblock customers classes. The manager is also able to send messages to customers based on customerID in the ***void sendMessage()*** method.
    - **Market:** this class used singleton design pattern and is able to get all stocks related informations such as **List<Stock> getAllStocks(), List<Stock> getAllUnblockedStocks(), boolean isBlocked() and Stock getStock()**. The

market is also able to ***fetch the real life stock price*** which is implemented in the PriceFetcher class.

- **Message:**this class is used to instantiate the Message object with the parameters***(i.e. messageId, personIdFrom, personIdTo, message).***

- **Message Center:** it is implemented with the singleton design pattern and allow customers to get messages, delete messages with the method ***List<Message> getMessagesInInbox()*** and ***void deleteMessage()*** and managers to send messages void with the method ***void sendMessage().***

- **Person(abstract class):** the abstract class is used to extend to different types of persons such as customer and manager in this project. It holds ***parameters(i.e. Id, firstName, lastName, username, password, dob, ssn).***

- Position: this class tracks the position of each stock that customers hold with the ***parameters (i.e. accountID, securityId, quantity, quantitySold, currentPrice, avgBuyPrice, realizeProfitLoss, unrealizedProfitLoss).*** IIt calculated unrealized profit/loss and realized profit/loss at each position and allows customers to sell and buy stocks with the method ***void buy() and void sell()*** and update the unrealized/realized profit/loss and balance in the TradingAccount database.

- **PriceFetcher:** this class implemented the API to fetch the **real stock price** and update into out database from this website: https://telescope-stocks-options-price-charts.p.rapidapi.com/stocks/

- **Security(abstract class):** this is an abstract class to be extended to different types of stocks such as stock in this project. It holds ***parameters name, securityId, price.***

- **Stock:** it is extended from Security class as one type of securities and used to instantiate stock objects with ***parameters stockId, name, price, ticker.*** In this class, we implemented a method ***void changePrice()*** for managers to manually change stock's price and it will be updated into the database StockDao.

- **TradingAccount:** it implements Account interface and is used to instantiate a trading account object with ***parameters(i.e. accountNumber, personId, balance, unrealizedProfitLoss, realizedProfitLoss).*** Its static methods are able to get account related information from the database TradingAccount such as ***TradingAccount getAccount(), TradingAccount getAccountNoRefresh, List<TradingAccount> getAllActive(), List<TradingAccount> getAllPending(), void update(), void delete(), void deleteFromPending(),***

void addFromPending(), void addTradingAccount(), void addPendingAccount(), List<TradingAccount> getAllBlocked(), List<Positions>getAllPositions()

- **Transaction:** this class is used to instantiate a transaction object with the **parameters*(i.e. transactionId, accountId, stockId, quantity, price, timestamp, action).*** It connects with the database class TransactionDao and is able to retrieve the transaction with ***static methods(i.e.List<Transaction> getAllTransactionByAccount(), List<Transaction> getAllTransactionByStock(), void(addTransaction()).*** Based on the above features, customers and managers are able to see transactions by stock or by account.

3. **UI Classes:**
    - **AddStockFrame:** manger can use this class to add new stocks
    - **ApproveAccountFrame:** manager can approve a new account that is requested by the customer
    - **BlockAccountFrame:** manager can block a trading account
    - **BlockedAccountPage:** display all blocked trading accounts
    - **BlockedAccountsWithdrawPopup:** money can be withdrawn from the blocked trading accounts
    - **BlockSellPage:** blocked trading accounts are able to sell stock but not to buy stock
    - **BuyConfirmPopup:** confirmation page after buying stocks
    - **BuyStockPage:** customer buy stock via this page
    - **CustomerAccountView:** display all informations about a trading account of a customer
    - **CustomerHomePage:** display all accounts of a customer
    - **CustomerInformationFrame:** display all customers information for manager
    - **DeleteAccountFrame:** manager can delete trading accounts
    - **DepositPopup:** deposit money into the trading account
    - **LoginFrame:** customer and manager login via this page
    - **ManageAccountFrame:**manager can manage accounts in this page
    - **ManageMarketFrame:** manager can manage markets(stocks) via this page
    - **ManagerFrame:** display the main page of a manager account
    - **MessagePopup:** display messages sent from manager for customer
    - **SellManageFrame:** the customer can sell stocks through this page
    - **SignUpFrame:** new customer can sign up a new account in this page
    - **TransactionHistoryFrame:** display all transactions for customers or managers
    - **UnblockAccountFrame:** manager can unblock a trading account
    - **ViewAccountsFrame:** manager can view accounts of a specific customer
    - **ViewActiveAccountFrame:** manager can view all active trading accounts
    - **ViewBlockedAccounts:** manager can view all blocked trading accounts
    - **ViewBlockedStocksFrame:** manager can view all blocked stocks

- **ViewBlockSellStockTransaction:** manager can view all blocked stocks transactions
- **ViewPendingAccountFrame:** manager can view all pending trading accounts
- **ViewPositionsFrame:** manager can view positions of a trading account
- **ViewSellStockTransaction:** customer can view all transactions of stocks that have bought
- **WithdrawPopup:** the popup window for customer to withdraw the money from a trading account

4. **Database Classes:**
    - **(Interface)AccountDao** - An interface with behaviors that all DAOs that deal with Account types must implement, includes fetching and storing Accounts
    - **DatabaseConnection** - a simple wrapper class to hold a connection to an arbitrary database, in our case SQLite DB
    - **MessageDao** - Class that contains all the CRUD operations that involve Message objects, essentially an interface for operations on the SQLite DB for Messages
    - **PeopleDao** - Class that contains all the CRUD operations that involve Managers and Customers in the same way as the other DAOs. Holds responsibility for fetching and saving objects of type People from the database
    - **PositionDao -** class that contains all the CRUD operations involving Position objects on SQLite database, including reading/writing data and object creation
    - **StockDao -** class that contains all the CRUD operations involving Stocks. Fetches and saves Stock objects from SQLite database.
    - **TradingAccountDao -** implements AccountDao specifically for TradingAccounts and allows for CRUD manipulation of TradingAccount objects with the SQLite database
    - **TransactionDao -** class that contains all the CRUD operation behaviors involving Transaction objects on our SQLite database.

5. **Design Choices**:

In this project, we chose to implement a singleton design pattern for classes including Market class, StockDao, MessageDao, PeopleDao, TransactionDao, TradingAccountDao, and PositionDao. This allows only a single object created and can be accessed directly without need to instantiate the object of the class. All the Dao class files contain CRUD operations on the SQLite database, so if we make them into singleton, when customer and manager fetch data from the database, this would allow them not be able to access the database directly, which protects the database to some extent.

As to our design for block and blocking stocks, instead of incorporating this state as part of the base objects, we decided to use the tables of our database to indirectly store the state of that particular state of account. For instance, if a Stock were to change state from unblocked to blocked under the command of the Manager, it would move from being held in the stocks table to the blockedStocks table. This dictates what functionality it has and also how other classes can now interact with this stock. When the market of buyable stocks is displayed to customers, only stocks from the stocks table are included, meaning that blocked stocks cannot be purchased anymore. However, when selling stocks, all stocks that are currently held by that user are displayed, which will even include the blocked stocks. Therefore, the behavior of blocking is done through the modular storage of these stock objects in our database. The same thing occurs for blocked accounts. Blocked accounts are unable to be deposited into and cannot buy new stocks.

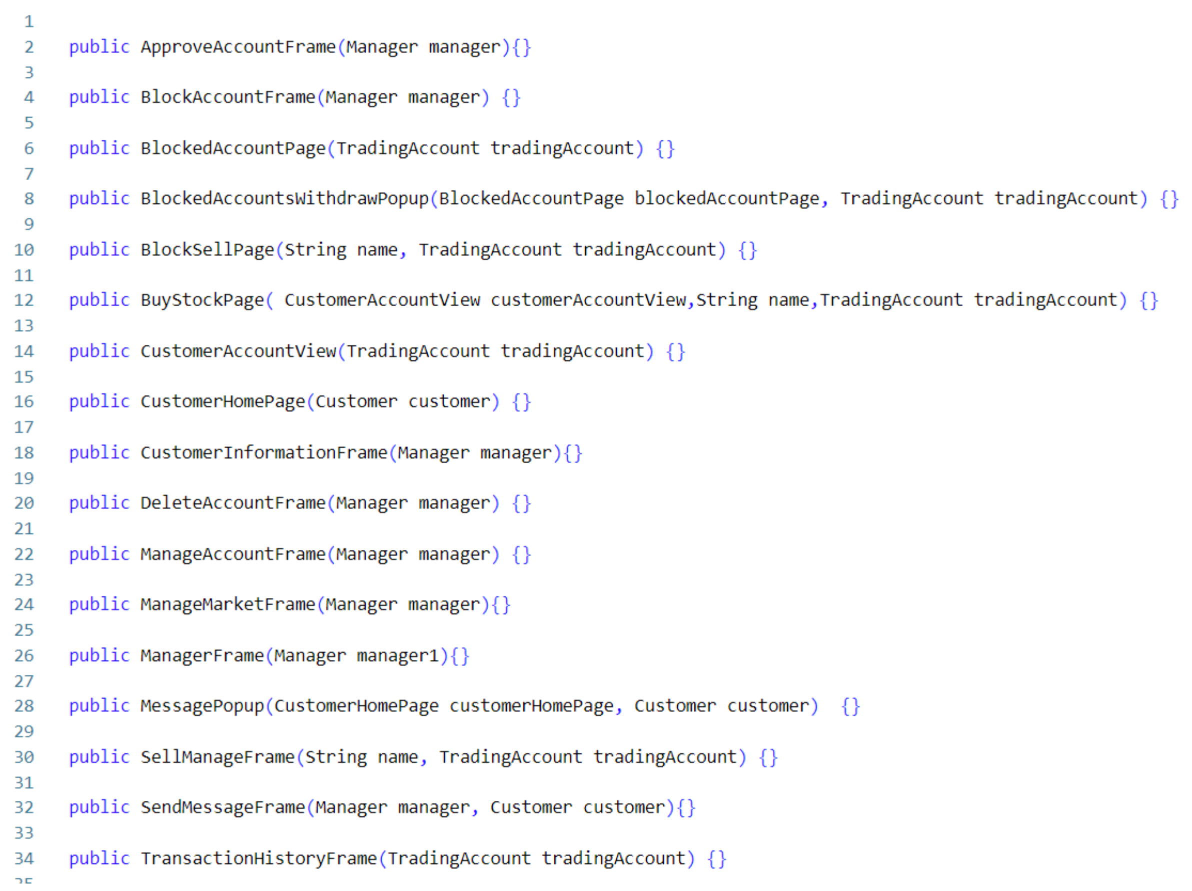
6. Object Model:

Information about all of the objects in our object model can be found above in the implemented classes section. There is an included UML class diagram in the repository as well for showing connections between the classes that were not explicitly explained above.

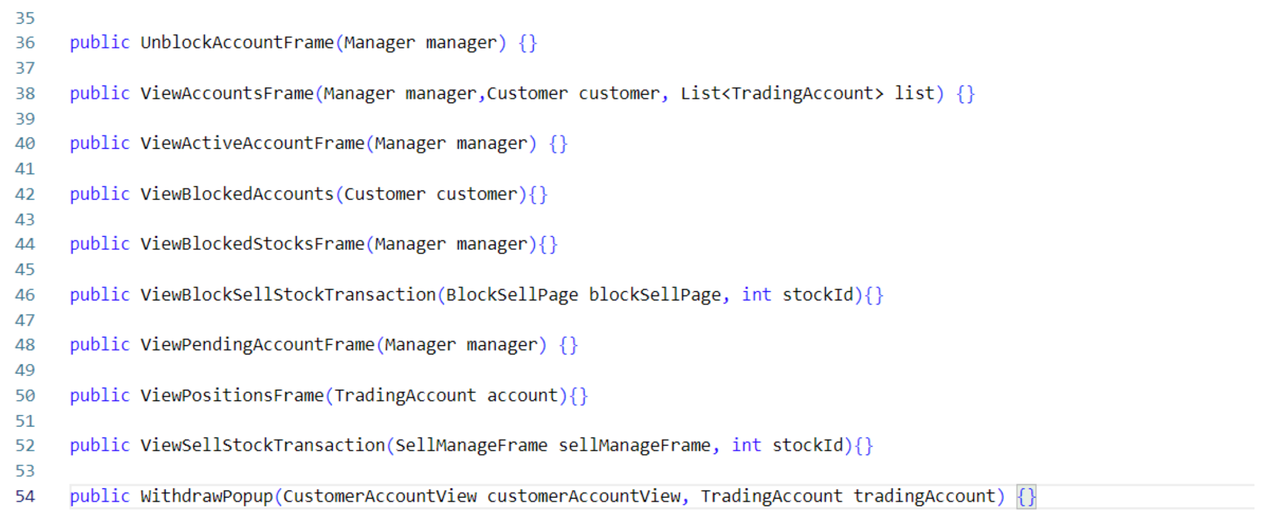
7. Object and GUI Relationship:

Instead of designing a RESTful API to facilitate communication between the frontend and backend, we decided to have the object model be more directly incorporated into the GUI classes. The idea generally was to allow instances of the windows to hold state about the objects that they were trying to display directly. This would allow the UI easy access to backend routines since the objects that held state are themselves full Java objects that hold routines necessary for the fetching of new data and critical operations. An example of this could be the Account View Page. When a Customer clicks on an Account they want to view, the Customer object of that class has the routines within it to fetch the Trading Accounts that are related to that Customer. So what that allows us to do is to create the Account View Page with the specified Trading Account that is requested by passing it to the constructor for that page as a parameter. Therefore the Page will be drawn modularly by fetching data from the given TradingAccount that it is now holding in its state. Essentially that frame of the UI is holding the object that it is supposed to be representing. The methods of those objects gives the UI a way to interact with the object model and the database indirectly. Continuing this example, once the UI page for drawing a Trading Account View is done, there is a button that allows a user to view the Positions that are being held in a Trading Account. The TradingAccount object has behaviors that allow a client to get all related Positions for that trading account. So by passing this object along to the UI page which intends to allow users to view positions of an account, the page will have access to all the related Positions for that TradingAccount and draw them on the screen.

We took the objects created by the above classes section as the parameters into the GUI classes(UI classes). The detailed implementation of each GUI Object is as follows.



```java

public ApproveAccountFrame(Manager manager){}

public BlockAccountFrame(Manager manager) {}

public BlockedAccountPage(TradingAccount tradingAccount) {}

public BlockedAccountsWithdrawPopup(BlockedAccountPage blockedAccountPage, TradingAccount tradingAccount) {}

public BlockSellPage(String name, TradingAccount tradingAccount) {}

public BuyStockPage( CustomerAccountView customerAccountView,String name,TradingAccount tradingAccount) {}

public CustomerAccountView(TradingAccount tradingAccount) {}

public CustomerHomePage(Customer customer) {}

public CustomerInformationFrame(Manager manager){}

public DeleteAccountFrame(Manager manager) {}

public ManageAccountFrame(Manager manager) {}

public ManageMarketFrame(Manager manager){}

public ManagerFrame(Manager manager1){}

public MessagePopup(CustomerHomePage customerHomePage, Customer customer)  {}

public SellManageFrame(String name, TradingAccount tradingAccount) {}

public SendMessageFrame(Manager manager, Customer customer){}

public TransactionHistoryFrame(TradingAccount tradingAccount) {}
```



```java

public UnblockAccountFrame(Manager manager) {}

public ViewAccountsFrame(Manager manager,Customer customer, List<TradingAccount> list) {}

public ViewActiveAccountFrame(Manager manager) {}

public ViewBlockedAccounts(Customer customer){}

public ViewBlockedStocksFrame(Manager manager){}

public ViewBlockSellStockTransaction(BlockSellPage blockSellPage, int stockId){}

public ViewPendingAccountFrame(Manager manager) {}

public ViewPositionsFrame(TradingAccount account){}

public ViewSellStockTransaction(SellManageFrame sellManageFrame, int stockId){}

public WithdrawPopup(CustomerAccountView customerAccountView, TradingAccount tradingAccount) {}
```

8. Benefit of the Design:

We designed an interface as Account with basic methods that all types of accounts should have these properties such as **void *deposit(), void withdraw(), double getBalance(), int getAccountNumber(),*** which then can be implemented by specific accounts. For instance, the stock trading system needs customers to have trading accounts, so a trading account class can implement this interface and add more specific trading account features. In future, if we design other systems such as bank systems, then it would be easy to have a bank account to implement this Account interface, which meets the abstraction, extensibility and reusability of Java OOD. So, a tradable interface is designed as well.

Dao class files are all wrapped up based on the singleton design pattern and it restricts the instantiation of the class, and thus users can only access the method through this instance. As a result, the system is more effective and less likely to experience mistakes brought on by using many instances of the same item.

Data retrieval and manipulation are made simple when information about clients, trading accounts, stocks, and transactions is stored in different databases. This promotes data consistency and facilitates scalable data management.

To simulate the stock market close to real life, API was introduced to fetch the real life stock price from the website(this is explained in the above PriceFetcher class) and store it in our database. When the update real life price is clicked, it will update the price of selected stock immediately.

Finally, the connection between objects and GUI classes enables simple communication between the system's front and back ends. The fact that the UI classes save state about the items they are attempting to show directly makes it simpler to access backend procedures since the state-keeping objects are themselves full Java objects that contain routines required for essential actions and the retrieval of fresh data. As a result, a system that is easier to maintain and upgrade and is more modular and scalable is made possible.

9. Appendix: UML

The UML is too large, so we put a clear version UML diagram in our Github repository.