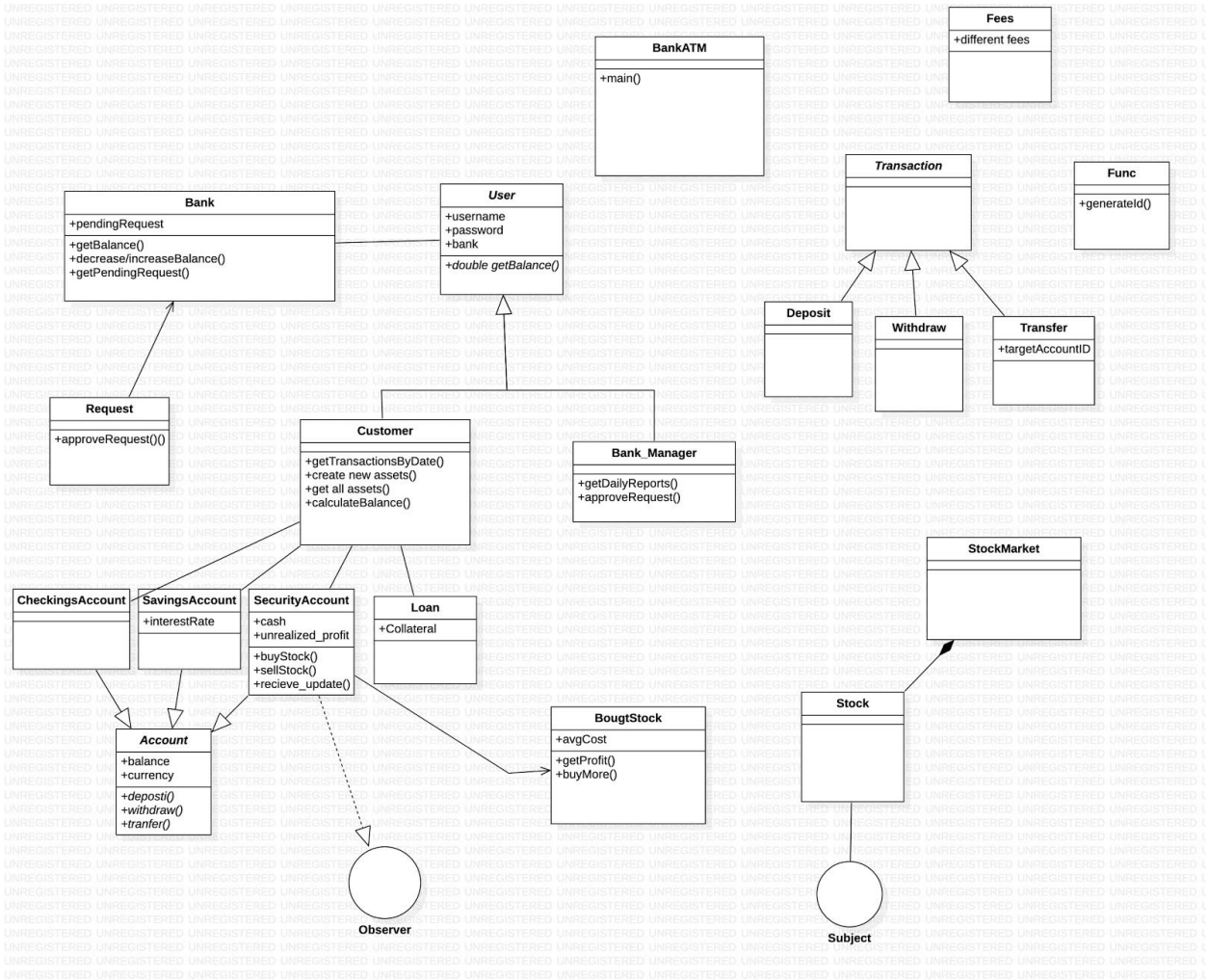
Back-End Classes



Loan: Loan made by the Customer
Func: helper functions like generate_id()
Fees: put all the fees as final static, allowing the ability to be changed easily

Request:
-approveRequest()
Bank
-balance
-increase/decrease balance
-addRequest
-pendingRequest

Intention:

The bank class is used to keep track of the balance of the bank and the pending request. Only the manager can see the balance and the pending request.

abstract class User

-access to bank

-specifies things like name, username and password

Customer extends User

-able to create accounts and see all accounts, balance, currency of the accounts

Bank_Manager extends User

-able to get Daily Reports

-ability see bank balance

-ability to see pending request in bank

-ability to approve request

Intention: Bank managers and customers are all users, the user classes enable functionalities like logging in with username and password. Bank managers can see the bank balance and approve requests using access to the bank attribute. Customer class use the bank attribute to make the bank process fees(increase/decrease balance in the bank, ect).

Subject Interface

-updatePriceChange(double newPrice);

-updateShareChange(int newShare);

Observer Interface

-receive_updatePrice(int stockId, double newPrice);

-receiveUpdateShare(int stockid, int newShare);

Intention: Used for observer's pattern between Stock class and SecurityAccount. When the bank manager makes a price or share change, SecurityAccount would be notified.

abstract class Account

-Has a currency

-abstract withdraw(amount, currencyType);

-abstract deposit(amount, currencyType);

-abstract transfer(amount, currencyType, transferToAccount);

CheckingsAccount extends Account

SavingsAccount extends Account

-Requires a minimum balance

SecurityAccount extends Account implements Observer

-enables buying and selling stock
-receives update from stock price change

Intention: checking, savings and security accounts are all accounts, they share commons like having accountid, balance and userid and Currency. They also share the same functionalities like deposit, withdraw and transfer, though implemented in different ways because some might have restrictions.

Currency
CurrencyType(enum class)

Intention: Each user has a default currency, a user can make transactions in different currency, that amount is converted to the user's default currency. All currency is defined in CurrencyType, which allows for scalability.

abstract class Transaction
TransactionType(enum class)
Transfer extends Transaction
Deposit extends Transaction
Withdraw extends Transaction

Intention:
A transaction is merely used for recording transaction class to display on the front end, the only difference between Transfer, Deposit and Withdraw is the toString() method.

StockMarket
-Used the Singleton Pattern because only one stockMarket exits
-keep tracks of all stock in the market
-has method to create, update stocks
Stock
-stock in the market
-has subscribers(securityAccount that bought this stock) and able to update the subscribers of price change.

BoughtStock
-stock owned by the user implements Subject
-keep track of average price the user bought the shares

-can calculate profit using currentPrice

Intention:
The bank manager has the ability to create and update change to stocks in the stock market. The BoughtStock class was created because we wanted a way to keep track of the average cost and equity in order to calculate profit. When a bank manager updates the stock price, the stock then updates all the subscribers - security accounts that owns the stock of the change.

Front End:

SignLog: Startup page for the app - gives the option of either creating a new account or logging in
LoginDemo: Login directory - either the manager (only one with specific credentials) can sign in or the customer
LoginView - Login form for user - must know their user id in addition to their username and password.
ManagerLogin - Login form for manager

Intention: In order for the user to actively use the app and navigate through it - there must be a rigorous authentication process.

CustSaveAccount - Customer's saving account. If they have enough money, they can transfer $1000 minimum to a securities account but must maintain $2500 in their current account.
CustCheckAccount - Customers checking account. Does the same thing as a savings account except allow securities transfer.
ManagerAccount - The manager account - allows the manager to see all that is going on with his customers.
StockViewCust - the securities account page that allows users to play in the stock market

-Also any time an account is open or closed or a checking account makes a transaction or withdrawal - there is a fee that is paid. Also customer can choose what currency to deposit or withdraw from.

Intention: Many different users can have several different accounts, which is why there needs to be multiple places/pages where the user can sign into.There is only one manager account so we don't need to worry too much about having to create multiple different accounts.

CustDeposit - a form where users can deposit any amount of money. However, there is a fee for checking account users.
CustWithdraw - a form where users can withdraw any amount of their money. However, there is a fee for checking account users.

LoanReq - a form where users can request a loan. However, there is a fee for checking account users.
CustProfile - a form allowing the user to change any of their information.
CustTransfer - a form allowing savings users to transfer at least $1000 to a securities account if they have at least $5000.
CustTransactions - allows the customer to see all the transactions they have made according to each day.
CustomerLists - a page showing all the other accounts the customer has.
CustomerStocks - shows the stocks that the customer owns.

Intention: Both securities account users and checkings account users have access to some of the same classes like deposit and withdraw since both accounts use the same functions for them. The difference is the stocks in which Securities accounts uses the stock market.

DateReport - a page showing the manager all transactions made on that day
Manager Report - a form allowing the manager to enter the date of the transactions he wants to see
StockViewManager - allows the manager to see what stocks are on the market
ChangeStocks - allows the manager to change the price of the stocks put up
ManagerSearch - allows the manager to search out any user he wishes, depending on their user id.
ManagerSees - a page showing the information on that customer's transactions
SellStocks - allows the customer to sell his or her stocks.
UnrealizedProfit - allows to user to see their unrealized profits on whatever stock they wish to see. Realized profit is used when the customer buys a stock
BuyStocks - allows the user to buy a stock from the stock market

Intention: Both the user and manager can see the stock market, but while the customer can only buy and sell stocks, the manager can change the prices and does not need to pay for stocks.

All GUI interfaces contain actionListeners which act as the controllers moving between pages.

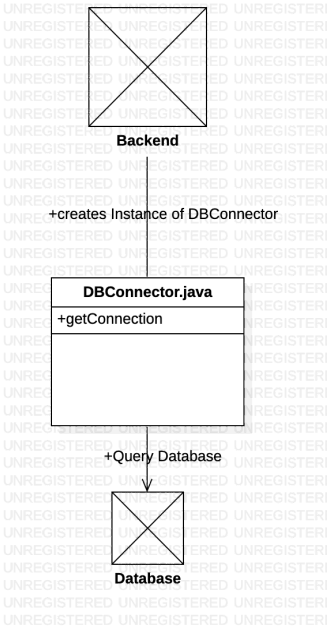
Database

For the database we decided to utilize a mySQL relational database to store our data. We decided to use a relational database because by having unique identifiers for the different elements in our database we would be able to make sure that no two rows in the database would be the same thus resulting in efficient queries. We selected mySQL specifically because that was the one we had the most familiarity with. Another benefit of using a relational database over just writing and reading text files is that we were able to host the database AWS allowing for easy and consistent access across developers.

DBConnector.java

- getConnection() connects to the database.

Intention:
This class is meant to serve as a layer between the backend and the database. It contains the functionality to connect to the database (which is hosted on AWS) and utilizes the JDBC (Java Database Connection) object to create this connection and run all of the queries. For easier access to the backend, DBConnector class provides methods for each query that can be accessed once an instance of the DBConnector class is initialized (they are not static methods). When an instance of the DBConnetor class is initialized it establishes a connection to the database. By implementing the class in this way each query does not need to make its own connection to the database but it can instead use the connection for which that instance has already been established.



Improvement on Design for connecting to the Database:
In the current implementation of DBConnector and connecting to the database all of the queries can be found inside the methods of the one file DBConnector.java. This caused a mostly unorganized file where we just kept adding queries as needed. A better implementation would take a more Object-Oriented approach to the system. This could be done by making DBConnector.java a super class then having a class for each table in the database that extends DBConnector. For example, to access the transaction table in the database we would have a class called DBTransactions which would extend DBConnector. When the call is made to the super constructor we would then establish the database connection however DBTransactions would contain the methods for the queries related to the transaction table. By applying object oriented design principles here we are able to create a more organized and efficient system.