Documentation of the classes you implemented

Front-End:

ManagerPortfolio.java

- Utilizes the observer by implementing ButtonObserver and proxy pattern. HomePage is the Subject that keeps track of every window open in the application. When notified by any observer that it has done a action that modifies the database, it tells all observers to re-read from the database, ensuring no stale data is used. So when ManagerPortfolio makes a change to the date, all windows reflect that change. ManagerPortfolio also utilizes Proxy pattern's benefits, as unlike CustomerPortfolio, it has direct access to the Bank object allowing for methods that wouldn't be available in the ATM.

CustomerPortfolio.java:

- Page where customer can do every action specified in ATM. Utilizes the observer by implementing ButtonObserver. The proxy pattern allows us to prevent the front-end from viewing the specifics of what is going on in the back-end and abstracts what is going on. This further adds protection as CustomerPortfolio has no access to the bank object, enabling different actions to be performed between CustomerPortfolio and ManagerPortfolio as ManagerPortfolio has direct access to the bank object, allowing for more options in ManagerPortfolio.

HomePage.java:

- Utilizes the observer pattern by implementing ButtonSubject. HomePage is the Subject that keeps track of every window open in the application. When notified by any observer that it has done a action that modifies the database, it tells all observers to re-read from the database, ensuring no stale data is used.

ButtonObserver & ButtonSubject.java:

- Interfaces used for Observer pattern on ManagerPortfolio, CustomerPortfolio, and HomePage.java.

RegisterWindows.java:

- Simple register window that uses Login.java's register method.

ManagerLogin & CustomerLogin.java

- Simple login page that checks with Login class for authentication. Manager has no option to register as that would mean anyone could create an account and become a manager.

Back-End:

Bank.java

- Uses Proxy pattern & singleton pattern, where the front end will only have an instance of ATM and never an instance of the actual bank. All methods in this class are called by ATM.java and never called from the front-end. It is also a singleton to avoid passing instances of Bank to every Account, Customer, etc. As an Account by itself has no reference to other existing accounts, same for customers and so on. This program was designed with one bank in mind.

ATM.java

- Uses Proxy pattern, where the front end will only have an instance of ATM and never an instance of the actual bank. All methods will be called on the ATM. This was done to

provide a layer of abstraction and protection for the end user. All methods have a return type to indicate to the front end if method succeeded in doing its job

BankInterface.java

- Uses Proxy pattern, this interface ensures that all methods supported by ATM are properly implemented into Bank as well.

Account.java

- Abstract class for all types of accounts, and handles operations performed by all of these accounts such as withdrawing, depositing, and grabbing total balance (all currencies combined)

CheckingAccount.java

- CheckingAccount has no special properties and its only distinction is its accountType. The file exists to support future implementations that could be added CheckingAccounts if needed. The withdraw fee is implemented in Bank rather than CheckingAccount as CheckingAccount by itself has no reference to other users in the bank. All transactions are handled in bank as bank has the reference to all users in the system.

SavingsAccount.java

- SavingsAccount differ from CheckingAccount as every time a user deposits money or withdraws money, we must check if it enables or disables their security account. It overrides Account's withdraw and deposit methods and calls not only account's withdraw and deposit but performs the additional checks as well

SecurityAccount & SecurityObserver & SecuritySubject.java

- Utilizes the Observer Pattern. SecurityAccount supports all methods callable from a SecurityAccount such as buying and selling stocks. It implements SecurityObserver, meaning when a manager that implements SecuritySubject calls a method regarding a stock a portfolio owns (such as selling or removing), it lets all securityAccounts know that it should update its unrealized profits or sell off their stocks. SecurityAccount is always in USD as well and therefore have a different structure (only USD of currencies is being used from Account)

User.java

- Abstract class extended by Manager and Customer and holds shared attributes and methods for both classes

Customer.java

- A basic class that contains the attributes of Customer. Customer has accounts and allows for a single customer to access multiple accounts associated it at the same time

Manager.java

- A manager similar to Bank is singleton. And unlike customers in the front end, managers when logged in the front end have methods that the manager can directly call towards the bank, such as changing date, changing stock prices, etc. By having Manager have access directly to the bank, we can make Manager call methods not available on the ATM (which customers have access to), creating an additional layer of security. Implements Observer Pattern, where if a manager updates or removes a stock, it should notify all portfolios under the manager of their new unrealized profits (if updated) and sell all stocks (if removed). Without this pattern, a customer would constantly have to see if Manager has made any changes to the database.

Currency.java

- Currency.java exists to simplify the creation of money where instead of always dealing with a pair of objects (Currency Type, Amount), we can pass in a object of type Currency. Currency.java also includes helper methods regarding currencies.

Loan.java

- Loan class made to avoid having large tuple of data and instead one object class.

Stock.java

- Helper class to avoid pairs of data and for clear distinction of what a stock is. Represents a stock in the market

StockDetails.java

- Helper class for creating StockDetails objects. This differs from Stock.java as Stock.java is used for displaying current price for displaying on the stock market. StockDetails is used by the Customer and keeps track of what price a customer bought a number of stocks at (this is necessary for) accurate unrealized profits and realized profits calculations. Having one price won't work (i.e. Customer buys 1 share at 50$, another at 100$, and the stock is now worth 200$.) Profit calculation would need a history of both prices.

StockPortfolio.java

- Helper class consisting of methods for calculating a value of a StockPortfolio of a Customer. StockPortfolio requires too many bookkeeping to keep an internal state and updating to the database at the same time. Instead, we always get and write to the database

Transaction/StockTransaction.java

- Helper class consisting of attributes representing a Transaction/StockTransaction. This helps with grabbing and writing from the database a single instance of a Transaction/StockTransaction rather than having tuples

Main.java

- Main class to start our code.

CustomerDatabase.java

- Database class regarding all things customer related especially accounts. All persistent data is saved in to a CSV. There may seem to be a lot of repetitive code, but it is necessary as every CSV has a unique structure to it and each method needs all the boiler plate code with minor modifications in most cases.

LoanDatabase.java

- Database class regarding all things customer related especially accounts. All persistent data is saved in to a CSV. There is a lot of boiler plate code but it is required as each method has small differences that make writing a method very difficult

ManagerDatabase.java

- Gets current instance of manager which contains all instances manager should know about such as all customers, all transactions, and all stocks. Also supports manager login.

StockDatabase.java

- Database class regarding all things stock related which includes operations from the manager changing prices to what price a customer bought a stock at

TransactionDatabase.java

- Class in charge of writing and reading all transaction related data to the database.

Explaining your design choices & Benefit of your design.

Patterns

- Singleton Pattern: Singleton was used for manager and bank. It didn't make sense where there would be 2 different banks on the same ATM system. This goes same for the manager. Where if we had more than one manager, we would have to specify how customers pay which Manager how much. Manager can instead log in to the same account with more than one username and password. With more than one bank, it would unnecessary bloat code constantly passing in a bank to every class.
- Proxy Pattern: Proxy pattern used on Bank to create ATM to not only abstract and simplify methods for Customers in the front-end, it allows for different permissions if a user has access to a bank (manager) or a ATM (customer). There are functions in Bank that will never be called by the customer and only by the messenger such as changing the date.
- Observer Pattern: This pattern allows for our program to not only have multiple windows open but also to never read stale data. By notifying all windows open to re-read from the database if one of the observers (windows) has done a change to the database, we don't need to do constant polling in the front-end to see if the database has changed.
- We considered using Strategy pattern for Accounts when it comes to withdrawing and depositing as each account has slightly different behavior (savings calls a check in security check to see if security account should be disabled or not). But we saw that we had to pass in too many attributes from the Account.java class such as all the different types of currencies, its balance, etc and when exiting out of a method such as SavingsBehavior.java's withdraw() function, we had to modify the value of Account.java's balance value, update its currencies and we found that simply overriding the Account.java's withdraw() and deposit() method if needed was the best way to move forward without making our design convoluted
- We also considered using Factory Pattern for generation of different account types but quickly found that it was also getting convoluted as unlike our previous assignment's ItemFactory that generated random items and would swap between factories such as PotionFactory, ArmorFactory which all inherited Item, the database would always return one instance of a Customer or a Transaction, which would not have no parent class from each other, and we quickly found that creating a pattern just added the exact same logic that our Database classes are currently doing but with extra boiler code with no benefits.

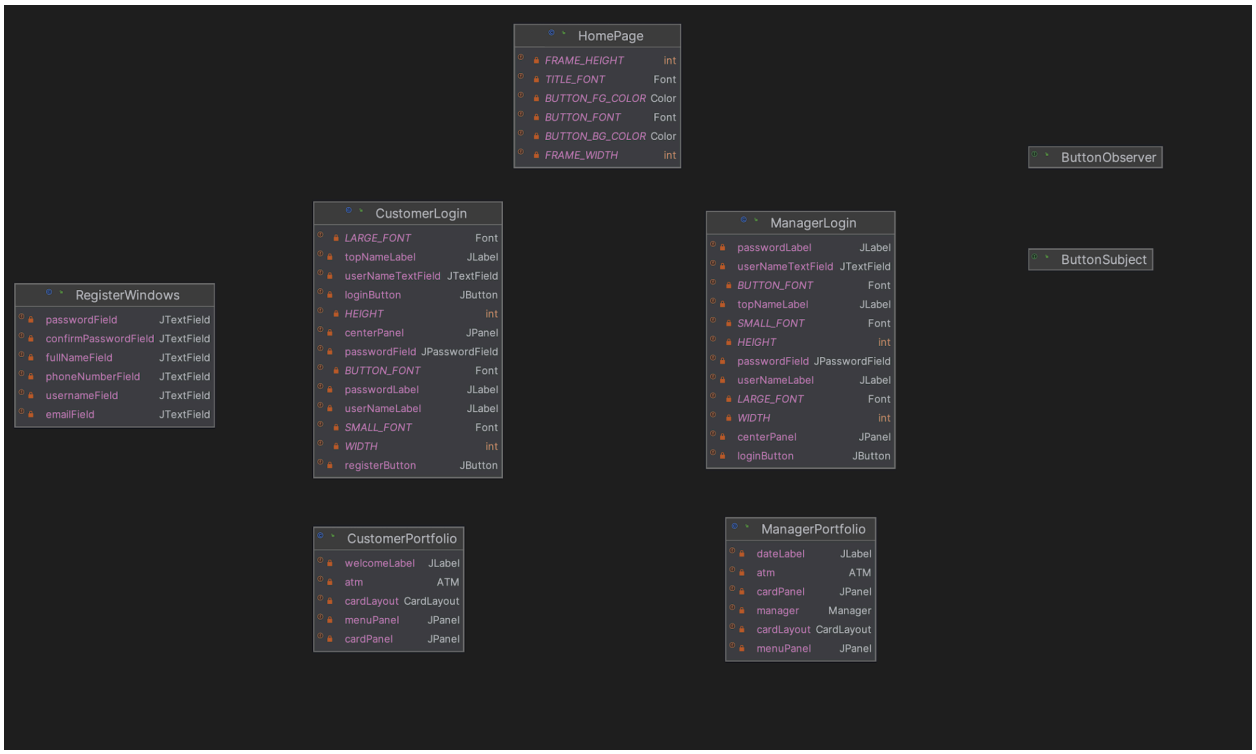
General Design Choices:

- We made an abstract class Account.java all do the most basics of a withdraw and deposits. (which is simply adding and removing money to an account). If any more checks are needed we can simply extend from Account.java and override the withdraw and deposit method while calling the super method. This prevents repetitive code and allows for unique checks while doing the most fundamental operations of an account.
- We approached our object design with the "X has Y", where we keep it consistent to where only banks can connect 2 customers together rather than making customers have

direct access to any other customer's account. As it wouldn't make sense for one customer to have another customer as an attribute.

- Rather than keeping multiple different values of stocks and also keeping track of unrealized profits, we simply do the math based off the data to display unrealized profits to prevent (which can be done with calculations based off the price customer bought a stock at and its current price). This prevents the need to recalculate and save into the database a customer's unrealized profits whenever a stock price changes
- Stocks also can be closed at any time, meaning that a Manager has a reference to all stocks and can notify them if a stock is removed, leading to all customers in the database selling their stocks rather than losing that money.
- All database classes only have static methods as its only goal is to write and read from a text file and doesn't require attributes to complete its job.
- All checks (expect for input type validation) are done on the backend rather than frontend to prevent repetitive checks in different front-end files (also better practice to do the checks in back-end)

Object and GUI relationship
Front-end UML



Database & Back-end UML