The Bank Project

by Vasily llin, Adam Streich, and Simon Rodriguez

Software stack: Originally our plan was to use a mongo database to store all of the bank's information. However, after running into difficulties with Mongo dependencies in Java we decided this was not going to work. We were not too familiar with Gradle or Maven projects so we thought that it would be better to switch to a less robust approach. We then switched to JSON files that store JSON objects for the data we need to store (which coincidentally is almost the same thing as MongoDB just with less searching and query capabilities), and upon login and closing the application the read and write files are called. The dependency that we used is called json-simple, a java library used for JSON related functionalities.

Process: To plan our design, we used a google document (attached at the end) that contained all of the components of the future Bank. We communicated with each other via Slack and we met consistently to demo the progress that we had made to each other and decide new plans of action after every meeting.

Framework: We chose to use a loose Model View Controller framework for this project as it seemed to make the most sense. A series of classes that dealt solely with functionality (deposit, create accounts, ect.) would be the controller so that the project could then interface (not in the java sense) with any view and controller. The GUI classes ended up serving as both the controller and the view. If we had more time and wanted to stick more closely to the framework one idea of how to split this up would be by jPanels. You could require each jFrame to have at least 2 jPanels where one panel functions as strictly updated from the model and the other panel handles all the inputs to interact with the model.

OOD Implementation: We have a standard object structure: a class for every major part of the bank. We were able to utilize the existing LocalDate and Currency classes in Java, so we did not have to make them ourselves. There are a few decisions that we would like to explain:

1) **Account behaviors are abstracted out.** This improves scalability, flexibility and extendability of the Account class. A new account class can easily be defined by giving it a set of the four behaviors: withdrawBehavior, depositBehavior, transferBehavior, and endOfMonthBehavior.
2) **A loan is a type of account.** A loan shares almost all properties of an account: name, balance (negative or zero), transactions (deposits and transfers to the loan), currency. It has the same deposit and transferTo behaviors (both simply increase the balance). The only difference is the way a loan is obtained (it needs to be requested first, and then approved), and the inability to withdraw our transfer to an account. Since we abstracted out the four behaviors of an account (withdraw, deposit, transfer, and end of month

behaviors), it is easy to simply give the loan the behaviors we want (CannotWithdrawBehavior, CanDepositBehavior, CannotTransferBehavior, EndOfMonthNegativeInterestBehavior). The way we implemented the requesting and approval/denial behaviors is by extending Loan with PendingLoan, which also holds a pointer to the savings account that the funds will be transferred to if the loan is approved.

3) **Manager, Customers, CurrrentDate and StockMarket are static attributes of Bank.** This is done so that there can be only one bank, with only one database of customers, etc. It also greatly simplifies GUI's because it gives the ability to use the static objects and not pass around users and stock market objects.
4) **We used JSON files to read and write state.** This works nicely with the OOD design of the bank. Every class has its own toJSON() and fromJSON(JSONObject jsonObject) functions. They can be combined like a Russian doll. For a good example of this, look at toJSON() in the Account class.

Below is the complete list of classes used in the Bank. We give brief comments about some of them, and indicate inheritance structure with tabs.

Core classes:

Main.java
Bank.java -- holds static pointers to manager, customers, current date and the stock market
Account.java
    CheckingAccount.java
    ManagerAccount.java
    SavingsAccount.java
    SecuritiesAccount.java
    Loan.java -- a loan is a type of account. It can be transferred and deposited to.
        PendingLoan.java -- pending loan. Becomes a loan if approved.
User.java -- user of the bank
    Customer.java
    Manager.java
StockMarket.java -- stock market has stocks and prices of stocks
Transaction.java -- holds date and amount
    TransactionBuyStock.java -- holds stock
    TransactionDeposit.java
    TransactionSellStock.java -- holds stock
    TransactionTransferIn.java
    TransactionTransferOut.java
    TransactionWithdrawal.java
JSONTools.java -- class with static methods to read from and write to JSON files. Each class has its own toJSON() and fromJSON() methods.

General classes:
Constants.java -- holds constants such as conversion rates between currencies, interest rates, etc.
General.java -- some general utility functions
IOTools.java -- some general utility functions related to I/O

Small helper classes:
Name.java - object for holding a user's first and last names.
Password.java - object to hold a user's password. Allows for specific conditional checks.
Permisions.java - a way to define what a user can do, say if you wanted different levels of customers, not really used fully, mostly for extendability.
Collateral.java -- collateral for a loan. Holds name and value of collateral
Credentials.java -- credentials for the user (name, username, password)
UName.java - object to hold a user's username. Allows for specific conditional checks.
Stock.java -- holds name and ID. Does not have price because prices are set by the stock market

Collections (each one is just a collection of the respective objects):
Accounts.java
Customers.java
Stocks.java
Transactions.java

Behaviors (each one is a behavior of an account):
DepositBehavior.java
    CanDepositBehavior.java
    CannotDepositBehavior.java
TransferBehavior.java
    CannotTransferBehavior.java
    UncheckedTransferBehavior.java
WithdrawBehavior.java
    CannotWithdrawBehavior.java
    CanWithdrawBehavior.java
EndOfMonthBehavior.java
    EndOfMonthInterestBehavior.java
    EndOfMonthNegativeInterestBehavior.java
    EndOfMonthNoBehavior.java

GUI:
General GUI:

BeginGUI.form - used to help create and modify the gui
BeginGUI.java - the “Launcher” for the app, creates a new Bank and manager if one doesnt exist, loads it if one does
GUI.form - used to help create and modify the gui
GUI.java - the GUI used to login or create a new customer

Manager GUI:
AddStockPanel.java -- panel for adding a new stock
EmitterPanel.java -- abstract class for a panel that emits text, and has listeners. Useful for the log of actions.
Listener.java -- listener (listens to EmitterPanel)
MainManagerFrame.java -- main frame
ManagerGUI.java -- entry point for manager GUI
ManagerToolbar.java -- toolbar panel
SetStockPricePanel.java -- panel to set stock prices
StockPanel.java -- panel holding AddStockPanel and SetStockPricePanel
TextPanel.java -- panel able to display text
TransactionPanel.java -- panel to get daily report
TimePanel.java -- panel to move time forward
CustomerPanel.java -- a panel to view the customer information

Customer GUI:
CustomerGUI.form - used to help create and modify the gui
CustomerGUI.java - go to Appendix A, lot to say here.
NewAccountGUI.form - used to help create and modify the gui
NewAccountGUI.java - presents an application used to created a new account
NewLoanGUI.form - used to help create and modify the gui
NewLoanGUI.java - simple GUI to request a new loan
TransferGUI.form - used to help create and modify the gui
TransferGUI.java - simple GUI to transfer money

Appendix A -- CustomerGUI, designed by Adam Streich

If starting this project over, I would change the entire way this has been implemented. As it stands right now, I recognise that it lacks a true OO design. This is due to me jumping in too quickly into building the GUI, before really doing research on how to best organize a GUI. The way that I would implement this if I were tasked with it again is to create a separate jPannel object for each tabbedPane. This would have made the whole project more organised and more easily expandable. Having everything for a frame in once class works in some instances (see NewAccountGUI or NewLoanGUI) because the tasks are small and singular, but once you try to

manage multiple accounts it quickly gets out of hand. Once each jPanel object would be created you could simply create the object, then add it to a new tab in the jtabbedPane. Another benefit of this would have been being able to take advantage of the GUI similarities of the Checking, Securities, and Savings account by creating a parent with the similar functionalities defined in it and just pass in the specific variables. However I believe you would still have to instantiate all the action listeners into their respective panel classes.

Originally I thought keeping everything in one class was the correct move, because to me it made the most sense to keep all the action listeners and the methods they call in the same class. I thought this way to keep the controller and the view aspect of the project from not bleeded into the model. This is because projects in the past I have done were much more simple and I did not fathom the scope of the project. Also in this class I definitely did not take advantage of the customer and stock market being static. I choose to pass the user around who was currently using the GUI, because it made the most sense to me and I also felt it worked better for some aspects of what I was trying to do.

Overall, I feel I learned a lot, both in GUI design and OO design while creating this class. I just wish I would have recognized all of this at the beginning of the design and implemented it rather than seeing my errors at the end.

Appendix B -- Class structure, designed by Vasily Ilin

The biggest challenge in designing the class structure was the initial step of understanding the functionality of each class. I had to do a good bit of refactoring (mostly with account behaviors) because I was lazy and did not think things through at the beginning. However, I noticed that it took me way less time to lay down the class structure for this project than for the QuestOfLegends. I believe this is exclusively due to what I learned in this class. The many hours of coding Tic-Tac-Toe and Quest paid off. If I was doing the design again I would only change a few minor details. For example, I would not have an object for each individual stock, and would store stocks as a Map<Stock, Integer>, Integer representing the number of stocks. It is interesting to note that I was not able to create an interface for buying and selling stocks, neither for the traders (customers and the stock market), nor for the objects traded (stocks), despite multiple attempts. I learned the valuable lesson that not everything needs to be perfectly Object-Oriented, and sometimes it is better to keep things simple.

Throughout the class structure design I kept the patterns we learned in class and used quite a few of them: the strategy pattern to abstract out the account behaviors, the listener pattern in GUI and the singleton pattern (somewhat) in designing the Bank class.

Appendix C -- Manager GUI, designed by Vasily Ilin

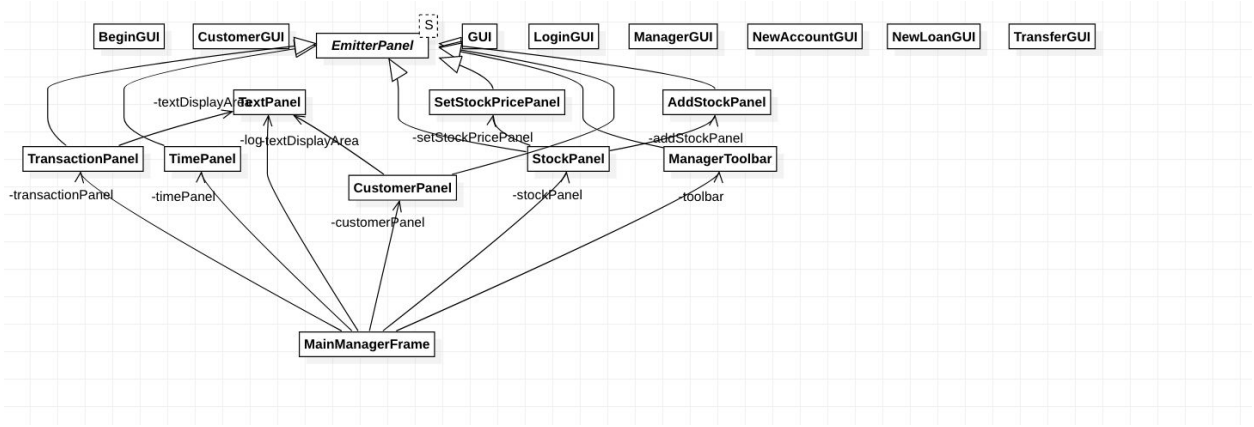
I had absolutely no prior experience with GUI a week ago. All my knowledge of Java Swing came from a 5-hour tutorial on YouTube I watched. It helped me design GUI in an Object-Oriented manner, which I am very happy about. I did not use any auto-generated code

that's impossible to read and maintain, so the ManagerGUI is extremely scalable and flexible. I spent many hours learning OOD and how OOD works in the GUI setting, and it paid off immensely. The UML diagram of the GUI package clearly shows the fruits of this labor.

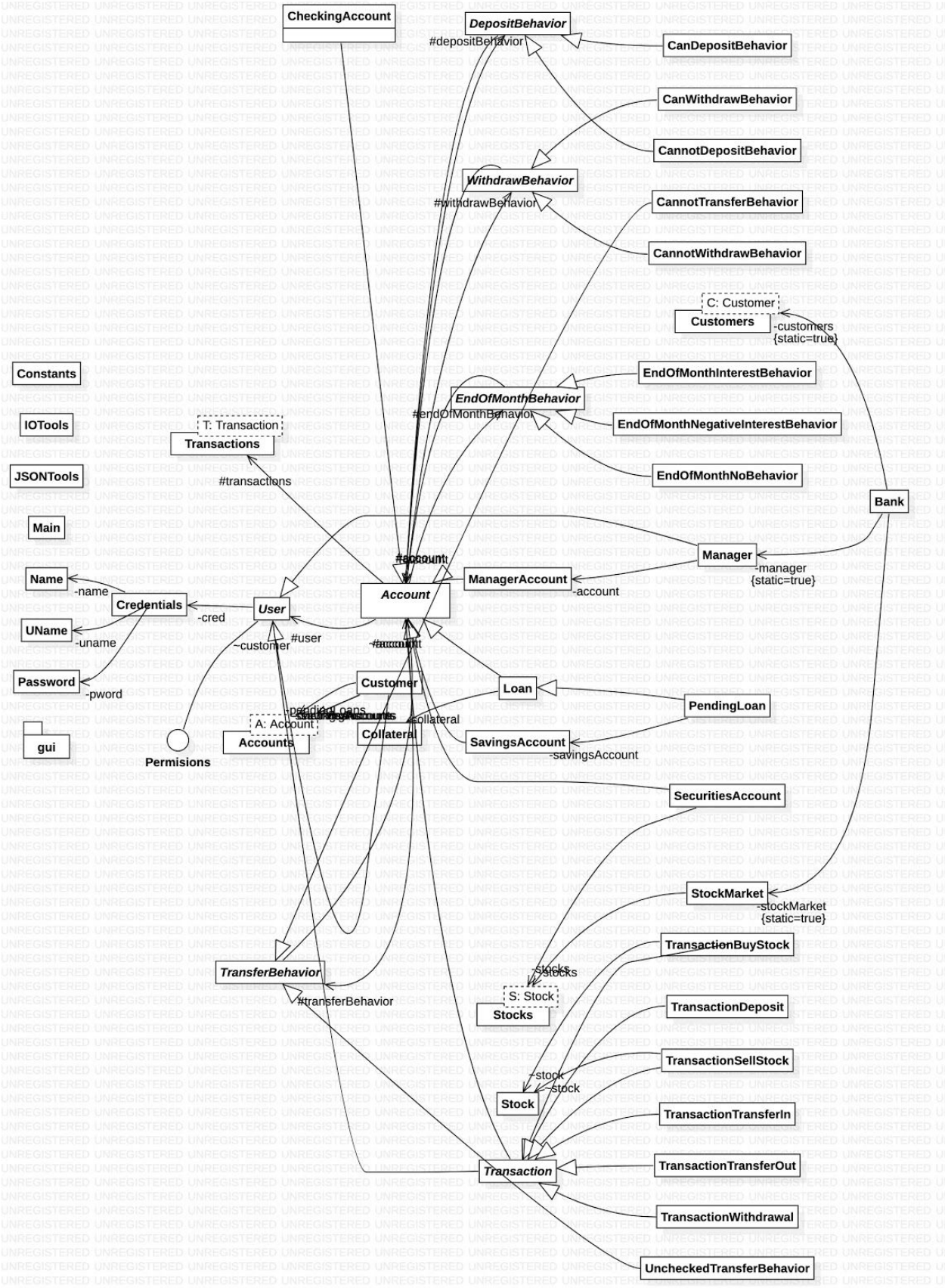
Appendix D -- JSON read and write, designed by Vasily Ilin

I had no experience with I/O in Java, except for reading and parsing files in the Quest assignment. My teammates wanted to use fancy tools such as MongoDB, which I was quite opposed to, given that this is a OOD class, and not a database class. I ended up right. The JSON-simple package provides excellent functionality that works extremely well with the spirit of OOD. Every class has its own toJSON() and fromJSON() functions. They can be combined like a Russian doll. This synergy between JSON and OOD framework allowed me to write code extremely fast, with few bugs (there are some issues, such as the fact that JSON-simple parses ints as longs, and they need to be converted back). I am happy that I learned how to do I/O in the Object-Oriented way.

Below is a UML diagram of the GUI package:



Below is UML diagram of the bank package:


CheckingAccount
DepositBehavior
#depositBehavior
CanDepositBehavior
CanWithdrawBehavior
CannotDepositBehavior
WithdrawBehavior
#withdrawBehavior
CannotTransferBehavior
CannotWithdrawBehavior
C: Customer
Customers
-customers
{static=true}
Constants
IOTools
JSONTools
Main
T: Transaction
Transactions
#transactions
EndOfMonthInterestBehavior
EndOfMonthBehavior
#endOfMonthBehavior
EndOfMonthNegativeInterestBehavior
EndOfMonthNoBehavior
Bank
Manager
-manager
{static=true}
Name
-name
Credentials
-cred
User
UName
-uname
Password
-pword
~customer
#user
Account
ManagerAccount
-account
Customer
Loan
PendingLoan
SavingsAccount
-savingsAccount
gui
Permisions
A: Account
Accounts
Collateral
-collateral
SecuritiesAccount
StockMarket
-stockMarket
{static=true}
TransactionBuyStock
TransferBehavior
#transferBehavior
-stocks
-stocks
S: Stock
Stocks
TransactionDeposit
TransactionSellStock
-stock
-stock
Stock
TransactionTransferIn
Transaction
TransactionTransferOut
TransactionWithdrawal
UncheckedTransferBehavior

Below is an attempt of a UML document that we made at the beginning:

I want to be a Bank Manager!!
It has always been my dream to wear a blue pinstripe suit and be a bank manager!!! And with your help, you will make my dream come true!! Your assignment is to build me a bank that I can manage. I want to be Bank Manager!!

There might be a bonus for displaying a picture of a bank manager in a blue pinstripe suit

Your assignment, should you choose to accept it, is to build me a bank that I can manage. My bank will offer my customers the ability to create checking and savings accounts, maintain deposits in at least three different currencies, and take out loans (if they have collateral). I want my bank to be highly service oriented, just as long as I never have to deal with my customers in person. My bank is going to be a true on-line bank. My customers should be able to do everything they need to do from the fancy Bank ATM that you will create for me. Through the ATM, my customers should be able to create any type of account they want – savings or checking, request a loan, and view their transactions and current balances. The bank manager off course (that's me!) should have the ability to check-up on a specific customer, all my customers – especially my poor ones who owe my money! I should also be able to get a daily report on transactions for that day. My goal for providing this service is to make money! I will charge a fee every time an account is opened or closed and a fee every time a checking account transaction or any withdrawal is made. I will only pay interest on savings accounts that have high balances, but I will charge interest on all loans! I do not care about having happy customers, I want to make money! I want my bank to be so popular and to provide as many services to my customer as possible. One of those services is the ability to play in the Stock market. But this service is only reserved for my very rich customers. Any customer who has more than $5000.00 in their savings account, can choose to transfer any amount over $1000.00 into a new securities account that they can use to trade stocks. But the customer must maintain a $2500.00 balance in their savings account. From within their securities account, a customer can enter trades (buy or sell stocks), see their current open positions, and both their realized and unrealized profit. The bank manager is responsible for maintaining the list of stocks that his customers are able to trade and is also responsible for updating the current price of each stock.

Idea for Manager GUI

Roledex(customer list)9	Running updates on all interaction (nonstock - checking, saving, loans, etc)	Stock tab (will have list of customers who have securities accounts, running tab of stock transactions)	High priority flags (loans, people who are behind in payments, etc)

Blue comments are Adam's.

Red comments are Vasily's.

Your assignment, should you choose to accept it, is to build me a bank that I can manage.

Probably want to have a manger class and then customer class

My bank will offer my customers the ability to create checking and savings accounts, maintain deposits in at least three different currencies, and take out loans (if they have collateral). I want my bank to be highly service oriented, just as long as I never have to deal with my customers in person.

Checking class, savings class, both inherit from an account class that can handle deposits and withdrawls in the 3 currencies. Also a loan controller class that can evaluate and issue loans.

Currency class with attributes: name, symbol (for displaying). Each account has a Currency attribute.

My bank is going to be a true on-line bank. My customers should be able to do everything they need to do from the fancy Bank ATM that you will create for me.

GUI

Through the ATM, my customers should be able to create any type of account they want – savings or checking, request a loan, and view their transactions and current balances.

Should have a customer class that holds references to the accounts created

Transaction class with a time stamp. Each account should have its own set of transactions.

The bank manager off course (that's me!) should have the ability to check-up on a specific customer, all my customers – especially my poor ones who owe my money! I should also be

able to get a daily report on transactions for that day.

This is interesting, we could save all transactions in a stack, but that seems wasteful and putting data in a place it shouldn't be, so then we have to organize the data (withdrawls) by date and have a function to get the days transactions from the database. Same goes for searching through the costumers for ones who owe money.

When displaying transactions for a specific account, filter by date and call toString on each transaction. When displaying transactions for a specific client, display transactions from all accounts. To get a daily report, display transactions for all users for that day.

My goal for providing this service is to make money! I will charge a fee every time an account is opened or closed and a fee every time a checking account transaction or any withdrawal is made. I will only pay interest on savings accounts that have high balances, but I will charge interest on all loans! I do not care about having happy customers, I want to make money!

Just add fees to things.

I want my bank to be so popular and to provide as many services to my customer as possible. One of those services is the ability to play in the Stock market. But this service is only reserved for my very rich customers. Any customer who has more than $5000.00 in their savings account, can choose to *transfer any amount over* $1000.00 into a new securities account that they can use to trade stocks. But the customer must maintain a $2500.00 balance in their savings account. From within their securities account, a customer can enter trades (buy or sell stocks), see their current open positions, and both their *realized* and *unrealized* profit.

Going to need a stock class, maybe an enum for the types of stocks (might get tedious), also a securities account class, look into the idea that you can only have one of these. And a way for the manager to create, update, and remove stocks. Probably want to make a market class that acts as the stock market that the manager sets up.

Stock class, with attributes: price, name. Stock market contains stocks (similar to the market in the Quest game), implements Collection<E extends Stock>

The bank manager is responsible for maintaining the list of stocks that his customers are able to trade and is also responsible for updating the current price of each stock.

Interfaces:
A customer class must have the following functions:

- Create checking account
- Create savings account

- Create securities account
- Close an account (where does the money go?)
- Request loan
- Deposit cash to an account
- Withdraw money from a checking account
- View transactions for an account (for a time period)
- View balance in an account
- Buy stock
- Sell stock
- Play on the stock market (repeatedly buy and sell stocks)
- See realized profit from the stock market
- See unrealized profit from the stock market

A manager class must have the following functions:

- View transactions of a specific customer (for a time period)
- View transactions of all customers (for a time period)
- View balances of a specific customer
- View balances of all customers
- Add a stock to the stock market
- Change the price on a stock in the stock market
- Move time forward

An account class must have the following functions:

- Open
- Close
- View transactions (for a time period)
- View balance

Stock market must have the following functions:

- Buy stock from a customer
- Sell a stock to the customer
- Create a stock
- Change price of a stock

Bank must have the following functions:

- Create a new customer (need name, give it a unique ID)
- Login
- Move time forward
- Load state from files
- Save state to files
- Load user from file
- Save user to file - leaves the option open to save user to file, after every interaction

OOD ideas:
Factory for accounts
Stock Market implements Collection<Stock>

OTHERS?

Things to do:

- Set up our database/memory system
- Think of class structure/ design a frame for how we want everything to be organized
- Look at GUI related stuff
- Find a useful pattern/patterns

Login
Class List:
Bank
User
    Manager
    Customer
Account: Transactions (implements Collection<Transaction>)
    Savings
    Checking
    Securities
Transaction
Stock
Stockmarket - simon

Java Interfaces:
Buyable (for the stocks)
Sellable (for the stocks)
Withdrawlable (for the accounts)
Depositable (for the accounts)

1. Functions
2. GUI
3. MONGO

1. Ask user to sign in, or create account if they don't have one
    a. Hard code manager account in mongo
    b. If user signs in as manager it will open a manager UI
2. Manager UI:
    a. Ask manager which bank user they would like to access
        i. Text box with username input

ii. Have some sort of username, let a user select a username that isn't already created
iii. Display a general list of the users to the manager, allow for some sort of filter by debt
    1. View balances(name+other info): if they select view balances, open some screen that displays the data
    2. Transaction date range: Enter a date: after 'submit' all of the transactions between the current date and the date entered pop up
    3. Transaction by day: a button that displays all of the current days transactions