

Final Project Report

Vivian Gunawan, Qingyang Shi, Yunshuang Tao

Our code can be found [Fancy Bank on Github](#)

[Introduction](#)

[Implementation](#)

[Technology Stack](#)

[Abstract Design](#)

[Model Design](#)

[Database Design](#)

[Data Access Objects](#)

[View Design](#)

[Customer View](#)

[Schematic UML](#)

[Login and Registration](#)

[UML](#)

[User Interface](#)

[DashBoard](#)

[UML](#)

[User Interface](#)

[Stock Market Functionalities](#)

[UML](#)

[User Interface](#)

[Manager View](#)

[Schematic UML](#)

[Login and Registration](#)

[UML](#)

[User Interface](#)

[Dashboard](#)

[User Interface](#)

[Discussion](#)

[Conclusion](#)

Introduction

This report serves to clarify our implementation and discuss the design choices made in regard to the assigned final project for CS 591 P1 Spring 2020. Despite not delivering a complete project, there are various implementations we believe we did well in and others with room for improvement.

The project requirements consist of creating a functional Java desktop application that represents an online bank. The application shall provide both customers and managers of the bank multiple functionalities.

The bank customer should be able to create multiple bank accounts, such as checking and saving accounts. To store multiple currencies, multiple accounts of the same type are permitted, as each account can only deal with one currency. However, a fee is with every creation or deletion of the account. Money within the accounts can be transferred between different customers and also different accounts they own. Customers can also apply for loans if they have collateral. The application should also allow customers to view the state of their accounts such as their balances and transaction history. Customers with a deposit of more than \$5000 in their savings account, can optionally interact with the stock market through opening a security account.

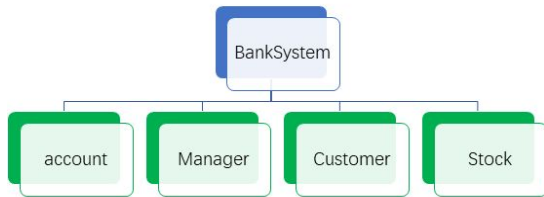
The bank manager should have the ability to check information of all the customers, including their personal information, their accounts' information. The manager also has the power to approve loans, charge and pay interest and fees. The bank manager also has full control of the stock market. In addition to these functionalities, the application should provide the bank manager with daily transaction reports, and how much money the bank is earning.

Implementation

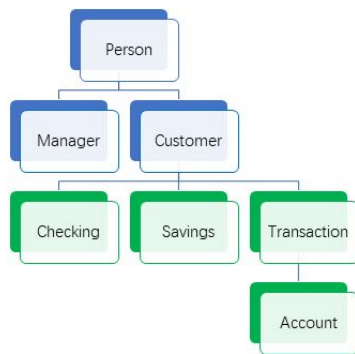
Technology Stack

The functionality of the bank system is complex. Thus, a well-organized structure is needed. In our program, we used Java Swing for our frontend, PostgreSQL for our database, and Java for our backend.

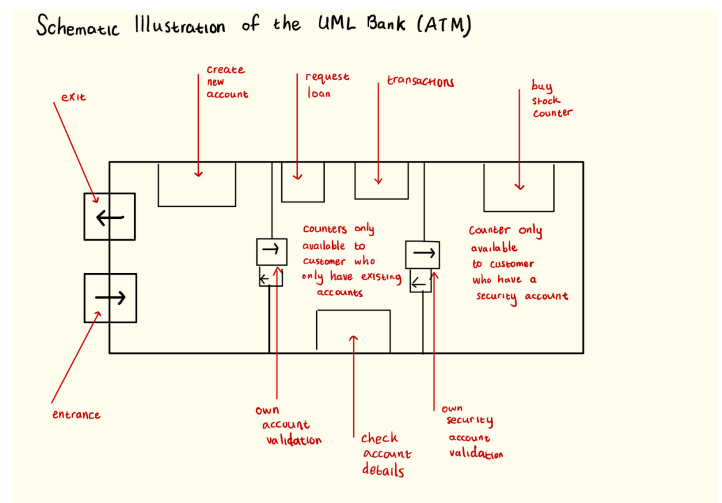
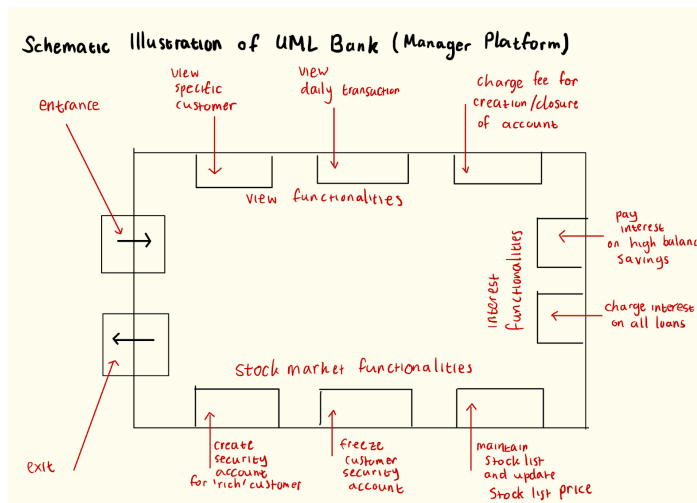
Abstract Design



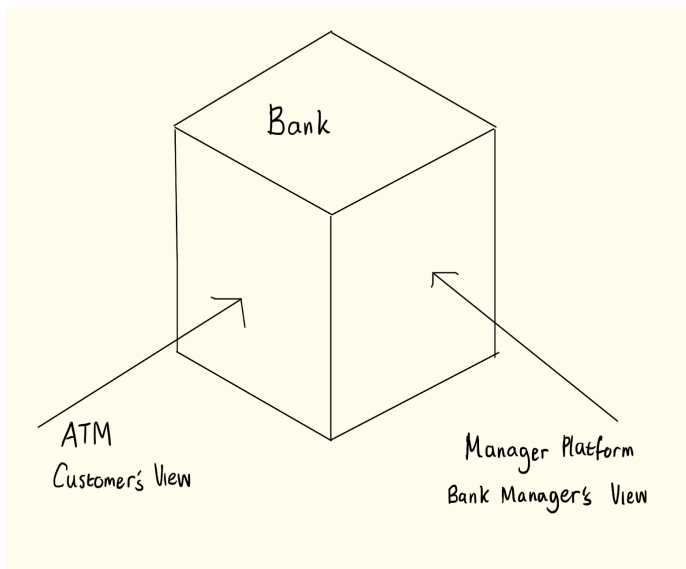
We referred to this abstraction when designing the BankSystem. A Bank System consists of bank accounts, manages, customers, and stocks.



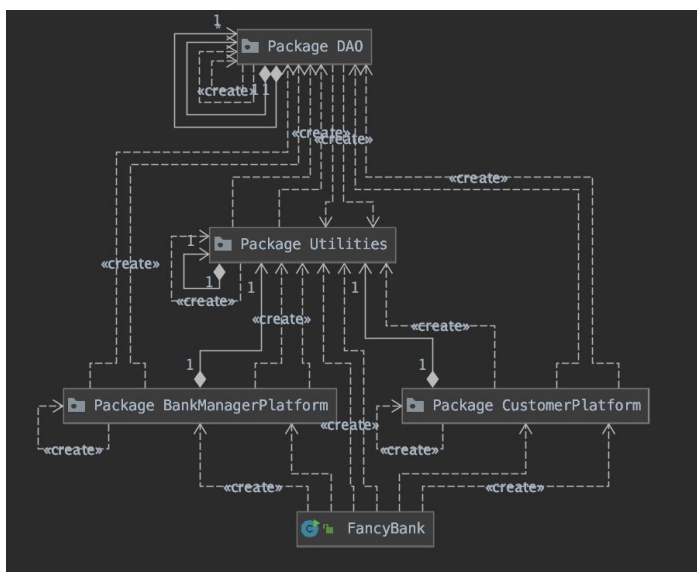
In our abstraction, we also keep in mind that managers and customers are people. Accounts must also be owned by customers.



Model Design



The image is a general representation of how managers and customers have different views of the bank system. Meaning customers and managers only interact by updating the state of the model represented by the box labeled Bank.

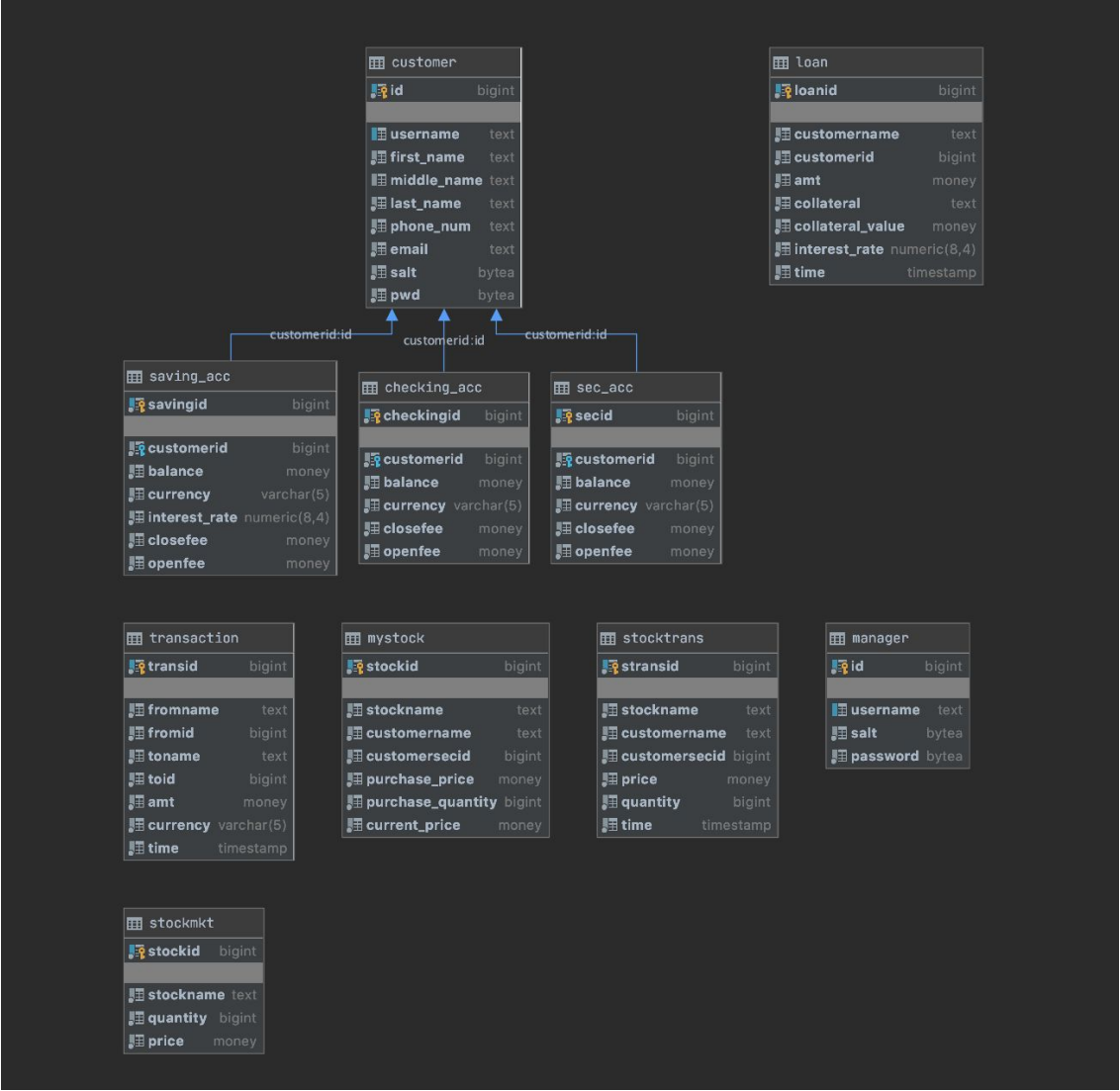


This UML diagram shows how we implemented the general representation. There is no direct interaction between the BankManager Platform and Customer Platform. Instead, data access objects (DAO) are created by both platforms and used for updating the state of the model.

Database Design

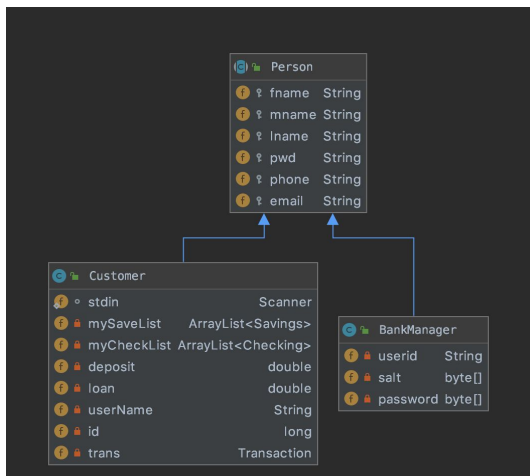
The database stores the states of the bank. This includes customers, managers, accounts, and stock information.

Entity Diagram



Data Access Objects

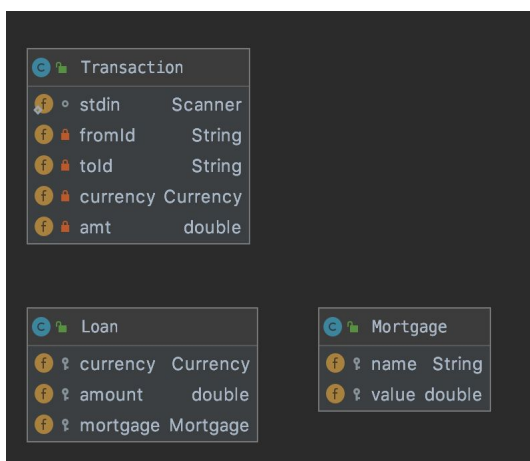
Data access objects are used by functions to implement a proxy pattern, this helps in translating client requests into SQL statements.



The Person class represents a person in the real world and their attributes.

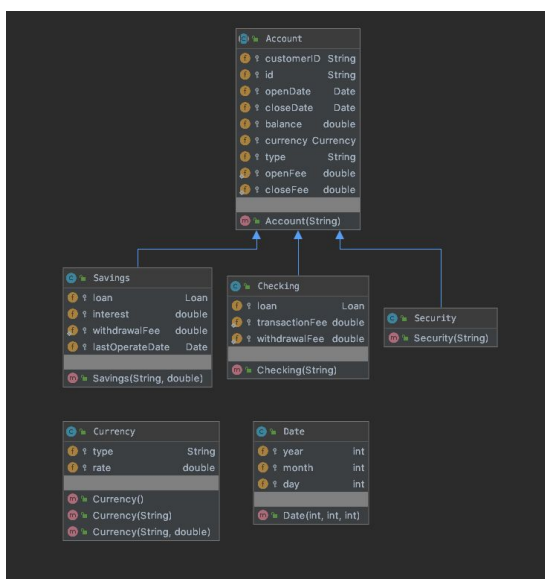
Customer and BankManager both extend from Person, inheriting its attributes.

Customer class has extra attributes such as loan, deposit, and transaction so that each time they want any of the objects, they can create a new instance of it.



Transaction, Loan, and Mortgage classes do not have inheritance relationships with each other, but they are actually similar types.

All of them are forms of money flow, whether between different accounts or between accounts and the bank.



The three types of accounts, Savings, Checking, and Security, are inherited from an Account abstract class. The Account class contains basic attributes and methods of bank accounts.

Checking upon a transaction from a checking account, a fee is charged.

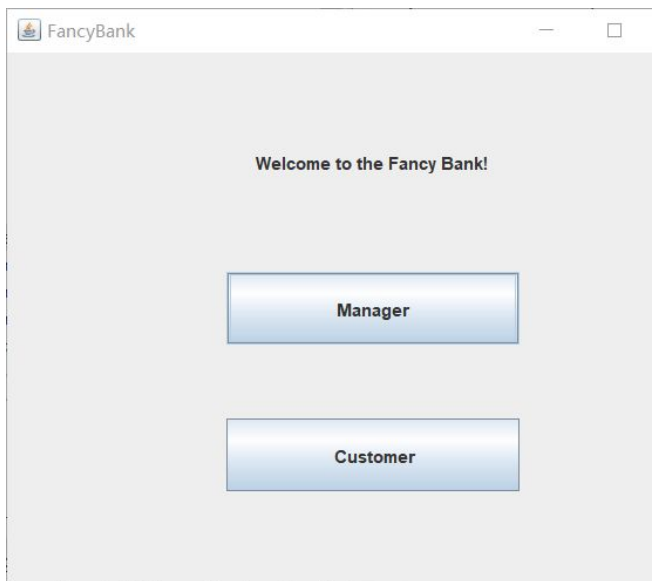
Savings: an interest placed on this account based on the date it is last used (deposit..etc)

Security: Left unimplemented. Usage is for interacting with the stock market.

Here we also created our own Currency and Date objects.

View Design

Main View



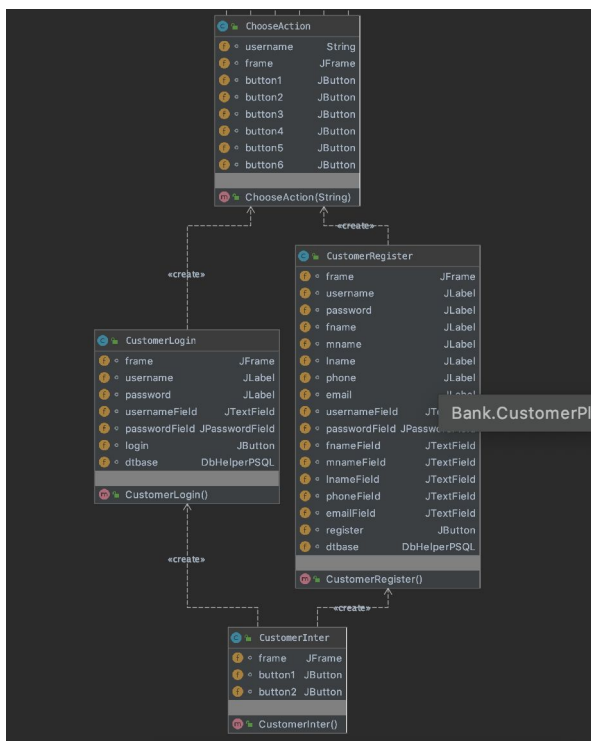
When running the application, this is the first view presented to the user.

Depending on what the user selects, the user is redirected to the appropriate login platform.

Customer View

Login and Registration

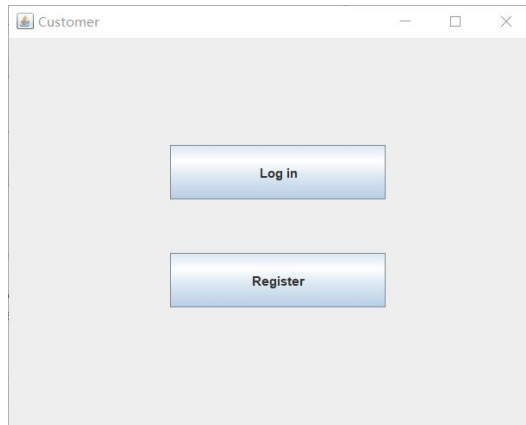
UML



CustomerInter represents the customer platform GUI. It welcomes the user and redirects them to CustomerLogin or CustomerRegister appropriately.

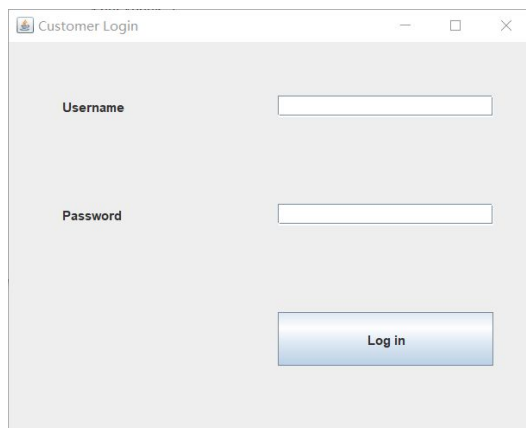
CustomerLogin and CustomerRegister are also GUIs within the customer platform. They will eventually redirect the user to the dashboard upon authentication.

User Interface



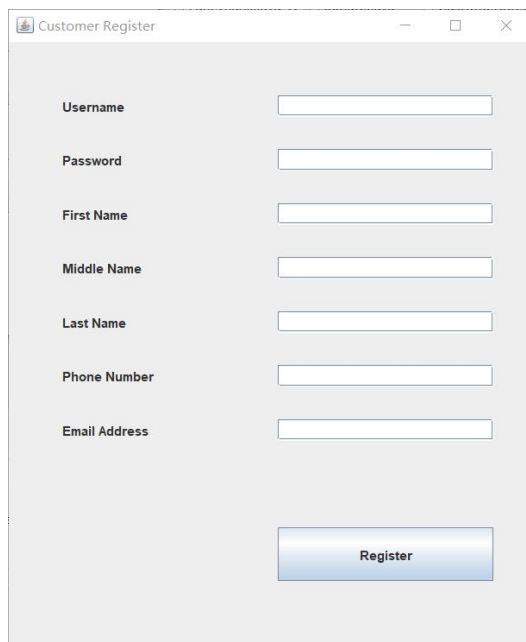
A window titled "Customer" with a light gray background. It contains two blue buttons with white text: "Log in" and "Register". The buttons are stacked vertically in the center of the window.

If the user is a customer, he should choose whether to log in or to register.



A window titled "Customer Login" with a light gray background. It contains two text input fields: "Username" and "Password". Below the "Password" field is a blue button with white text labeled "Log in".

If the user chooses to log in, he should enter the right username and password. Then the program will search the database to see if this customer exists. If so, check the password. If it is right, the user should be able to log in.



A window titled "Customer Register" with a light gray background. It contains seven text input fields: "Username", "Password", "First Name", "Middle Name", "Last Name", "Phone Number", and "Email Address". Below the "Email Address" field is a blue button with white text labeled "Register".

If the user is a new customer, he should fill the registration form with the appropriate data.

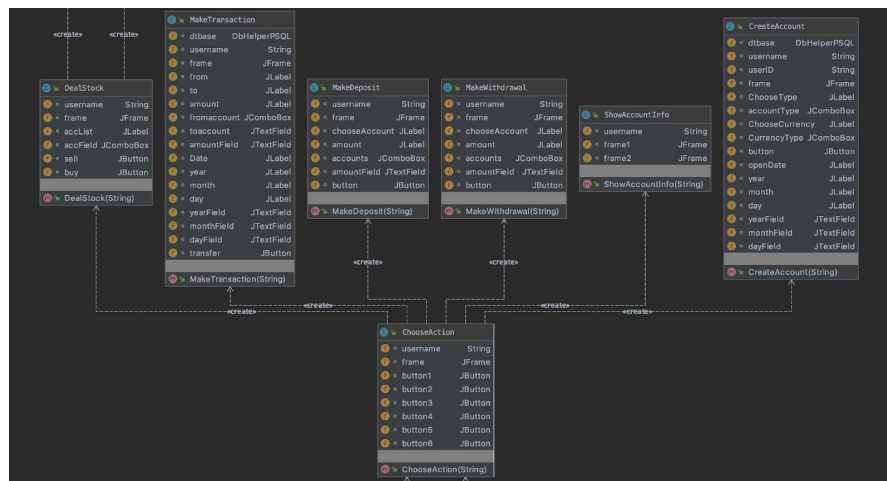
Upon successful registration, the user will be redirected to the dashboard.

If registration fails, a pop-up window indicating so will appear.

Unsuccessful registration may occur if the username is already taken.

DashBoard

UML



ChooseAction represents a dashboard for a customer to perform a set of operations. Different buttons lead to different action interfaces.

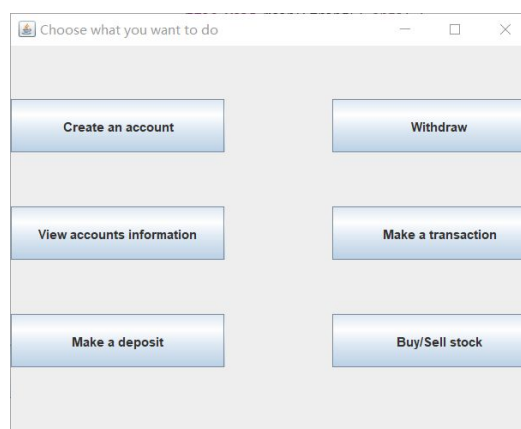
CreateAccount is a platform for a customer to create a new bank account. He can select the type of account he wants-checking, savings or security. If he wants to create a security account, the program will search the database to see if he has a deposit of more than \$5000 in one of his savings accounts.

ShowAccountInfo is a platform for a customer to see information about his accounts in a table form. His savings and checking accounts are shown on one table and security accounts on another table.

MakeDeposit and MakeWithdrawal are platforms that allow a customer to deposit money in or withdraw money from one of his accounts. MakeTransaction allows a customer to transfer money to another account.

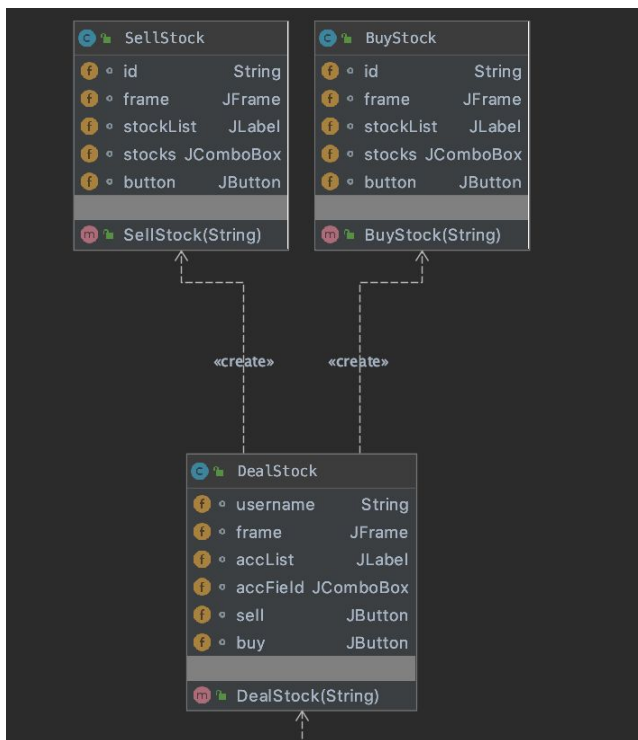
DealStock represents a platform in which a user can choose one of his security accounts(if he has) to sell or buy stocks.

User Interface



In this interface, a customer can choose to perform a set of operations. They can create a new account, view accounts' information and do other things.

Stock Market Functionalities

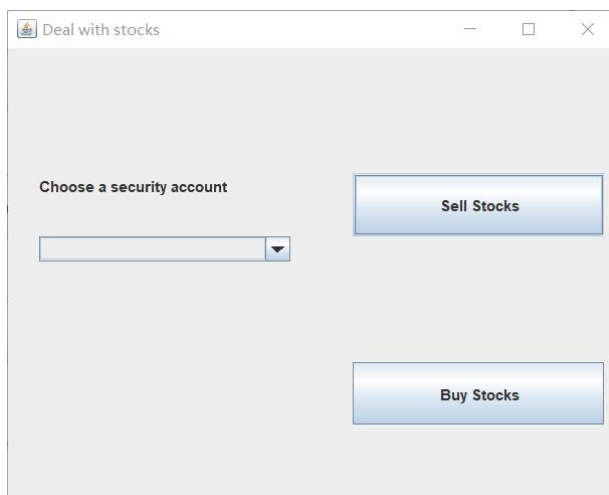


When a customer chooses a security account and wants to sell stocks, he goes to the **SellStock** platform and he can select which stock he wants to sell.

If he wants to buy stocks, he will go to **BuyStock** platform and choose which stock he wants to buy. If he has enough balance in this account, he can buy stocks successfully.

UML

User Interface

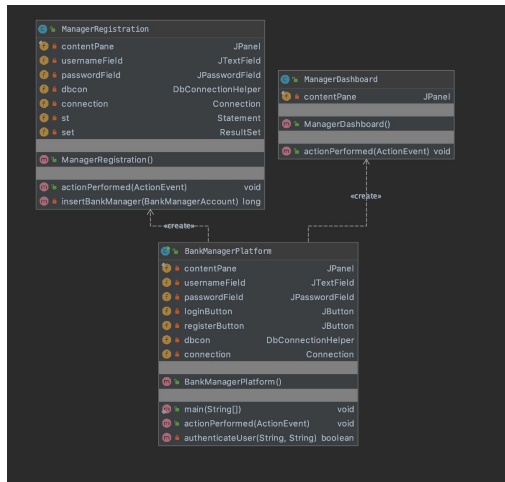


If the customer wants to deal with stocks, he should first choose a security account that belongs to him. and then sell or buy stocks.

Manager View

Login and Registration

UML

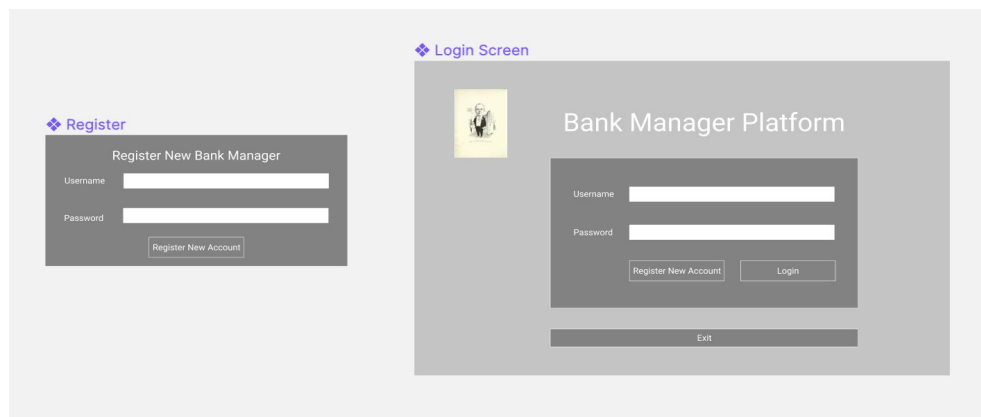


The BankManagerPlatform is the main GUI.

Upon the user entering the BankManager platform the user is prompted to log in. If the user doesn't have an account he may register on the pop-up window. A pop-up message will show if registration is successful or not. Usernames must be unique to register.

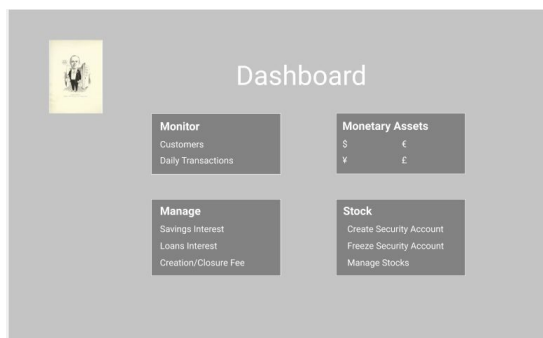
Upon successful authentication during login, the user is redirected to the dashboard.

User Interface



Dashboard

The dashboard is left unimplemented in the Bank Manager Platform. Below is the UI design.



Discussion

Due to time constraints, our group was not able to deliver a completed project, there are multiple functionalities within the project requirements that are left unimplemented. In addition to this, we also realize the flaws in our current implementation that we weren't able to fix in time.

We wanted to implement the model view controller design pattern, in which a controller coupled with the proxy design pattern would update the state of the model. However as seen in our implementation, there is no controller implemented. In our implementation, the user directly creates the object and implements the proxy pattern instead. Even this is not uniformly implemented as different group members work on different parts of the project.

For example, when the customer at the front end choose to create a new account and need insert the new account's information into the database, it has to create a new database-helper instance and is designed to call the insert-new-account method of the instance to do the updates in the database, which is inefficient. The correct way should be the information from the front-end directly put into the database with PostgreSQL command, and there should be no need to create a new object and call a method out of the current class. This does not happen because the customer front-end and the database-helper are implemented at the same time by different group members, and the reason why put the insert-account method in database-helper is that the database member thinks the database-helper function does not require the front-end member to manipulate the database directly.

We planned to implement a facade pattern within the dashboard of each platform to unify all the functionalities. A bit of this design pattern can be seen in the implementation of the customer platform. However, once again is left incomplete due to time constraints. The dashboard in the bank manager platform is left completely unfinished.

Although security accounts are not implemented, we believe that a null object pattern could be suitable.

We planned to implement an observer pattern to connect the bank manager and the stock market so that the bank manager can be the only one that can control and update the stock market. But in the real implementation process, the bank manager class and the stock class and the table in the database that stores the stock market information are designed by different members, and after we finish our own parts, we found it hard to modify the observer pattern.

We have completed most parts of the database. For the customer platform, it is almost completed. However, due to time constraints, we were not able to connect those functions with the database. Also, the database, the customer platform, and the manager platform are designed by different members. Since we all have our own habit of design, we found it is hard to combine our parts because we cannot see each other's codes on time.

Conclusion

Through this project, we have learned to use Java Swing and PostgreSQL coupled with the object-oriented design patterns taught in class. Despite not being able to complete the project, we have learned to successfully collaborate and communicate as a team. We exchanged ideas regarding implementations and design and managed to commonly agree on one overarching implementation discussed in this report. We've also distributed the task among our members fairly, we do not blame anyone specifically on the lack of completion of the project.