CS591 P1 - Team 10

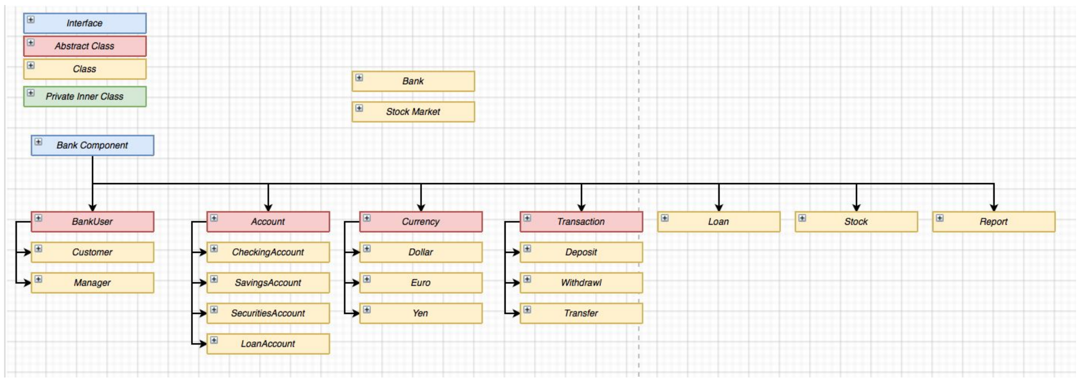
My Fancy Bank - Final Project Design Document

Jorge Jimenez
Ziyu Shen
Priya Kumari

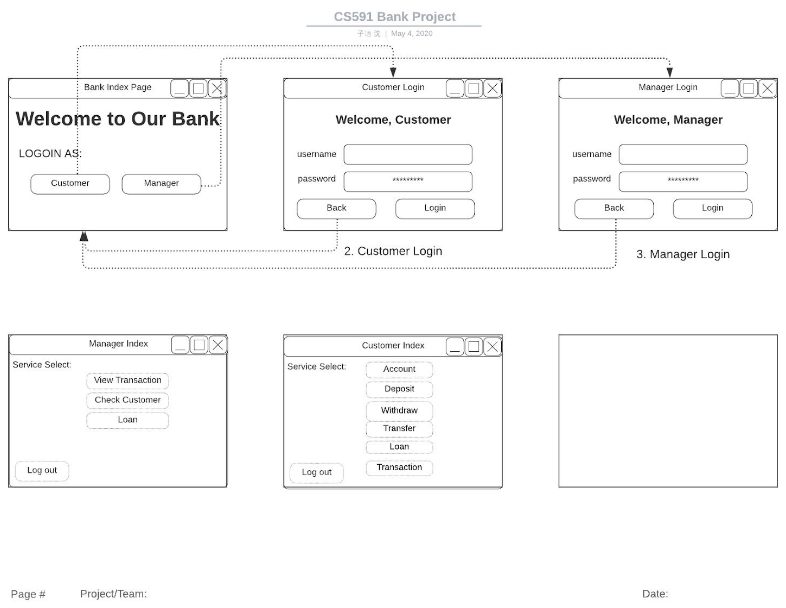
Brainstorming and Planning

As soon as we received our team assignments, we set up a **Github** repository for our project, a **Google Drive** for important documents and a **Slack** channel to be able to communicate through the design and implementation process. For brainstorming and planning, we each individually read through the PDF and through a **Zoom** conference call, we began talking about the class structure that we wanted to implement. As learned in class, we decided to think about a real-life bank: all the different components that make a bank possible, like the different transactions that happen on a daily basis, customers walking in and out of the bank. By comparing it to the different mechanisms and functions of a real-life bank, we identified most of the objects that we would need for the class structure.

For the design process, we used two technology stacks to facilitate our thought process and help us guide through the implementation phase: **Draw.io** (recommended by Jorge) and **Lucidchart** (recommended by Ziyu). Both of these were essential. Draw.io helped us visualize our class structure before even having a single line code, allowing us to see the levels of abstractions that we wanted to reach with each class and the different relationships between all the different components. Lucidchart allowed us to further visualize the GUI, which was important since it would be tying all the different components together like glue.



XML diagram of our class structure during the design phase



Lucidchart Design for Our GUI

Before even writing a single line of code, we already had a working class structure and we even had identified some properties that would later shape our classes. We noticed that we will need objects to represent every single type of currency that we want present on our bank, abstract class for the account but have the different accounts instantiated on their different objects, the different users, stocks, the stock market, loans, and much more!

Design Patterns

One of the last topics we studied in the course were the different design patterns, and this became a very important topic during the design phase. We wanted to include as much as these design patterns in our class, and we began brainstorming how to include it. We identified two important design patterns: **the Factory Pattern and the Singleton pattern**. We noticed two things: there is always only one bank and there are always accounts created and closed on a day to day basis, but there are different types of accounts (Savings, Checkings, Loan, Securities, etc.). We easily tied the two together and decided that we would implement the Singleton Pattern with the Bank object, and the Factory Pattern by creating an AccountFactory. The Singleton Pattern would be used by making the private constructor of the Bank class and creating a getInstance() method to ensure that there was only one Bank object created. The AccountFactory class would have the appropriate method to create different account types.

Interfaces

For interfaces, we decided to implement four major interfaces to enforce certain restrictions on certain Bank classes that needed to be implemented and identify relationships between certain objects.

BankComponent.java: This is the interface that is implemented by any of the physical components of the Bank. We noticed that in a mobile bank, every “element” is associated with a unique ID. Transactions, accounts, customer accounts and even digital currency all have these. For this reason, we created the BankComponent interface, which enforces any component from the bank and basically creates a contract between the Bank object. This also gives a lot of room for scalability as any element in the future that would be tied physically to the bank would need this interface.

Exchangeable.java: We decided to implement the Currency exchange for multiple reasons. First of all, any physical element that directly contains a currency attribute should be convertible to another currency. Through this interface, it also enforces elements that contain a value (Currency object, Transaction object and Account object) should be able to have their “value” converted to a different value.

Tradeable.java: Similar to how Items could be sold in the market in the Quest, we quickly identified that Stocks were essentially the same thing. We also knew that in real life, there are other things that are traded, like bonds. We implemented the Tradeable interface for this exact reason, to enforce restrictions on elements being traded on a market.

InterestTaxable.java: This is an interface to enforce restrictions and specify implementations on elements that can be applied as an interest rate tax. This is implemented on both the Loan and the SavingsAccount objects, and can be used for scalability in the future (for example, if you wanted to add a CarLoan or a Mortgage object in the future).

Abstraction

Abstraction was a key component for this design structure to work. Right of the bat, we identified multiple components that all shared commonalities between each other, and thought about a way where we could abstract all these data and methods that they shared in common on an abstract parent class and keep the important methods and attributes that the Manager would need on an inherited class. It specified the properties their child classes had to implement and the way they needed to implement these. Our abstract classes demonstrate the the scalability and reusability in our code: with a single abstract class, we gave creation to multiple components that represent the physical components of the bank. Keeping this abstracted also allowed us for proper encapsulation as well.

Account.java: Abstract class for the different account objects (Checking, Savings, Securities, etc.). This implements both the BankComponent and Exchangeable interfaces, and the majority of the functionality of every type of account, for example, a unique id, an array of transactions associated with the account, the current amount, opening fee and closing fee. This abstract class allows scalability for more types of account and re-usability between the accounts as well.

BankUser.java: Abstract class representing a user of the bank (customer and manager). Implements the BankComponent interface. Both a Manager and a Customer are a user in the bank, so we abstracted their commonalities into a parent class for storing and accessing their username and password. This also allows for scalability in the future (if we wanted to implement a Worker object which represented someone who works for the Bank but does not have the same level of clearance as the Manager.

Currency.java: Abstract class representing both the physical and the monetary representation of a currency. This is extends into the dollar, euro and yen objects, and allows room for scalability and reusability to make thousands of other currencies in the future (pounds, Canadian dollar, dinar, etc.). We would agree that this was one of the most important classes in our design structure, as currency played a vital role throughout the bank and mostly every other class that implements BankComponent included a Currency object somewhere. A Currency object would specify both the value of the object and the physical representation of the object. Even though it was a harder task to code since both had to be tied together, it was the proper approach to our class design structure as it represented better when a virtual dollar would be, from an OOP perspective.

Transaction.java: Abstract class for a Transaction (Withdrawal, Deposit and Transfer). Essentially a transaction between an account and a user or an account and another account where the exact same thing at the end of the day: an inflow or outflow of money with a specific unique ID and a date. This allowed room for a lot of code reusability: we could store all the different transactions in a single array instead of having separate arrays for Withdrawals, Deposits and Transfers, for example. All the actual methods and data attributes of a transaction are stored in this class, which makes this essential for scalability if we wanted to add different types of transactions in the future.

Users and Accounts

At the end of the day, both the Manager and the Customers are all simply users of the bank, what differences themselves are the functionality and access levels that both of them have. For this reason, both of them have the parent class BankUser.java, which is abstract as this is never instantiated. From here, they extend to the **Manager.java and Customer.java** class, which handle the bulk of their respective implementations. We decided to go this approach as it has a more concise design in terms of the OOP perspectives, and adds a ton of room for scalability in the future (maybe a Supervisor.java that could extend this class). The approach that we decided to take with the account system is that most of the functionality of every account is maintained on this abstract parent class (**Account.java**). From here, this is extended into four important child classes that are the heart of our implementation: **CheckingAccount.java, SavingsAccount.java, LoanAccount.java and Securities.java**. They are pretty straight-forward what they do, but it was important that we made an object-distinction between these four different key players in our implementation. Transactions are maintained and evaluated at the account level according to their specifications.

GUI

To provide an interface that users can easily interact with, we created several front-end classes that extend Java's JFrame class and represent the different frames that appear based on the action the user wants to perform. The application starts off by displaying the frame created from the file **Login.java**, which represents a login that both the manager and the customer can log in from. We decided to use a single screen for this since customers will log in the same way the manager does. From there, a message appears informing the user that they've logged in as either the manager or as a user (customer).

We created our design to allow the manager to have as much control as possible. Once the manager logs in, a frame created from **ManagerFrame.java** appears, giving them the option to register a new customer, check on a customer, generate a report, adjust stock prices, advance the date, view the transaction history in the bank, or log out. The classes **AddStocksFrame.java, AdjustStocksFrame.java, ManagerStockFrame.java, CustomerRegistrationFrame.java,** and **CustomerInfoFrame.java** all support these operations.

We also wanted the customers of the bank to be able to easily perform all the basic functions they would ordinarily seek to perform at a bank, namely, view their account details, make a deposit, make a withdrawl, make a transfer, view their loans, request a loan, and trade stocks. We accomplished this by adding these options as buttons on a frame produced by

CustomerFrame.java. The classes that support these functions include **CustomerAccountsFrame.java, CustomerLoansFrame.java, DepositFrame.java, LoanFrame.java, PayLoanFrame.java, StockMarketFrame.java, TransferFrame.java,** and **WithdrawlFrame.java.**

Persistence

For persistence, we decided to go with the approach of reading and writing files versus using an external database such as SQL. First and foremost, we went with reading and writing files using Java’s io library because of all the tools that Java already provides us to write and parse text files, which all of our team members were familiar with. Our class design had some components with multiple attributes and different types of relationships, which meant that we would probably have to ORM these using JDBC, Hibernate and SQL, which were all libraries that we haven’t worked with in the past. The persistence frame uses a series of folders and text files in order to persist information regarding: customer username and password, account information at the account and at the transaction level, information on any pertinent stock data and the date of the bank.