Object Design Documentation

1. Design Choices:

According to the given assignment statement, we derived the basic functions required by the customer and manager.

Customer:

1. register and log in
2. create/close two basic types of account: checking, saving
3. view balance, saving, withdraw
4. transfer
5. open a security account (if the customer has more than $5000 in his saving account), trade stock through a security account
6. view transaction
7. request loan (use collateral)
8. view personal information

Manager:

1. register and log in
2. manage the users
3. change the currency exchange rate
4. manage the stock market
5. approve/refuse the loan request, charge interest for all loans (automatically executed by the system)
6. view the daily transaction report
7. charge fees for account open, transaction, withdraw (automatically executed by the system) and get financial reports based on above

Simulation Time:

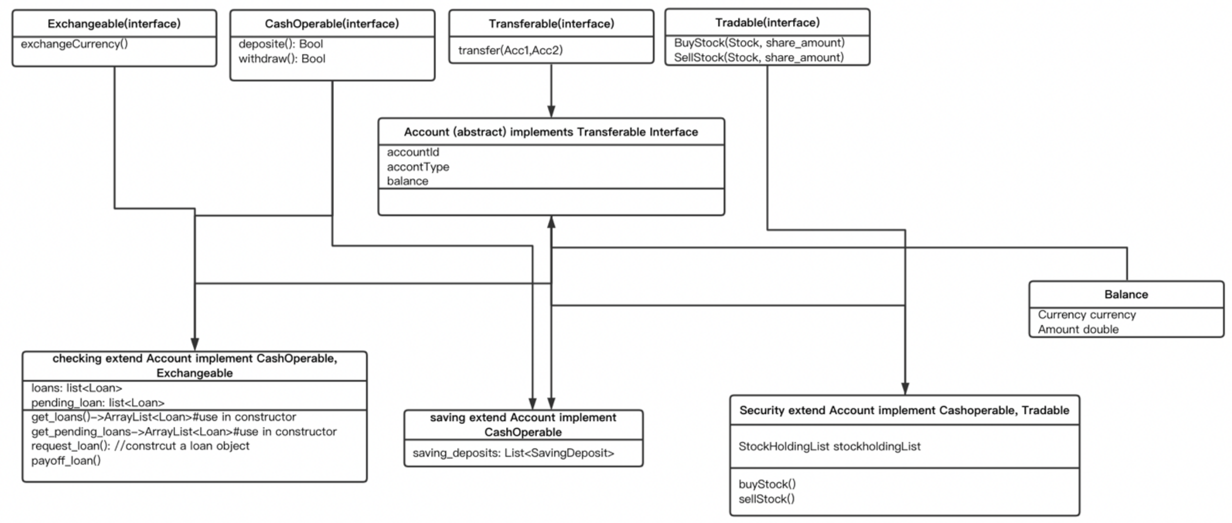
use a time button to control this function, it appears in manager main page

2. Object model:

This project serves as a bank and offers options for users and manager to use. Our persistence mechanism is implemented by storing data in CSV files. The GUI is built with java swing.

There are ten main modules in this project. They are account, currency, loan, stock, transaction, user, bank, data, utils and GUI. Here we use the brief descriptions and simple class diagrams to show the relationships between entity objects. For simplicity, we omit some of the attributes and operations of the classes.

1) Account


Exchangeable(interface)
exchangeCurrency()
CashOperable(interface)
deposite(): Bool
withdraw(): Bool
Transferable(interface)
transfer(Acc1,Acc2)
Tradable(interface)
BuyStock(Stock, share_amount)
SellStock(Stock, share_amount)
Account (abstract) implements Transferable Interface
accountId
accontType
balance
Balance
Currency currency
Amount double
checking extend Account implement CashOperable, Exchangeable
loans: list<Loan>
pending_loan: list<Loan>
get_loans()–>ArrayList<Loan>#use in constructor
get_pending_loans–>ArrayList<Loan>#use in constructor
request_loan(): //constrcut a loan object
payoff_loan()
saving extend Account implement CashOperable
saving_deposits: List<SavingDeposit>
Security extend Account implement Cashoperable, Tradable
StockHoldingList stockholdingList
buyStock()
sellStock()

Account class: (abstract) implements Transferable interface
Account class defines a general bank account object, which contains basic information like account ID and balance. Account class implements Transferable interface because in our belief, any account can transfer money to another account.

Exchangeable: (interface)
This interface defines the behavior of exchanging money currency between two accounts, with the purpose of exchanging between different currencies.

CashOperable:(interface)
This interface defines behaviors that concern cash operations for bank accounts. More precisely, it regulates the cash-related operations that an account could do. Cash operations are withdrawals and deposits.

Transferable: (interface)
This interface defines inter-account money transfer. Any subclass of account class that could transfer money from itself to other accounts should implement this interface. In our case, all accounts should implement this interface.

Tradable: (interface)
This interface defines the behavior of an account that can invest in the stock market. Therefore, important methods like BuyStock, SellStock are included in this interface.

CheckingAccount Class:
extends Account class, implements CashOperable, Exchangeable
The CheckingAccount class represents a bank checking account with specific currency type and contains information of cash loans made with this account.
Given that different checking accounts maintain different currencies, it implements the “Exchangeable interface” so that the checking account serves the currency exchange functionality.
It also implements the CashOperable interface because cash operations are the fundamental purpose of checking accounts.

SavingAccount Class:
extends Account class, implements CashOperable
The SavingAccount is set up for saving money for interest generation and as a source of funding to the security account. For calculating interest, it defines an interest rate variable. It implements CashOperable because direct cash operation on saving accounts should also be available.

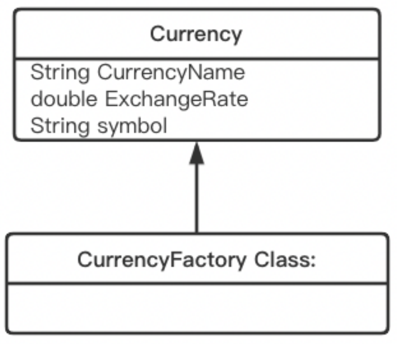
SecurityAccount Class:
extends Account class, implements CashOperable, Tradable
The Security Account class mostly serves stock investment purposes. Apart from the between account transfer function, it allows for stock trading.

Balance Class:
This class represents a bank account balance, which consists of the balance amount as well as the currency type. Within this class, it also handles balance exchange between different currencies, implicitly.

Money Class:
This class represents money in the bank account. It helps to get the currency and amount of the current account.

StockHoldingList Class:
A class that assists with the management of StockHolding objects.

2) Currency

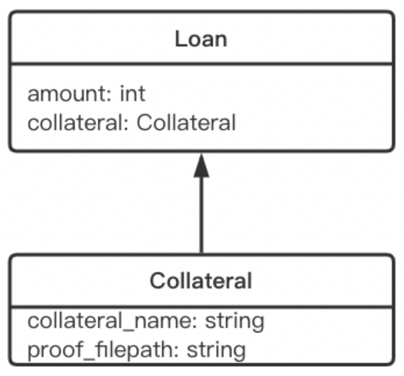


Currency Class:
Currency class represents the status of a currency, which includes the currency symbol, its exchange rate with USD, and currency name.

CurrencyFactory Class:

This class interacts with the database and provides a list of currencies that are available.

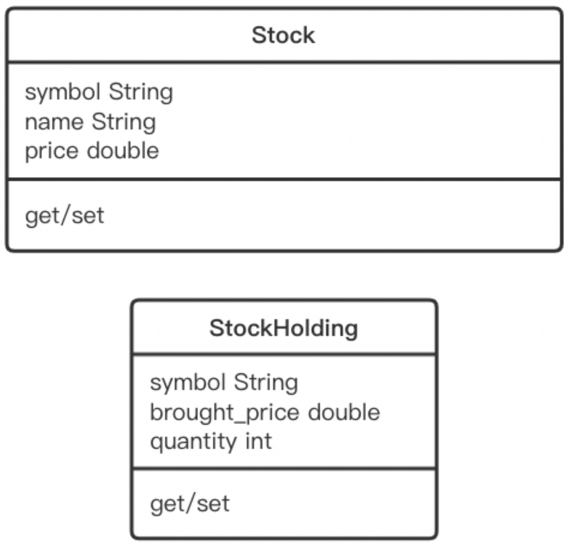
3) Loan



Loan Class:
As customers are allowed to request a loan from the bank, we define a Loan object that represents each loan. Each Loan object should include information about the loan amount, interest rate, a proof of collateral and whether the loan is approved.

Collateral Class:
This class represents an object that is the proof of property that is part of a loan that, upon approval, grants the customer the loan. It is treated as a computer file.

4) Stock



Stock Class:
This class represents a stock object in the market, where the public information of the stock is maintained.

StockHolding Class:
This class represents a piece of stock share that was purchased by a customer. It will contain more transactional information like purchase price, amount of share purchased. This will aid us in determining the realized profit and unrealized profit.

StockMarket Class:
This class represents the entire stock market, where we employed the singleton design pattern to ensure the uniqueness. This class interacts with data storage and provides information of stock market prices, stock purchase availability.

5) Transaction

Transaction class:(abstract)
To keep a record of any bank atm operations, we introduce the transaction class that records that specific information regarding each transaction.

The common variables for any transaction are amount, operation type, transaction source, transaction target.

CashTransaction Class: extends Transaction class
This type of transaction is cash related transaction and it’s to record when accounts make deposits or withdrawals.

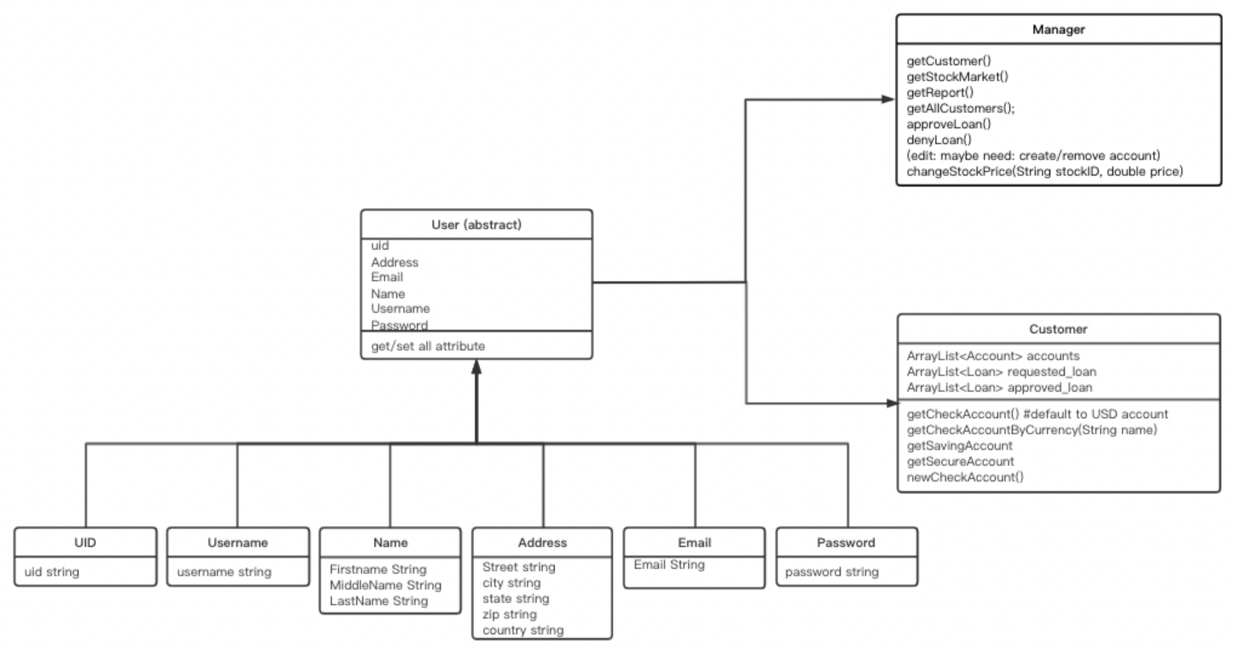
TransferTransaction Class: extends Transaction class
This type of transaction is a transfer transaction, and it’s created when accounts transfer money to other accounts.

AccountTransaction Class: extends Transaction class
This transaction type records account opening and closing.

InterestTransaction Class: extends Transaction class
This type of transaction is interest related transaction and it’s to record when paying the loan interest.

TransactionType Class:
This is an Enumerated types to enumerate all transaction types.

6) User



User class(abstract):
This class defines any user who could operate on the ATM machine. Anyone who could operate on the machine needed to login to the machine and provide personal information for verification. Therefore, the class should have member variables like UID and PersonalInfo, which is the intuition for designing this class. The two classes that extend this object would be Customer class and BankManager class.

Personal Info related class:

- UID class

This class contains UID information.

- Username class:

This class contains related username.

- Name class:

This class contains all name related information: first name, middle name, last name.

- Email class:

This class contains email information.

- Address class:

This class contains all address related information.

- Password class:

This class contains related password.

This class has many attributes that define and help identify a person. Examples of class variables are personal address, name.

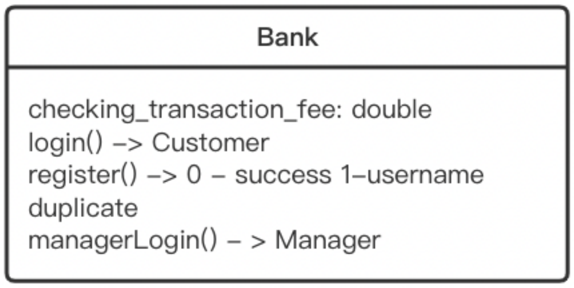
Customer Class:(extends User class)
This class represents an instance of a bank customer, an instance who could operate the ATM. A bank customer has his/her personal accounts with different purposes. A customer can have multiple checking accounts that support multi-currency, saving accounts for generating interest revenue and security accounts for investment.

Manager Class:(extends User class)
A bank manager is also a user of the ATM, but the operations are fundamentally different from that of customers. The bank manager can access information of all customers, define trading availability of stocks and manipulate stock prices.

Report Class:
This is a class represents detailed report when paying interests and charging fees.

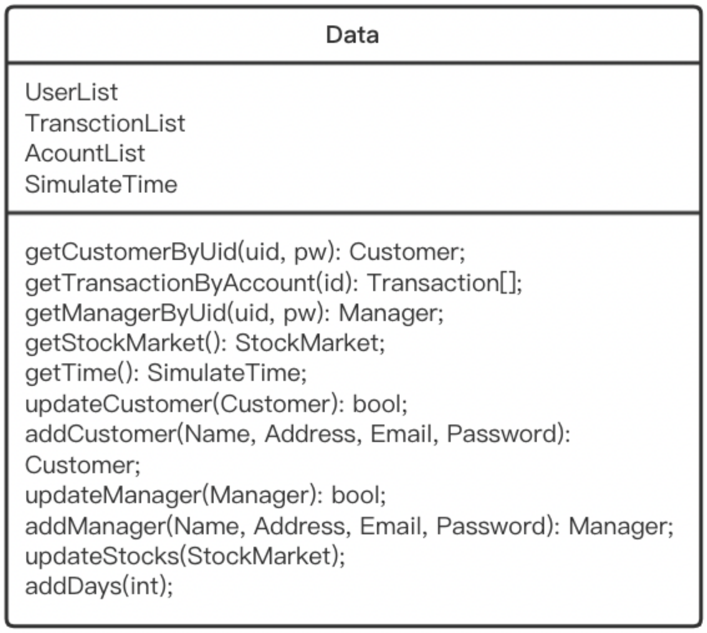
FeeType Class:
This is an Enumerated types to enumerate all fee types.

7) Bank


Bank
checking_transaction_fee: double
login() –> Customer
register() –> 0 – success 1–username duplicate
managerLogin() – > Manager

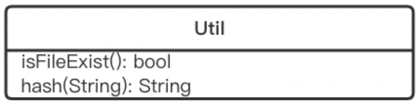
Bank Class:
This class represents an instance of a bank. Two types of users can use the bank to register and log in. Also, the user who is given the administrative rights can modify the interest rate.

8) Data


Data
UserList
TransctionList
AcountList
SimulateTime
getCustomerByUid(uid, pw): Customer;
getTransactionByAccount(id): Transaction[];
getManagerByUid(uid, pw): Manager;
getStockMarket(): StockMarket;
getTime(): SimulateTime;
updateCustomer(Customer): bool;
addCustomer(Name, Address, Email, Password): Customer;
updateManager(Manager): bool;
addManager(Name, Address, Email, Password): Manager;
updateStocks(StockMarket);
addDays(int);

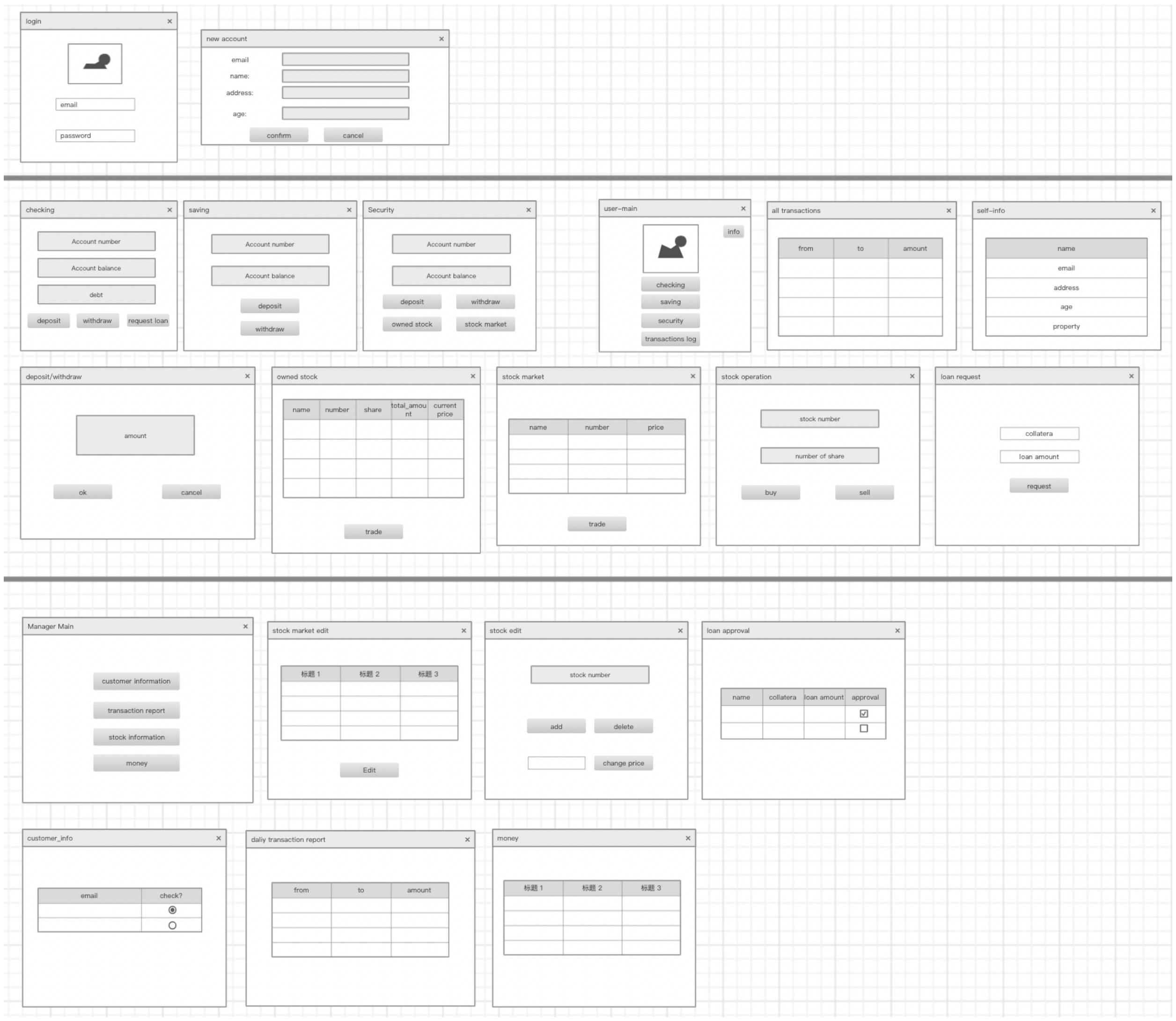
This module includes the data and data processing parts.

9) Utils



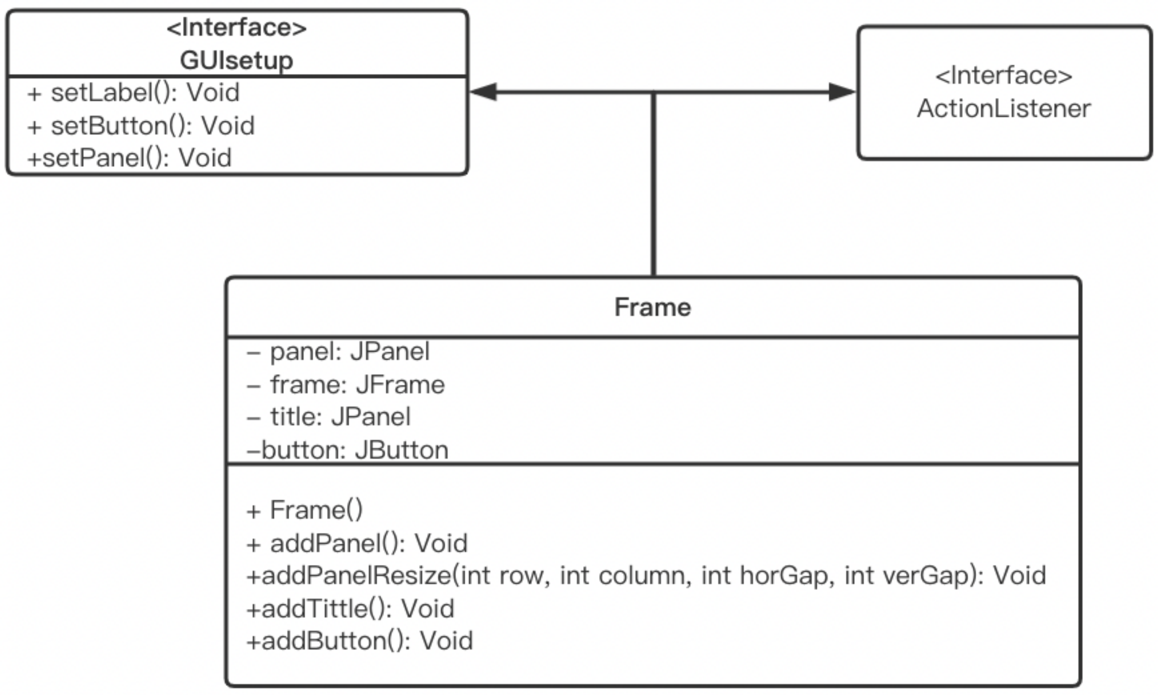
This module is a data connected module. If the database (files which used to store the data) exists, do connection.

10)GUI



The UI pages is set up with java swing. The prototype diagrams are shown above.

3. Object and GUI relationship



For frontend, we used JAVA swing to implement. Since the creation of each page allows us to pass in objects as constructor arguments, we took advantage of that and kept passing necessary object references that closely pertained to the current page. Many of the backend functions are triggered by click of a button, therefore, we execute these backend-provided methods like "account.transfer(targetAccount, Balance)" as the user clicks the "submit" button. Therefore, the backend provides the function implementation while the front end calls these functions based on the user input.

ApproveLoanPage:
GUI of manager approving the loan with its features.

CheckingAccountPage:
GUI of checking account with its features.
CustomerEditPage:
GUI of process of manager editing an existed customer.
CustomerInfoPage:
GUI of showing customer information.
CustomerListPage:
GUI of showing existed customers in the bank system.
CustomerMainPage:
GUI of main customer menu that displays after successful login.
DailyTransactionPage:
GUI of showing daily transactions for manager.
ExchangePage:
GUI of processing currency exchange.
ExchangeRateChangePage:
GUI of manager processing existed currency exchange rate.
FileChooserPage:
GUI of customer choosing a proof for loan.
FinancialReportPage:
GUI of showing financial report for manager which including the fees and interests.
LoanDetailPage:
GUI of showing detail information of customers who requested the loan for manager.
LoginPage:
GUI of processing the log in function for two types of users.
ManagerMainPage:
GUI of main manager menu that displays after successful login.
MoneyOperationPage:
GUI of processing money operations including the deposit, withdraw, paying for the loan.
NewCustomerPage:
GUI of creating a user.
OwnStockPage:
GUI of Showing the owned stocks with its features.

RequestLoanPage:
GUI of requesting a loan by the customer.
SavingAccountPage:
GUI of saving account with its features.
SecurityAccountPage:
GUI of security account with its features.
StockEditPage:
GUI of editing existed stocks.
StockMarketPage:
GUI of showing the stock market.
StockOperationPage:
GUI of operating the stocks.
TransactionsPage:
GUI of showing transactions report for customer.
TransferPage:
GUI of processing the transfer function.

4. Benefit of the design

Starting from account, we design 3 types of accounts, checking, saving and security accounts. For both saving account and security account, there only exist accounts in US Dollar while we allow checking accounts with different Currencies, with US Dollar still being the default currency. Because different account types have common and different functionality, we define multiple operation interfaces to further specify the behaviors of different account types. While all accounts can transfer money, only checking accounts and saving accounts can make cash operations and only checking accounts support currency exchange. To allow customers to maintain cash in different currencies, each customer is allowed to have different multiple checking accounts that support currency conversion, as stated above. Talking about the account here we use the interface to support the variable functions in different accounts. And three sub classes inherit the abstract account class. It's a very elegant method to realize the OOD design. For loan function, this is only about the checking account.

The customer can pay the loan using money operation function and the bank will charge some interest on it. Also, we add a proof file upload function. After requesting the loan, the bank manager can browse the details for approval. We believe this kind of design is more in line with the reality.

The project is scalable to the point of handling data in database and compliant data added when running it. The approach that we used to implement our simulated database is to serialize handlers that contains all related object into a JavaScript Object Notation (JSON) string and save it to files. We included an external library GSON for the serializing feature. We choose this approach because objects only need to be loaded from files once when the system starts, and files would be simultaneity updated every time that any related object changes.