Design Document for Bank ATM Project

Course: CS591

Team #1

Camilla Satte (U19891895)
Bryan Zhang (U96291265)
Chenhao Tao (U93652744)
Beatrice Tanaga (U90936184)

Content:

Summary:

To provide a brief summary of our Bank ATM design, we built it to utilize reasonable inheritance, abstraction, and composition relationships between all of the classes and interfaces we had. In order to make the program more reusable and extendable, we tried not to have hard coded variables and structured the code as clearly as possible so that it would be very readable and understandable for external users. You will find in the src folder that we segmented our classes into 4 main packages: account, bankATM, database, and manager.

Technologies Used:

Java Eclipse IDE, MySQL Database, LucidChart (UML), Adobe XD (GUI design).

Design Pattern:

The design pattern we used is Models-View-Controller. The Model objects retrieve and store model object's modifications in the database. For example, a Client object can retrieve information that is allowed to retrieve (Client's balance, stocks, transactions, account information) from the Bank database, Client can execute operations (perform transactions Withdraw, Transfer, Deposit, etc) that will update the Bank database. Here, a Client (aka Model) has access to change only his own data. The View objects are available methods that are accessible to Client via GUI. All View components are displayed in our simple graphical user interface, that displays all available buttons depending on the Client's state. For example, a Client with less than 10,000$ in a Savings Account doesn't have access to the "Stocks" button, the "Stocks" button is enabled only for qualified Clients. Controllers components handle Client's interaction with the Bank. The Manager is the main Controller, and we defined interfaces that Manager implements: LoanController, AccountController, StockController, Interest and Fee Controllers. These controllers are maintained by the Bank Manager.

Also, on top of MVC pattern we have followed some of Data Access Object PatternThe main reason we used it is to separate low level database operations from the higher level bank service operations. The package `database` has a CRUD (Create-Retrieve-Update-Delete) interface which contains essential database operations. For each table in Bank Schema, we have created a separate class that implements the CRUD interface because each table corresponds to different Objects (Bank, Client, Manager, Stocks, Transaction, etc).

BACKEND. Implementation and Design.

Information on each package and the classes it contains:

[Account package]
Here you will find our abstract class Account.java, which all the other classes such as CheckingAccount.java, SavingsAccount.java, and SecuritiesAccount.java extend from. The reason we made this abstract class Account.java is because all the other 'types' of accounts would have very similar general functionalities and variables pertaining to each customer's personal information.

We then have the default account DepositAccount.java, where customers can choose to deposit their money. This account is purely for the customer to 'store' their money and there are no other functionalities in this class. If for instance, any of the other types of accounts are closed by the customer, he/she will still have this DepositAccount and all remaining money in the other accounts prior to their closure will automatically be transferred to the customer's DespositAccount. Henceforth, the DepositAccount will always exist and cannot be closed by the customer.

The other types of basic accounts are CheckingAccount, SavingsAccount and SecurityAccount, and each are defined in their own classes. These classes have the required specifications of each type of the account given in the assignment. The main difference between the CheckingAccount and the SavingsAccount is that the CheckingAccount implements a service fee (from the serviceFee interface, which we will elaborate on later) for every transaction that takes place, while the SavingsAccount gives an interest if the customer once the customer reaches the balance qualified for gaining interests. The SecurityAccount is the account where the customer can view their stock inventory or purchase new stocks.

The last type of 'account' we have is the LoansAccount. This class stores all the information of each customer's debt, if he/she has requested any loans and was approved by the bank. It contains functions to keep track of how much of the debt has been repaid, and how much he/she still owes the bank.

Finally, the last extra class we have is the OpenClose.java class which is an abstract class that simply declares the functionality of opening and closing accounts in each of the different types of account classes.

[bankATM package]
The bankATM package contains all the functionalities that the actual ATM provides for each customer. The main class which takes care of running the entire program is also found in this package.

First we have the Bank.java class. This is the primary class that holds the variables of all information the client needs to see when he/she is accessing his/her bank ATM account (e.g., withdrawal fees, current balance). The class also contains getters and setters for each of these variables. We also have a function findClientByEmail, which returns the client's credentials

through their email. If the client does not exist, this function will return null. We created this function so that it would be easy for the bank to retrieve a client's information from the database just by providing his/her unique email address.

Next we have the Client.java class, which contains all of each client's personal and private information such as their customer id and account password. The purpose of this class is to keep a record of what each customer does within his/her accounts, or if a new customer signs up for a default DepositAccount. It holds information such as when an account is opened or closed, or all past transactions made within the existing accounts. The settlement of any debt repayment is also done in this class. As each action is performed by the customer, the database is also updated accordingly. The customers themselves are able to access the information and functionalities kept within this class.

Next we have the Loan.java class. As the name suggests, this class takes care of any new or existing loans taken up by each customer. The construction of new loans (if the request has been approved) and any applicable interests to existing loans are taken care of in this class. The customer is also able to see his/her loan request status after it has been submitted. All in all, this class takes care of any functions pertaining to a customer's approved loan or pending loan request.

Next we have the Money.java class. All transactions between accounts are made in USD, although each customer may view a different currency when accessing their accounts. This means that although in the front end a customer may view their balance to be in a foreign currency, the amount will be converted in the backend to USD when a transaction is made and then converted back to whatever currency the receiving account is in when it appears on that account. All operations pertaining to this along with basic operations such as addition and subtraction of money from account balances is done within this class.

Next we have the Person.java class. As stated in our UML, both a customer and a bank manager are instances of a person. Hence this class holds all information regarding a person's id, name, birthdate, phone number, city and country that they live in. This class mainly contains the getters and setters of those variables, and a toString method to display such information when needed.

Next we have the Stock.java class. This class holds the variables containing the information of each particular stock. For instance, their stock id, name, price, and quantity available. Besides the getters and setters for these variables, the class also contains the addToDB function that updates the database when a new stock option is created, and a toString method to display all these information to the customer if he/she wants to purchase a new stock.

Next we have the PurchasedStock.java class, which as the name suggests carries all operations that can be done on stocks already bought by the customer (and stored in their security account). The constructor contains the information of each stock purchase, such as the

type, quantity and price. Another important reason we have a PurchasedStock object is to calculate whether it is Realized/Unrealized because it will save purchased price. For example, Client bought Stock X at day 1 for price 1$, and then he decides to buy Stock X at day 5 (but the price is bigger now 10$), so the PurchasedStock will have different purchased prices . When a stock is purchased, the database is also updated to keep track of each customer's stock inventory. And when a purchased stock is sold by the customer, the database is also updated. The sell function in this class takes care of that operation. Similarly, we have the SoldStock.java class that holds the information of any stock sold by a customer.

We have 3 enum classes in the bankATM package. The first is the Currency.java class, which is simply an enum class that contains the different currencies available or provided by the bank, and the exchange rates of each one respectively. Next we have the Status.java class that contains enumerates for each possible status a transaction or a variable can have. For example, a transaction or a loan request can be pending, completed or cancelled. And a stock can be marked as sold or purchased. All of such statuses are listed in this class. It also contains a toString method, if in any case the status of a transaction needs to be displayed to the customer or manager. Finally there's the Type.java class which contains the possible transaction types, stock types and account types, as well as a toString method for display purposes.

Finally, there are 2 interfaces declared in this package. The first one is the Interest.java interface. This interface defines the getInterest and setInterest functions that needs to be individually implemented in the other classes where interests applies, such as SavingsAccount.java and LoansAccount.java. The second one is the ServiceFee.java interface. Similarly, this interface contains the getter and setter abstract functions to be defined in other classes where they are needed. The reason why we chose to declare these as interfaces is so that we can have multiple inheritances in some of the account types, where they need to inherit both the default abstract class Account.java and also specific fees for transactions and interests. It is also because different types of transactions or 'balances' (e.g., savings account balance, loan/debt) have different fees or interest rates.

[Manager package]

The manager package contains the class and interfaces that hold specific functionalities that is only accessible/doable for bank managers, and not the clients. The main class, Manager.java, takes in all the other interfaces declared under this package. The manager also has credentials such as manager id, email, and password. Since the job of the manager is to attend to all activities regarding the bank accounts and stock options, he/she is able to adjust the fees for each operation the customer performs (those that already incur a fee) such as withdrawals, opening and closing of accounts, and the interest rates of loans and savings account balances. The manager can also update the available stock options that a customer sees on the stock page. An important function that is declared in this class is the collectLoanPayment function. In a case where a customer who has taken up a loan from the bank fails to repay a certain amount

within 30 days of the initial grant, the manager has the ability to collect the payment from the customer's deposit account.

Outside of this main Manager.java class, we have 5 interfaces declared in this package. There are four controller classes: InterestController.java, LoanController.java, ServiceFeeController.java, and StockController.java. These controller interfaces only contain the abstract getters and setters that are defined in the Manager.java class. The interface Settings.java also only contains the abstract setter of the current date to be displayed. The reason why we chose to implement these as interfaces is so the Manager.java class is able to inherit all of the separate controller classes.

[Transaction Package]
First we have the primary abstract class Trnasaction.java. Since each transaction incurs a service fee, it inherits from the ServiceFee interface. The constructor of each transaction holds the basic information such as transaction id, account (from which the transaction was made from), the amount of money, the date the transaction was made on, and the status. Every time a new transaction is made, it is added to the database. Besides the getters and setters for these information, the class also has a toString method to display the information to either the client when they want to check their past transactions, or the manager if he/she needs to consolidate that day or week's transactions.

Next we have the various classes that inherits from the Transaction.java abstract class. First is the Deposit.java class. This class contains the constructor for any deposits made by a client. The constructor initializes the basic information such as the deposit (transaction) id, account (the money is being deposited to), the amount of money, the date the deposit was made on and the status. Since deposits do not incur a service fee, the getter for service fee is set to 0.

Next we have the Withdraw.java class, which is very similar to the Desposit.java class except that withdrawals do incur a service fee. The getServiceFee function in this class has 2 cases, one where the withdrawal is from a savings account, and the other is when the withdrawal is from a savings account, since the service fee is different in the two cases.

Next we have the Transfer.java class, which again is very similar to the previous 2 mentioned, except that it takes in 2 account parameters. One for where the money is being taken from, and the other is for where the money is going to. Similar to deposits, transfers have their own service fee implemented for the different types of accounts the transfers are being made from.

Finally, we have the LoanPayment.java class. This class is specific to customers who have loan debts to pay off. The constructor is similar to the other types of transactions, containing an id, account, amount, date, and status. Loan payments do not have a service fee, hence the getServiceFee function is set to 0 once again.

[Database package]

This package holds all the classes and interfaces pertaining to updating the backend database connected to our bank ATM. For the database, we chose to use MySQL with the JDBC Driver java connector.

The DataBaseConnection.java class establishes the connection to the MySQL server. It has all of the credentials to connect to our server, such as the account name, password (this is unique to what password you created when you installed the server), the port number, and the database name, which in our case is 'bankDB'. It also contains the main function that initializes the connection.

Next we have the CRUDInterface.java interface, which like the name suggests defines the abstract functions for create, read, update and delete operations inherited and used in the other classes which we will go through later below.

To avoid redundancy, we will use the DBAccount.java class as an example to explain the functionality of all the other classes in this package. To start, all existing accounts are created and stored in the database through this class. In order to retrieve a specific person's account, the access to the database is made in this class and the information is pulled from the database using the retrieveClientAccount or the retrieveById functions. If for instance an account is closed, the account will also be deleted from the database through this class. Updates to any existing accounts (if a transaction or any other types of operations were made by either the client or the manager) will also be taken care of here. All the other classes such as DBBBank.java, DBClient.java and DBManager.java within this package hold the same CRUD based operations pertaining to all operations done in the backend of the bank ATM. All operations done within the system would need to be recorded and hence updated in the database server. All that is taken care of within the classes in this package.

FRONTEND. Implementation and Design.

The front-end consists of various classes that represent the flow of how the user is intended to use the Banking application. Beginning with LoginPage.java, this class prompts the user to login as either a Client or a Manager. Clients who are new to the bank can proceed to create their account here as well. Once a Client is logged in they can then go to the Client Home Page (represented by ClientHomePage.java) where there are buttons that can be used to access the various account features. Clicking the Accounts button leads to pages where a Client can access his/her various accounts and create new ones as well; this is handled in ClientAccountInfo.java.

Then, all the different features can be accessed in different packages. Starting with the Account management part, Clients can view their account transactions in the View_Account_Trans.java with a scrolling panel. And once the Client wants to close the account, it will give them options in

the Account_close.java to either transfer or withdraw the remaining money in the account and then pass to the corresponding case page. At deposit_Page.java, a Client can choose to deposit accounts he/she has, - Saving, Checking, and Deposit. Then, by clicking the options button, it will jump to the selected account page and require the Client to enter the amount of money to deposit. After that, it should've sent a successful deposit message or an error message If the entered amount is over his balance in that account. Withdrawing features has the same process as the deposit (covered in Withdraw package). For the Loan, on its main page(Loan_Page.java), Clients can choose between “request a loan” or “pay a loan”. Then it will lead to pages where clients can enter the amount of loan they want to borrow or pay. Once the transaction is completed, it will send the Client to a successful message page otherwise to the Failed page.

Next, at the Stock Home page, Clients can choose to buy a stock or to sell his/her own and view his/her personal stock history. For buying a stock, it will jump to the Stock market, Clients can select their interesting stock to buy from a stock list. By clicking the select button, it will lead to a page that gives specific info for that selected stock. Then if the client decides to buy after reviewing this detail of this stock, it will require clients to enter the amount of stock they want to purchase. Once the transaction is confirmed, it will send the Client to a successful message page otherwise to the Failed page. Same to the Selling stock, it has the same steps for clients, but the difference is that, instead of going to the stock market, it will send clients to My Stock Page to give them a look at the stock they have. And after selecting the stock they want to sell, it will send a successful message or failed message based on the transaction result. (represented by Success_trans.java and Failed_trans.java in the Stock package) Finally, in the Mytrans package, it gives clients to view their monthly transaction history by entering the corresponding year and month.

Meanwhile, if a Manager logs in, they are led to their own Manager Home Page, which has buttons that can be used to access Manager features (this is in ManageHomePage.java). ManagerHomePage has various buttons and methods to handle what happens when a button is clicked. Each button has a corresponding method that leads to a new frame that represents something like Transactions, Loans, Stocks, etc. ManageLoans.java, ManageServiceFees.java, ManageTransactions.java, ManageStocks.java, and ManageInterest.java respectively contain the GUI interfaces for a Manager to deal with loans, service fees, transactions, stocks, and interest rates. To view accounts, the method createManageAccounts() within ManagerHomePage.java is used.

DATABASE. Implementation and Design.

The database design for Banking system schema consists of 10 tables.

1. **Bank**. State of the bank. It has information about all different fees, interest and loans rates, overall bank balance, current date (to make testing easier, all object creations and transaction operations are executed depending on the bank's current time).
2. **Person.** To emulate the real world, we defined this database to test virtual populations that exist in the BankATM reality. It has basic information that every human being has. For example: Full Name, Birthdate, Location. Not all humans can become Bank Clients, some humans can be Bank Managers and also be Bank Clients, or just be Clients.
3. **Managers.** List of managers that have access to safely login and control the accounts and Bank state.
4. **Clients.** This is for Client information, to maintain and control accounts, stocks, and loans. Every Client has a unique id that is associated with a person. Clients can have many accounts.
5. **Accounts**. Every account is of different types (Checking, Savings, Loans, Deposit, Security). Every account has its own separate balance, and can perform a specific set of operations depending on the account type. The “type” column helps to create the correct AccountType Object which extended from the abstract Account.
6. **Loans.** Loan information includes the total amount left to pay, the last paid date, initially fixed interest that adds up every month, and also status (Approved or Rejected). The loan is deleted from DB upon reaching Client’s owe amount to zero.
7. **Stocks.** The stocks are existing independently from the bank, but the prices can be controlled through the manager's interface.
8. **Purchased_Stocks.** It is separate from Stocks because every time a Client buys the stock, the purchase price is different. Inorder to calculate realized/unrealized and move it to Sold_Stocks.
9. **Sold_Stocks.** It is separate from Stocks and Purchased_Stocks for easier access, and just to keep in history.
10. **Transactions.** This table contains ALL transactions made by clients and managers. Essential information such as amount of money, date, account, the type and status of transaction are accessible and available for the bank manager to approve or reject.

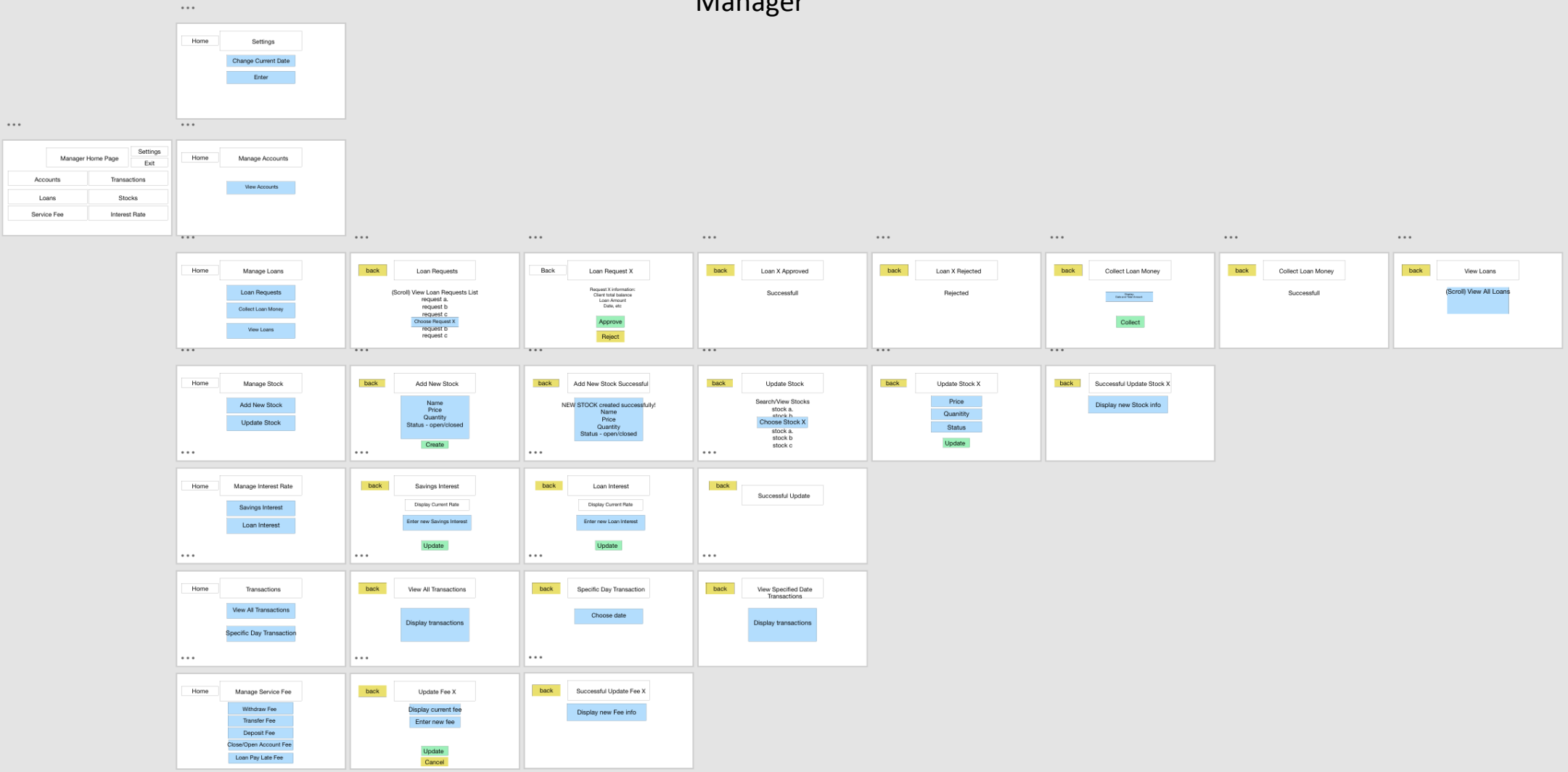
Comments.

Rough. Tough. Don’t Cough.

Diagrams:

Link to access interactive GUI Design:
https://xd.adobe.com/view/fc97a59e-5242-4d8c-7ec7-1dd15fbb80b1-8aa6/

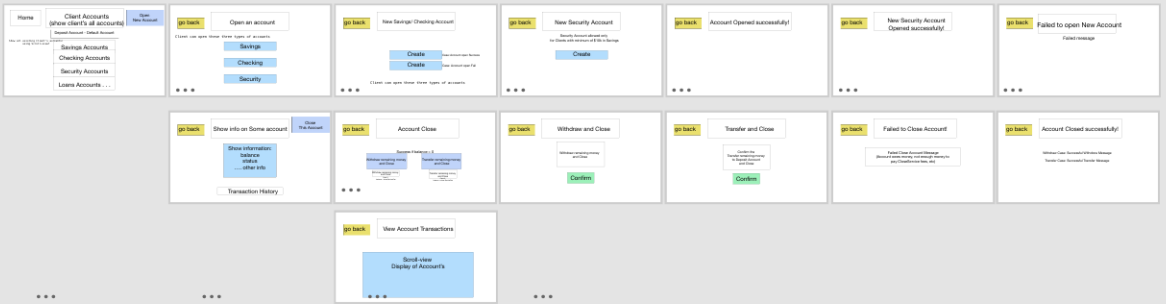
Manager


Manager Home Page
Settings
Exit
Accounts
Transactions
Loans
Stocks
Service Fee
Interest Rate
Home
Settings
Change Current Date
Enter
Home
Manage Accounts
View Accounts
Home
Manage Loans
Loan Requests
Collect Loan Money
View Loans
back
Loan Requests
(Scroll) View Loan Requests List
request a.
request b
request c
Choose Request X
request b
request c
Back
Loan Request X
Request X information:
Client total balance
Loan Amount
Date, etc
Approve
Reject
back
Loan X Approved
Successfull
back
Loan X Rejected
Rejected
back
Collect Loan Money
Collect
back
Collect Loan Money
Successfull
back
View Loans
(Scroll) View All Loans
Home
Manage Stock
Add New Stock
Update Stock
back
Add New Stock
Name
Price
Quantity
Status - open/closed
Create
back
Add New Stock Successful
NEW STOCK created successfully!
Name
Price
Quantity
Status - open/closed
back
Update Stock
Search/View Stocks
stock a.
stock b
Choose Stock X
stock a.
stock b
stock c
back
Update Stock X
Price
Quanitity
Status
Update
back
Successful Update Stock X
Display new Stock info
Home
Manage Interest Rate
Savings Interest
Loan Interest
back
Savings Interest
Display Current Rate
Enter new Savings Interest
Update
back
Loan Interest
Display Current Rate
Enter new Loan Interest
Update
back
Successful Update
Home
Transactions
View All Transactions
Specific Day Transaction
back
View All Transactions
Display transactions
back
Specific Day Transaction
Choose date
back
View Specified Date Transactions
Display transactions
Home
Manage Service Fee
Withdraw Fee
Transfer Fee
Deposit Fee
Close/Open Account Fee
Loan Pay Late Fee
back
Update Fee X
Display current fee
Enter new fee
Update
Cancel
back
Successful Update Fee X
Display new Fee info

Client


Welcome to the bank!
Clients
Manager Login
go back
Manager Login Page
UserName
Password
Login
go back
Client Login Page
UserName
Password
Login
New Client
go back
New Client Registration
First Name
Last Name
Birth_date
Phone
City
Country
Next
go back
New Client Registration
Email (also used as username)
Password
Register


Client Home Page
Settings
Exit
Accounts
Deposit
Loans
Transfer
Withdraw
Stocks
Transaction History

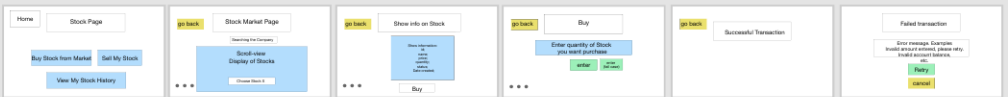

Home
Client Accounts
(show client's all accounts)
Open New Account
Savings Accounts . . .
Checking Accounts
Security Accounts
Loans Accounts . . .
go back
Open an account
Savings
Checking
Security
go back
New Savings/ Checking Account
Create
Create
go back
New Security Account
Create
go back
Account Opened successfully!
go back
New Security Account Opened successfully!
go back
Failed to open New Account
Failed message
go back
Show info on Some account
Close The Account
Show information:
balance
status
.....other info
Transaction History
go back
Account Close
go back
Withdraw and Close
Confirm
go back
Transfer and Close
Confirm
go back
Failed to Close Account!
go back
Account Closed successfully!
go back
View Account Transactions
Scroll-view
Display of Account's


Home
Deposit Page
Savings Accounts
Checking Accounts
Deposit Accounts
go back
Deposit To Account X
Enter Amount of Money to Deposit
Currency
Enter
Enter
fail case
go back
Failed Deposit
Failed Deposit Message
go back
Successful Deposit


Home
Withdraw Page
choose account transfer from...
Savings Accounts
Checking Accounts
go back
Withdraw Page
Enter Amount of Money to Withdraw
Currency
enter
Enter
fail case
go back
Successful Withdrawal
go back
Failed Withdrawal
Invalid amount entered, please retry.


Home
Transfer Page
choose account transfer from...
Savings Accounts
Checking Accounts
...Other Accounts
go back
Transfer Page
choose account transfer to...
Between my accounts
Transfer to another Client
go back
Transfer Page
choose my account transfer to...
Savings Accounts
Checking Accounts
...Other Accounts
go back
Transfer Page
Another Client email
enter
go back
Incorrect Account
Invalid email
Retry
go back
Transfer Page
Enter Amount of Money to Withdraw
Currency
enter
go back
Successful Transfer
go back
Failed Transfer
Invalid amount entered, please retry.


Home
Loan Page
Request a loan
Pay a loan
go back
Loan Request Page
Enter Request Amount
Currency
enter
go back
Loan Request
Display Success Message
go back
Loan Payment Page
Debit Amount
Currency
Enter Payment Amount
Currency
enter
enter
fail cases
go back
Your Payment Was Successful
Leftover Debit Amount
Currency
go back
Payment Failed
Invalid amount entered


Home
Stock Page
Buy Stock from Market
Sell My Stock
View My Stock History
go back
Stock Market Page
Searching the Company
Scroll-view
Display of Stocks
Choose Stock #
go back
Show info on Stock
Buy
go back
Buy
Enter quantity of Stock
you want purchase
enter
enter
fail cases
go back
Successful Transaction
Failed transaction
Error message. Examples:
Invalid amount entered, please retry.
Invalid account balance,
etc.
Retry
cancel
go back
My Stock Page
Searching the Company
Scroll-view
Display of Stocks
Choose Stock #
go back
Show info on Stock
Sell
go back
Sell
Enter Amount of Stock
you want sell
enter
enter
fail case
cancel
go back
My Stock History
Searching the Company
Scroll-view
Display of Client's Stocks Transactions


Home
View My All Transactions
Enter Month year
enter
go back
View My All Transactions
Scroll-view
Display of Client's All Transactions
During Entered Month - year

Bank UML

Camilla Satte | May 6, 2020

Backend

Bank

+name
+chekingAccountFee
+withdrawFee
+savingsInterest
+loanInterest
+closeAccount
+openAccount
+depositFee
+balance
+currentDate

+getName()
+setName()
+getCheckingAccountFee()
+setCheckingAccountFee()
+getWithdrawFee()
+setWithdrawFee()
+getSavingsInterest()
+setSavingsInterest()
+getLoansInterest()
+setLoanInterest()
+getBalance()
+setBalance()
+addToBalance()
+getCurrentDate()
+setCurrentDate()
+getCloseAccountFee()
+setCloseAccountFee()
+getOpenAccountFee()
+setOpenAccountFee
+getTransferFee()
+setTransferFee()
+getDepositFee()
+setDepositeFee()
+findClientByEmail(String) : Client

Settings

+setCurrentDate(Date) : void

implements

LoanController

+getLoans() : ArrayList
+approveLoan(Account) : boolean
+issueLoan(Account) : Loan
+ collectLoanPayment(Date) : void

StockController

+addStock(Stock) : void
+setStatus(Stock) : void
+setPrice(Stock , Money) : void
+setQuantity(Stock ,int) : void

ServiceFeeController

+setWithdrawServiceFee(Money) : void
+setTranserServiceFee(Money) : void
+setDepositServiceFee(Money) : void
+setCloseAccountFee(Money) : void
+setOpenAccountFee(Money) : void

InterestController

+setSavingsInterest(double) : void
+setLoanInterest(double) : void
+ payAllInterests() : void

BankManager

+id
+person : Person
+email
+password

+openAccount()
+closeAccount()
+collectMoney()
+issueLoan()
+addStock()
+setIsOpen()
+setPrice()
+setQuantity()
+setWithdrawServiceFee()
+setTranserServiceFee()
+setDepositServiceFee()
+setSavingsInterest()
+setLoanInterest()

isA
manages
implements
implements
implements
implements
manages

Money

+value : double
+currency :enum Currency

+setValue()
+getValue()
+setCurrency()
+getCurrency()
+add(other Money)
+conversion(other Currency)

Currency

enum : USD, EUR, GBP, ...
rate : float
string : String

Constant

Type : enum
{ Withdraw, Deposit, Transfer..., }
Status : enum
{ Pending, Complete, Requested, Approved }

Client

+id
+person
+created : Date
+email
+password

+getId()
+setId()
+getPerson()
+setPerson()
+getCreated()
+setCreated()
+getEmail()
+setEmail()
+getPassword()
+setPassword()
+openDepositAccount()
+openCheckingAccount()
+openSavingsAccount()
+openSecurityAccount()
+openLoansAccount()
+getAllAccount()
+getAccountsOfType()
+getDepositAccount()
+getLoansAccount()
+getTransactions()
+requestLoan()
+payBank()
+updateDB()
+getBank()
+closeAccount()

Text
isA
owns

Person

+id
+firstName
+lastName
+birthDate : Date
+phone
+city
+country

+getId()
+setId()
+getFirstName()
+setFirstName()
+getLastName()
+setLastName()
+getBirthDate()
+setBirthDate()
+getPhone()
+setPhone()
+getCity()
+setCity()
+getCountry()
+setCountry()

Account

+Client
+id : account no
+balance : Enum
+status : Enum
+created : Date
+balance : Money
+type

+addToDB() : void
+getId()
+setId()
+getNewId()
+getClient()
+setClient()
+getStatus()
+setStatus()
+getBalance()
+setBalance()
+getCreated()
+setCreated()
+getType()
+setType()
+deposit(amount) : Deposit
+transfer(amount, destination) : Transfer
+withdraw(amount) : Withdraw
+getTransactions() : ArrayList
+getInfo()
+updateDB()

implements
implements
Modifies
views
maintains

ServiceFee

+setServiceFee()
+getServiceFee()

Transaction

+id
+account : Account
+client : Client
+created : Date
+amount :Money
+status : Enum
+type : Enum

+addToDB()
+getId()
+setId()
+getNewId()
+getAccount)
+setAccount()
+getTransactionId()
+setTransactionId()
+getCreatedOn()
+setCreatedOn()
+getNewCreated()
+getAmount()
+setAmount()
+getClient()
+setClient()
+getStatus()
+setStatus
+getType()
+setType()
+getIndo()
+updateDB()

isA

Stock

+id : String
+name : String
+price :Money
+quantity : int
+status : String
+created : Date

+addToDB()
+getId()
+getNewId()
+setId()
+getName()
+setName()
+getPrice()
+setPrice()

Withdraw

+Account
+id
+created : Date
+amount
+status : Enum
+type : Enum

+withdraw()

Transfer

+Account
+id
+created : Date
+amount
+status : Enum
+type : Enum
+destinationAccount : Account

+transfer()

Deposit

+Account
+id
+created : Date
+amount
+status : Enum
+type : Enum

+deposit()

StockOperation

+Account
+id
+created : Date
+amount
+status : Enum
+type : Enum
+stock : Stock

+sell( ) : SoldStock
+buy( ) : PurchasedStock

LoanPayment

+Account
+id
+created : Date
+amount
+status : Enum
+type : Enum
+loan : Loan

+getLoan()
+setLoan()
+getType()
+setType()

InterestEarned

+Account
+id
+created : Date
+amount
+status : Enum
+type : Enum
+interestEarned

default account

DepositAccount

+Client
+id : account no
+balance : Enum
+status : Enum
+type : Enum
+created : Date

+open(Client) : void
+close(Client) : void

LoansAccount

+Client
+id : account no
+balance : Enum
+status : Enum
+type : Enum
+created : Date
+interest

+getAllLoans() : ArrayList<Loans>
+getNotPaidLoans() : ArrayList<Loans>
+setBalance()

CheckingAccount

+Client
+id : account no
+balance : Enum
+status : Enum
+type : Enum
+created : Date

SavingAccount

+Client
+id : account no
+balance : Enum
+status : Enum
+type : Enum
+created : Date
+interest

Security account

+Client
+id : account no
+balance : Enum
+status : Enum
+type : Enum
+created : Date
+type : Enum

+getPurchasedStocks() : List<PurchasedStocks>

request

Loan

+id
+Account : LoanAccount
+amount :Money
+status : Enum
+requested : Date
+approved : Date
+interest : float
+lastPay()

+ addToDB()
+ request() : void
+ setApproved() : void
+ setStatus() :
+ applyInterests()
+payLoan(Money) : boolean

implements

Interest

+setInterest()
+getInterest()

PurchasedStock

+Account : SecutiyAccount
+Stock
+quantity
+status
+purchased :Money
+created : Date
+type : Enum

+getPrice()
+getQuantity()
+getStatus()

SoldStock

+account :SecutiyAccount
+Stock
+quantity
+status
+purchased_price :Money
+sold_price :Money
+created : Date
+type : Enum

+getPrice()
+getQuantity()
+getStatus()

DB Access CRUD operations

DataBaseConnection

getConnection()

DBPerson

DBClient

DBAccount

DBTransaction

DBTransaction

DBStocks

DBPurchasedStocks

DBSoldStocks

DBSoldStocks

DBLoans

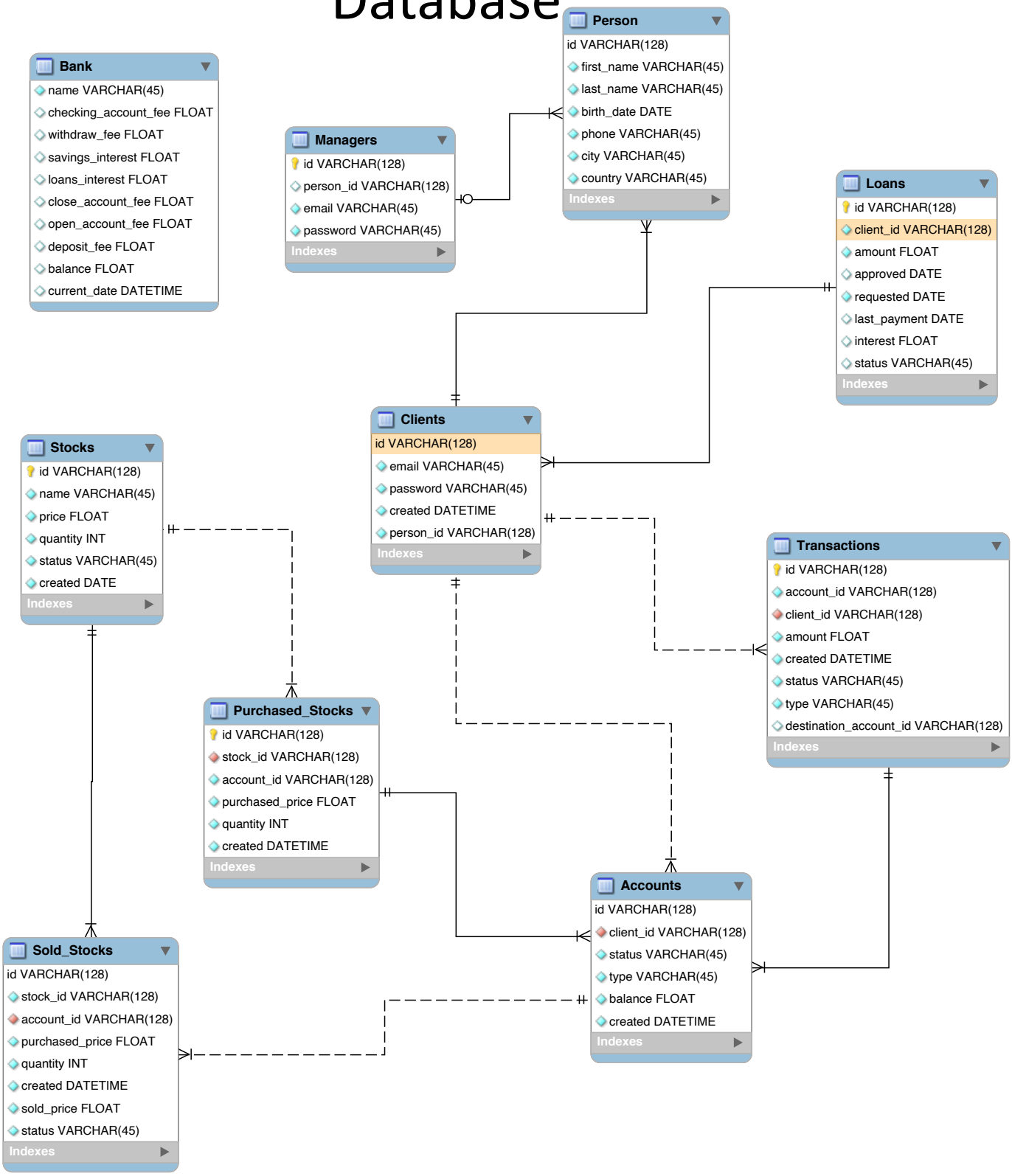
DBManager

implements

CRUDInterface

+create(T)
+retrieve(T)
+retrieveById(String)
+update (T)
+updateById(String)
+delete(T)
+deleteByID(String)

Database


Bank
name VARCHAR(45)
checking_account_fee FLOAT
withdraw_fee FLOAT
savings_interest FLOAT
loans_interest FLOAT
close_account_fee FLOAT
open_account_fee FLOAT
deposit_fee FLOAT
balance FLOAT
current_date DATETIME
Person
id VARCHAR(128)
first_name VARCHAR(45)
last_name VARCHAR(45)
birth_date DATE
phone VARCHAR(45)
city VARCHAR(45)
country VARCHAR(45)
Indexes
Managers
id VARCHAR(128)
person_id VARCHAR(128)
email VARCHAR(45)
password VARCHAR(45)
Indexes
Loans
id VARCHAR(128)
client_id VARCHAR(128)
amount FLOAT
approved DATE
requested DATE
last_payment DATE
interest FLOAT
status VARCHAR(45)
Indexes
Clients
id VARCHAR(128)
email VARCHAR(45)
password VARCHAR(45)
created DATETIME
person_id VARCHAR(128)
Indexes
Stocks
id VARCHAR(128)
name VARCHAR(45)
price FLOAT
quantity INT
status VARCHAR(45)
created DATE
Indexes
Transactions
id VARCHAR(128)
account_id VARCHAR(128)
client_id VARCHAR(128)
amount FLOAT
created DATETIME
status VARCHAR(45)
type VARCHAR(45)
destination_account_id VARCHAR(128)
Indexes
Purchased_Stocks
id VARCHAR(128)
stock_id VARCHAR(128)
account_id VARCHAR(128)
purchased_price FLOAT
quantity INT
created DATETIME
Indexes
Accounts
id VARCHAR(128)
client_id VARCHAR(128)
status VARCHAR(45)
type VARCHAR(45)
balance FLOAT
created DATETIME
Indexes
Sold_Stocks
id VARCHAR(128)
stock_id VARCHAR(128)
account_id VARCHAR(128)
purchased_price FLOAT
quantity INT
created DATETIME
sold_price FLOAT
status VARCHAR(45)
Indexes