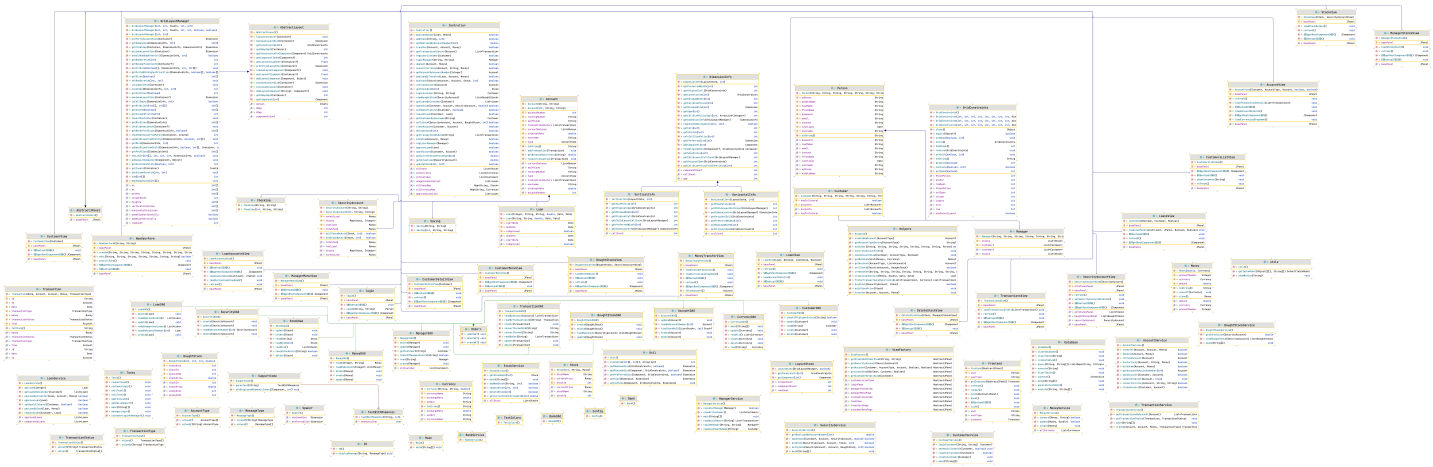
<<<<<<< HEAD

Design Document

Our overall structure is like this below (including service, DAO and also object classes):



You can also check the image in this folder to get a better view.

Project Structure

For the project structure, please check the `MVC` document in the same folder.

Object Model

Currency

	Type	Desc
currencyName	String	
symbol	String	
exchangeRate	Double	The currency rate comparing to US dollar

Money

	Type	Desc
amount	double	
currency	Currency	
accountNumber	Integer	
convert(Currency)	Money	Convert the currency into a new currency

Bank

	Type	Desc
bankName	String	
bankId	Integer	
isOpen	Boolean	
Branch	String	

Account <<abstract>>

	Type	Desc
accountNumber	Integer	
rountingNumber	String	
swiftCode	String	
type	AccountType	
currentBalance	List<Money>	
transactionHistory	List<Transaction>	
interestRate	double	
username	String	
getBalanceByCurrency(String)	double	

Saving <- Account

	Type	Desc
Saving(int, String, String)		
Saving(String, String)		

Checking <- Account

	Type	Desc
Checking(int, String, String)		
Checking(String, String)		

Loan <- Account

	Type	Desc
dueDate	Date (unix timestamp)	
startDate	Date (unix timestamp)	
isApproved	boolean	
setApproved		

Security Account

	Type	Desc
stocks	Map<Stock, int>	key is the Stock, value is the units bought for this stock. So when updating the currenct price for one stock, we can just use the object to calculate the current money of this stock. $Stock.currentPrice*unit$
realized	Money	
unrealized	Money	
totalPaid	Money	
buyStockByUnit(Stock, int)	Boolean	
sellStockByUnit(Stock, int)	Boolean	

Stock

	Type	Desc
stockId	int	
stockName	String	
currentPrice	Money	Current price of this stock per unit

BoughtStock

	Type	Desc
accountNo	int	
stockId	int	
stockUnit	int	finish stock and stock account part, including buying and selling
stockPrice	double	

Person <<abstract>>

	Type	Desc
username	String	
firstName	String	
lastName	String	
middleName	String	
email	String	
password	String	
contact	String	
address	String	
lastLogin	Date	

Manager <- Person

	Type	Desc
stocks	List<Stock>	
customers	List<Customer>	

Customer <- Person

	Type	Desc
hasCollateral	Boolean	
accounts	List<Account>	

Transact <<interface>>

	Type	Desc
makeTransaction(TransactionType, object, object, Money)	Boolean	
getTransaction(int)	Transaction	
getTransactions()	List<Transaction>	

Transaction <<abstract>>

	Type	Desc
id	int	
date	Date	
from	Account	
to	Account	
money	Money	
transactionType	TransactionType	
transactionStatus	TransacttionStatus	

TransactionType

- REGULAR_TRANSACTION
- BILL_PAY
- SERVICE_FEE
- ...

TransactionStatus

- SUCCESS
- FAILED
- PENDING

Database

User

This table includes both customers and managers

Attribute	Type	Desc
Username	Pk	
First_name	String	
Middle_name	String	
last_name	String	
Password		
email	String	
contact	String	
address	String	
is_customer	Bool	
last_login	String	
has_collateral	Bool	

Currency

Attribute	Type	Desc
account_no	int	
account_type	string	
username	string	
routing_no	int	
swift_code	int	
interest_rate	double	the currency rate comparing to USD

Bank(ATM)

Attribute	Type	Desc
bank_id	pk, int	
bank_name	string	
branch	string	
is_open	bool	

TransactionTable

Attribute	Type	Desc
account_no	int	
currency	string	
amount	double	

Account

The balance of the account can be get from the Money table, as one account can hold money of different currency

Attribute	Type	Desc
transaction_id	pk, string	
transaction_status	string	
from_account	int	
to_account	int	
transaction_type	string	
amount	double	
currency	string	
date	string	

Money

Attribute	Type	Desc
account_no	pk fk	
start_date	string	
end_date	string	
approved	int	

Loan

Attribute	Type	Desc
stock_id	int pk	
stock_name	string	
current_price	double	

Stock

Attribute	Type	Desc
bought_id	pk, int	
account_no	int	
stock_id	int	
stock_unit	int	
stock_price	double	

BoughtStock

Attribute	Type	Desc
account_no	int, pk	
realized	double	
total_paid	double	

User Interface

UI Design:

1. abstract AbstractPanel
   All main pages inherit a common abstract class called AbstractPanel. AbstractPanel extends a JPanel, and defines an abstract method called getBasePanel().
   This is used in Frontend.java so that we can switch between panels effortlessly without having to create new JFrames.
2. ViewFactory:
   A Factory class to return different views. All functions return an AbstractPanel instance.
3. Frontend.java
   A singleton class that keeps track of the state. The state here comprises of the user object and type of user (manager or customer).
   This class always returns the same instance of the frontend and is used to traverse within the app.
   back() is used to pop the panel on the top to go back to the previous screen while next() is used to traverse forward.
4. Helpers.java
   Utility functions helping in working with the functions in Controller.java, to abstract out the functions away from ui classes.
5. .form files. Form files are bound to it's classes (classes with the same name) and are used to design the ui components