OO Project Part 6 - Banking System

1. *Name:* Allison Rodenbaugh
2. *Project Description*: A Banking System where Customers can have bank accounts (checking and savings) and the Bank has an account list that keeps track of account number, owner, and balance as well as a customer list that keeps track of the customerID and their different accounts. The bank can add customers and set up their accounts, as well as set up loans for the customers and apply interest rates. The customer can deposit, withdraw, and transfer from their accounts, and view their account details.

3. *List the features that were implemented (table with ID and title).*

Functionality ID #	Requirement ID #	User Requirement
1	UC-01A	Customer shall be able to make withdrawals from their checking account.
1	UC-01B	Customer shall be able to make withdrawals from their savings account if the withdrawal is <= to current balance.
2	UC-02	Customer shall be able to make deposits to their accounts.
3	UC-03	Customer shall be able to make loan payments from their accounts.
4	UC-04	Customer shall be able to make transfers between their savings and checking accounts.
5	UC-05	Customer shall be able to view the balance of their accounts.
6	UC-06A	Bank Admin shall be able to add new customers.
6	UC-06B	The system shall generate a unique customer ID for a customer when the bank Admin adds a customer.
7	UC-07A	Bank Admin shall be able to add accounts to an existing customer.
7	UC-07B	The system can generate a unique account ID for an account when the Bank Admin adds an account for a customer.
8	UC-08A	Bank Admin shall be able to add a loan to an existing customer
8	UC-08B	The system shall generate a unique loan ID for a loan when the Bank Admin adds a loan to a customer.
9	UC-09	Bank Admin shall be able to apply interest rates to a loan account.
10	UC-10	Bank Admin shall be able to view account owner and account balance for a given accountID.
11	UC-11	Bank Admin shall be able to view loan details for a given loanID including the owner, amount, and interest rate.

12	UC-12	Bank Admin shall be able to remove an account from an existing customer.
13	UC-13	Bank Admin shall be able to remove a loan from an existing customer.
14	UC-14A	Bank Admin shall be able to remove a customer from the system if their corresponding accounts/loans are at zero balance.
14	UC-14B	The system shall remove corresponding accounts/loans if a customer is removed
15	UC-15A	The system shall cancel a withdrawal from a *Saving Account* if the account does not have enough money to cover the withdrawal.
15	UC-15B	The system shall notify the user of an "insufficient funds" error if the withdrawal was cancelled.
16	UC-16A	The system shall allow loan types of "student", "auto", "personal", and "business".
16	UC-16B	The system shall notify the user of an error if the loan type is invalid.

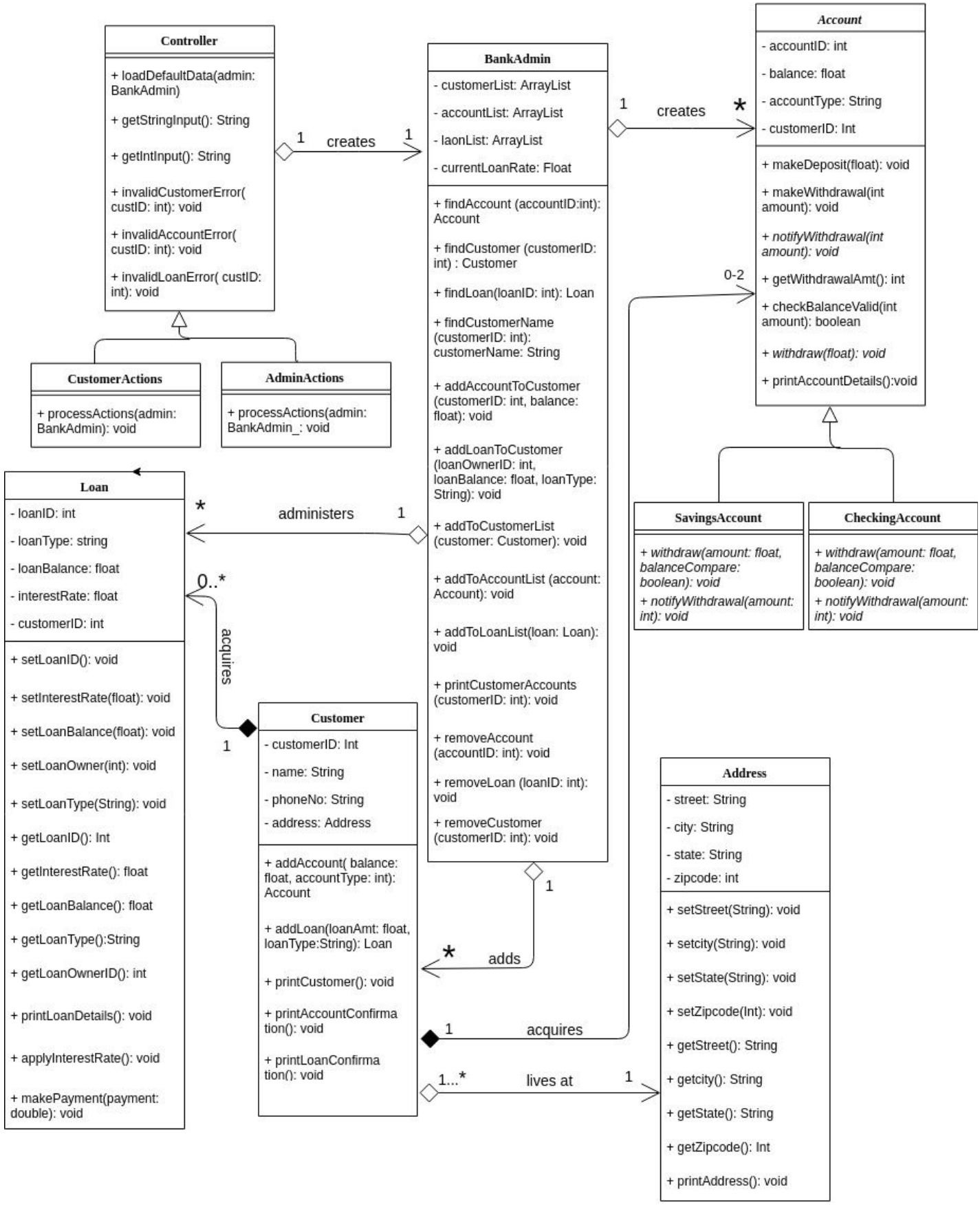
4. *List the features were not implemented (table with ID and title).*

17	UC-17A	The system shall transfer money from the customer's savings to checking and complete the withdrawal if the checking account has insufficient funds
17	UC-17B	The system shall cancel a withdrawal if both accounts combined cannot cover the withdrawal.

5. Show your final class diagram.
What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

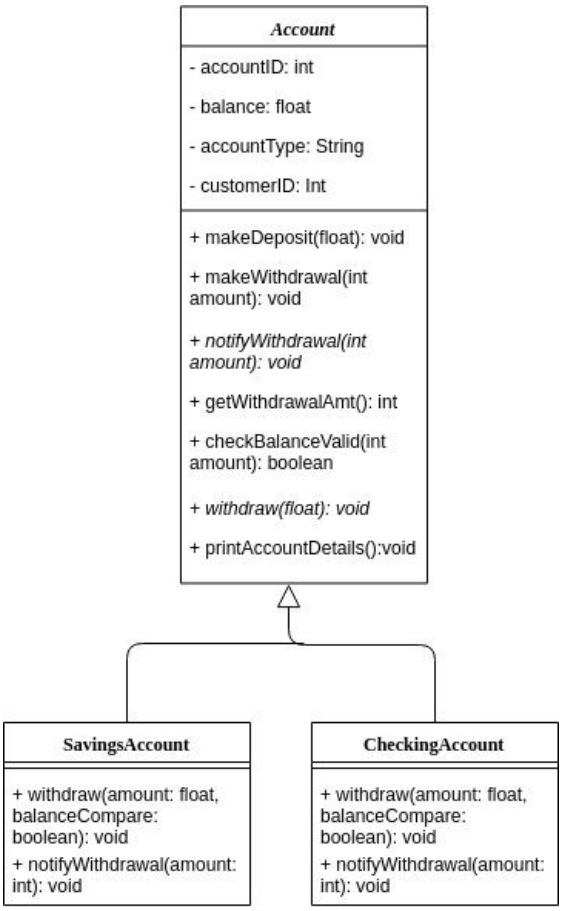
My class diagram moderately changed because I introduced helper methods in several of the classes that helped me implement the functionality. For example, I introduced helper functions to fetch the associated customer/account ID from the account and customer lists for the BankAdmin class to use so that the method using the data wasn't convoluted. This also allowed me to reuse these functions for several different purposes, rather than having repeated code in my methods. I ended up extracting repeated code from several of my methods so I could call on the helper function throughout my class. I also added a main controller class along with two more classes that help the controller gather input from the user because I didn't want my functionality classes to deal with user input. The main controller launches the application and finds out if the user is a Bank Admin or a Customer. I then have separate "sub-controllers" for both the Bank Admin and Customer roles to gather input from the respective user. The sub-controllers both take in a BankAdmin object because the bank admin stores the customer/loan/account lists and allow data retrieval/manipulation. Separating out the controller code keeps the user interaction separate from data-manipulation methods and makes the code more readable.

My class diagram also changed slightly for the three Account classes. I decided to implement the Template Design Pattern, so the Account class now includes the template method (makeWithdrawal()) and some helper methods along with abstract methods (*notifyWithdrawal()* and *withdraw()*) that the SavingsAccount and CheckingAccount classes override. **Note: I did not include the getters/setters in my diagram for the BankAdmin, Customer, and Account classes in order to conserve space in the diagram.**


Controller
+ loadDefaultData(admin: BankAdmin)
+ getStringInput(): String
+ getIntInput(): String
+ invalidCustomerError(custID: int): void
+ invalidAccountError(custID: int): void
+ invalidLoanError(custID: int): void
1
creates
1
BankAdmin
- customerList: ArrayList
- accountList: ArrayList
- laonList: ArrayList
- currentLoanRate: Float
+ findAccount (accountID:int): Account
+ findCustomer (customerID: int) : Customer
+ findLoan(loanID: int): Loan
+ findCustomerName (customerID: int): customerName: String
+ addAccountToCustomer (customerID: int, balance: float): void
+ addLoanToCustomer (loanOwnerID: int, loanBalance: float, loanType: String): void
+ addToCustomerList (customer: Customer): void
+ addToAccountList (account: Account): void
+ addToLoanList(loan: Loan): void
+ printCustomerAccounts (customerID: int): void
+ removeAccount (accountID: int): void
+ removeLoan (loanID: int): void
+ removeCustomer (customerID: int): void
1
creates
*
Account
- accountID: int
- balance: float
- accountType: String
- customerID: Int
+ makeDeposit(float): void
+ makeWithdrawal(int amount): void
+ notifyWithdrawal(int amount): void
+ getWithdrawalAmt(): int
+ checkBalanceValid(int amount): boolean
+ withdraw(float): void
+ printAccountDetails():void
0-2
CustomerActions
+ processActions(admin: BankAdmin): void
AdminActions
+ processActions(admin: BankAdmin_: void
SavingsAccount
+ withdraw(amount: float, balanceCompare: boolean): void
+ notifyWithdrawal(amount: int): void
CheckingAccount
+ withdraw(amount: float, balanceCompare: boolean): void
+ notifyWithdrawal(amount: int): void
Loan
- loanID: int
- loanType: string
- loanBalance: float
- interestRate: float
- customerID: int
+ setLoanID(): void
+ setInterestRate(float): void
+ setLoanBalance(float): void
+ setLoanOwner(int): void
+ setLoanType(String): void
+ getLoanID(): Int
+ getInterestRate(): float
+ getLoanBalance(): float
+ getLoanType():String
+ getLoanOwnerID(): int
+ printLoanDetails(): void
+ applyInterestRate(): void
+ makePayment(payment: double): void
*
administers
1
0..*
acquires
1
Customer
- customerID: Int
- name: String
- phoneNo: String
- address: Address
+ addAccount(balance: float, accountType: int): Account
+ addLoan(loanAmt: float, loanType:String): Loan
+ printCustomer(): void
+ printAccountConfirmation(): void
+ printLoanConfirmation(): void
1
*
adds
1
acquires
1...*
lives at
1
Address
- street: String
- city: String
- state: String
- zipcode: int
+ setStreet(String): void
+ setcity(String): void
+ setState(String): void
+ setZipcode(Int): void
+ getStreet(): String
+ getcity(): String
+ getState(): String
+ getZipcode(): Int
+ printAddress(): void

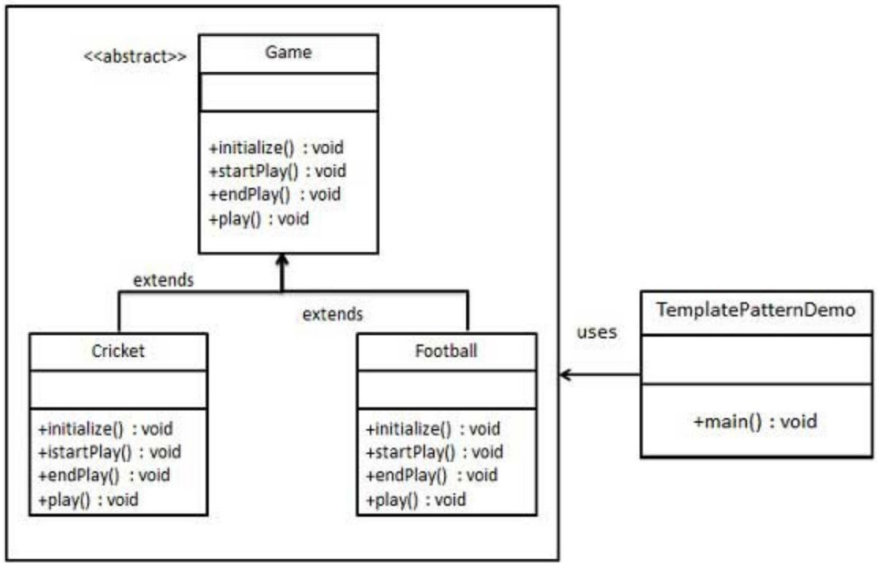
6. *For each design pattern implemented,*

- *Show the classes from your class diagram that implement each design pattern.*



- *Show the class diagram for the design pattern.*

Template Design Pattern:



- *Explain how you implemented the design pattern, why you selected that design pattern.*

I implemented the Template design pattern in my *Account* class and it's subclasses. The Gang of Four defines the use of the Template Design Pattern as "defining the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." My Account class contains the Template method which defines the structure of the makeWithdrawal() algorithm. The Savings and Checking accounts inherit from the Account class and override certain steps of the makeWithdrawal() method as per the functionality requirements.

Since I wanted the Savings and Checking accounts to compute withdrawals differently, I chose to use a template class because all of the other functionality would be the same. This way, I can reduce duplicate code and simply override the methods that need to treat the data differently. The steps that I outlined in the makeWithdrawal() template method in my Account class are:

1. *notifyWithdrawal()*: notify the user of how Saving and Checking accounts treat withdrawals differently.
2. getWithdrawalAmount(): get the user input for how much money they wish to withdraw from the account.
3. checkBalanceValid(): check if their account has sufficient funds to complete the withdrawal.
4. *withdraw()*: complete or deny the withdrawal based on subclass specifications.

My user requirements depict that Checking Account balances are allowed to be negative, whereas Saving Account balances are not. Thus, withdrawals need to be treated differently for the different account types, which is why I implemented the Template design pattern.

7. *What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?*

I have learned that drawing up class diagrams and user requirements helps significantly when creating the classes and beginning to write the code. If I hadn't created class diagrams, I would've had to refactor my code significantly more to keep it all organized. The class diagrams and user requirements forced me to thing through the whole application and plan out an organized, effective way to implement the functionality. Having class diagrams also allowed me to lay out all of my classes and the functions that I think I need before writing any functional code. From there, I was able to implement functionality one step at a time, rather than trying to think through all the moving parts I may encounter later.

Creating a system while using well thought-out diagrams is invaluable to the system's organization. Implementing a design pattern was a great learning experience and allowed me to make my system efficient and easy to understand. Going back to my user requirements throughout implementation of my system allowed me to stay on track with completing the entire project. It was very satisfying to scroll down my user requirement list and check off all of the functionality that I implemented. Overall, designing and drawing up class diagrams and user requirements for a system prior to writing *any* code allowed me to create a well-designed, organized product that I can be proud of.