**Banking System Programmer’s Manual**

Overview

This manual is designed to give software engineers a solid foundation on the structure and functionality of the banking system program. You do not need to read it all the way through and may use it merely as a reference if desired. There are 4 main sections to the manual, “Design Diagrams”, “Class Diagrams”, “Architecture Diagram”, and “Functions and Main Drivers”.

Before diving into the specifics of the architecture, it is important to know some basic details about the program. First, the program has a very simple tech stack. It utilizes Kotlin for the frontend interface and Java for the backend functionality. There is also a very simple method for utilizing a database. The method is to simply use CRUD operations on “records” recognized in .csv files. Each .csv file can be viewed as if it were a table in an SQL database. This allows us to not actually have to run any “server” for the database and instead, run the program on an as needed basis before saving the information to .csv files and ultimately shutting down until the next use.

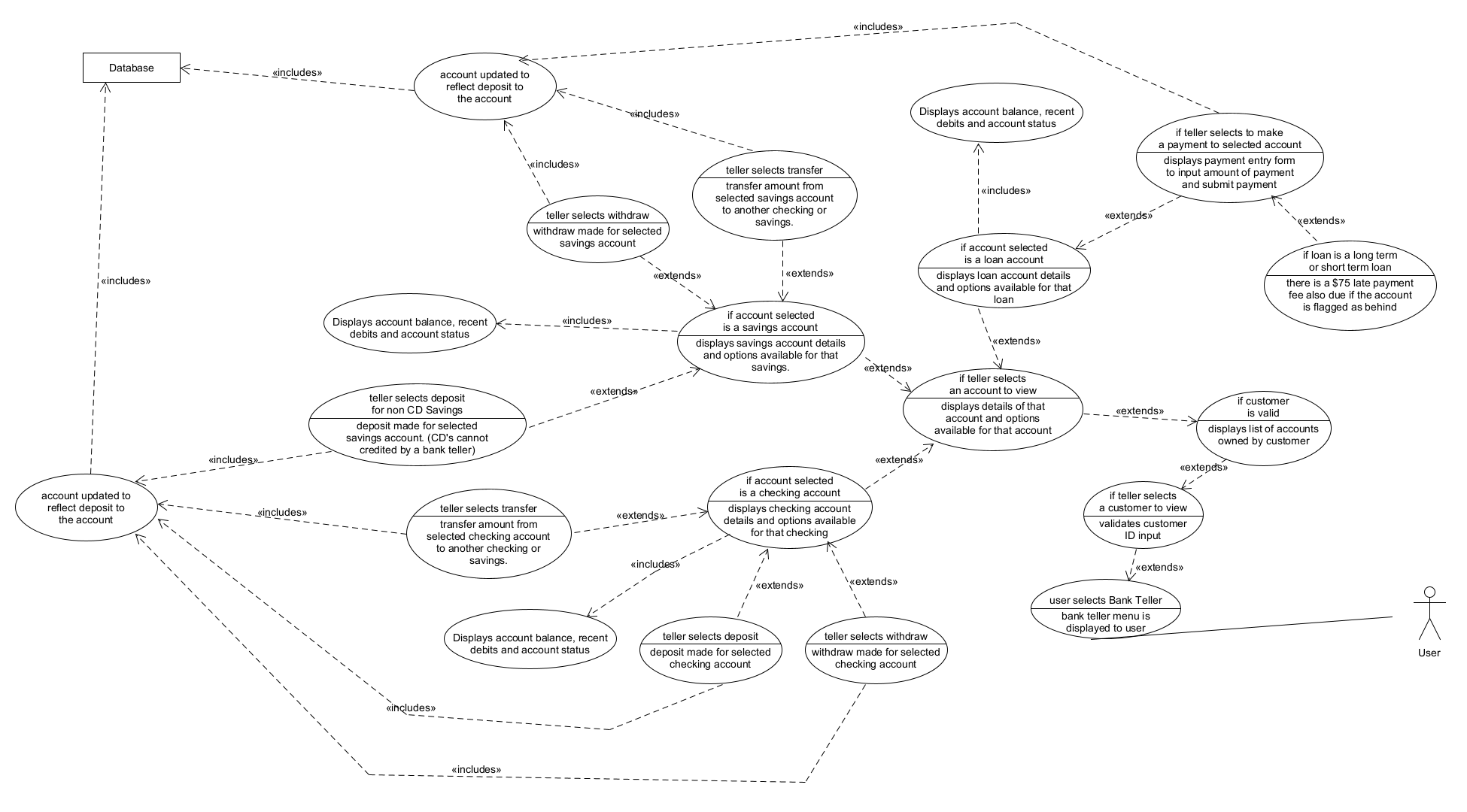
Lastly, it would be helpful to have an understanding of the basic operations and initial interfaces of the program and the purpose from a user perspective. The purpose of this program is to mock a banking system application that maintains user accounts and transactions at a bank. The application is built for 3 types of users, a customer, bank teller, and bank manager. The customer can manage their account with limited privileges, the bank teller can lookup accounts and make adjustments, and the bank manager can manage customer accounts as well as execute administrative tasks. Hopefully this overview will give enough context to adequately understand the following information.

Design Diagrams

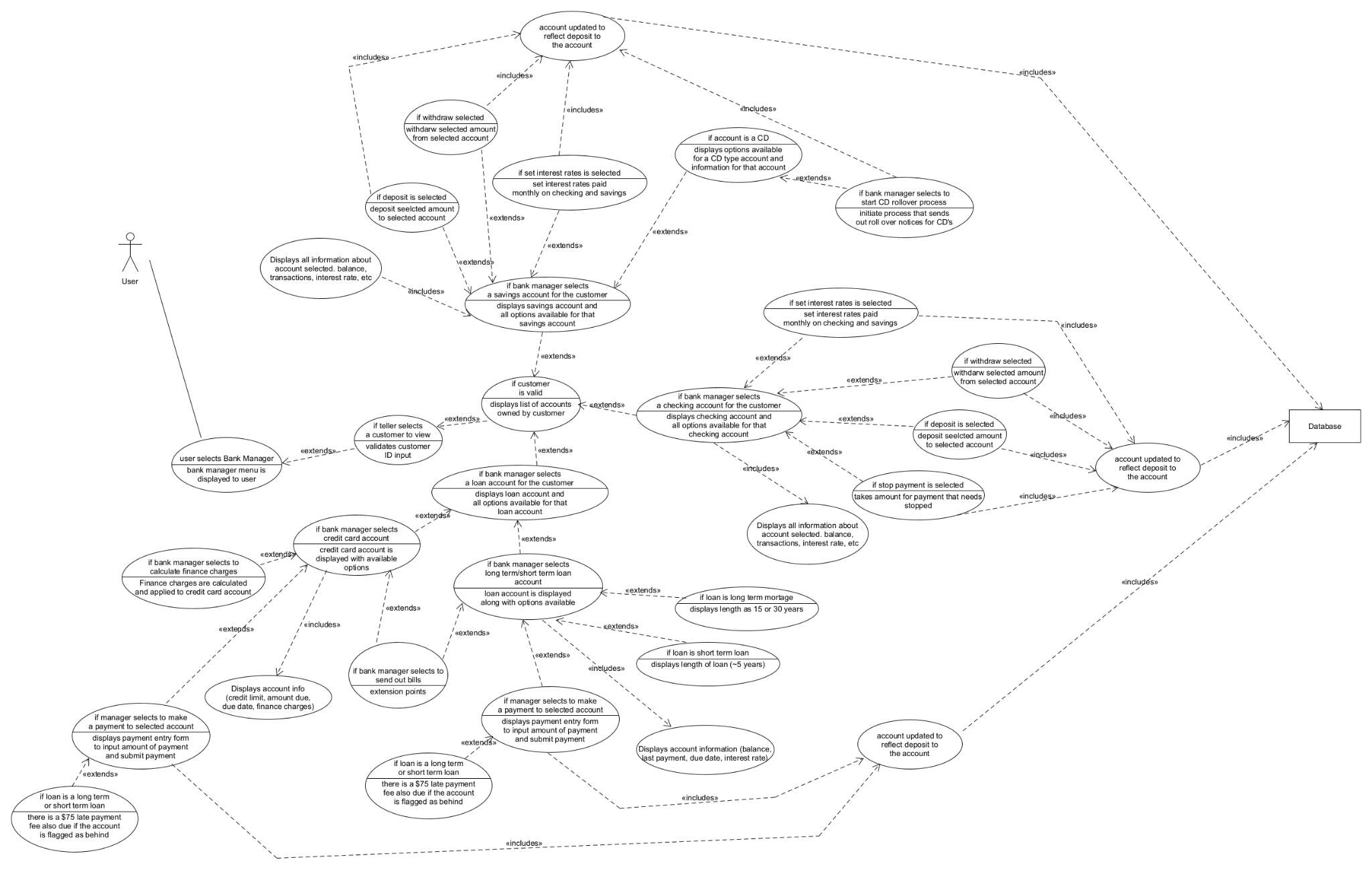
Below are 3 design diagrams that may be utilized to conceptualize the process of the interfaces and functionality of the program. First there is a use case diagram, then an activity diagram, and lastly a set of sequence diagrams detailing both the user repositories as well as the user data access objects.

*Use Case Diagram*

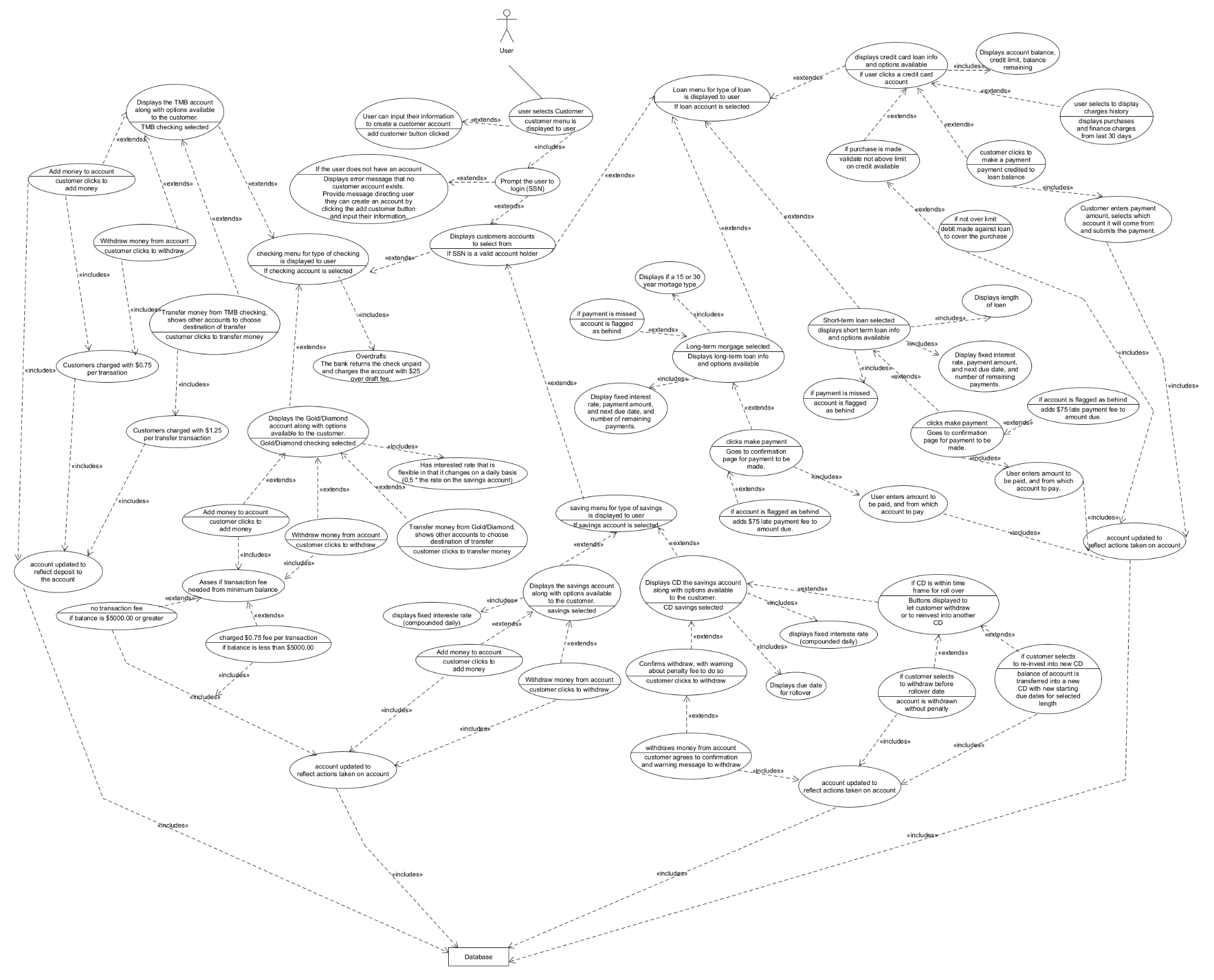
*Bank Teller*



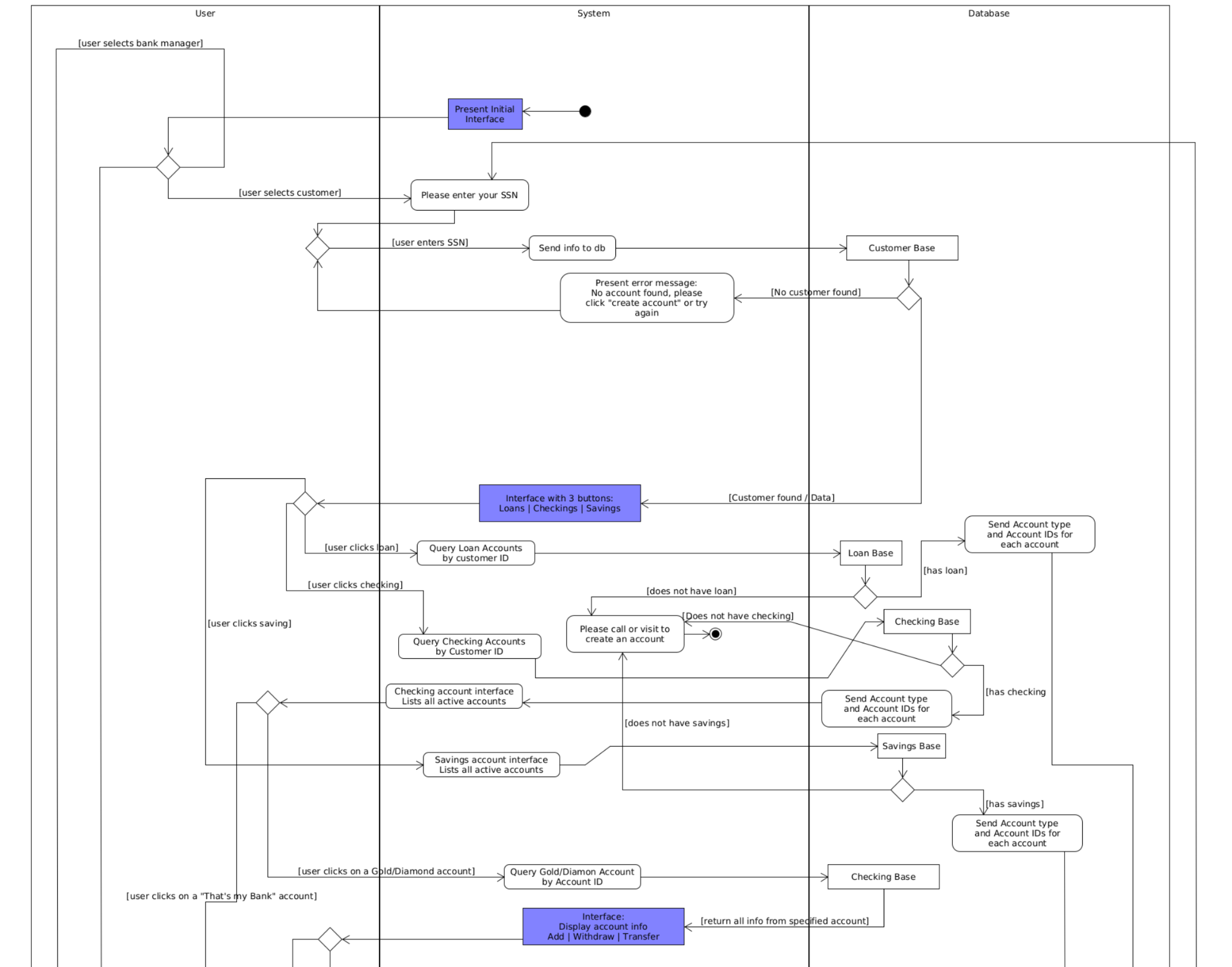
*Bank Manager*

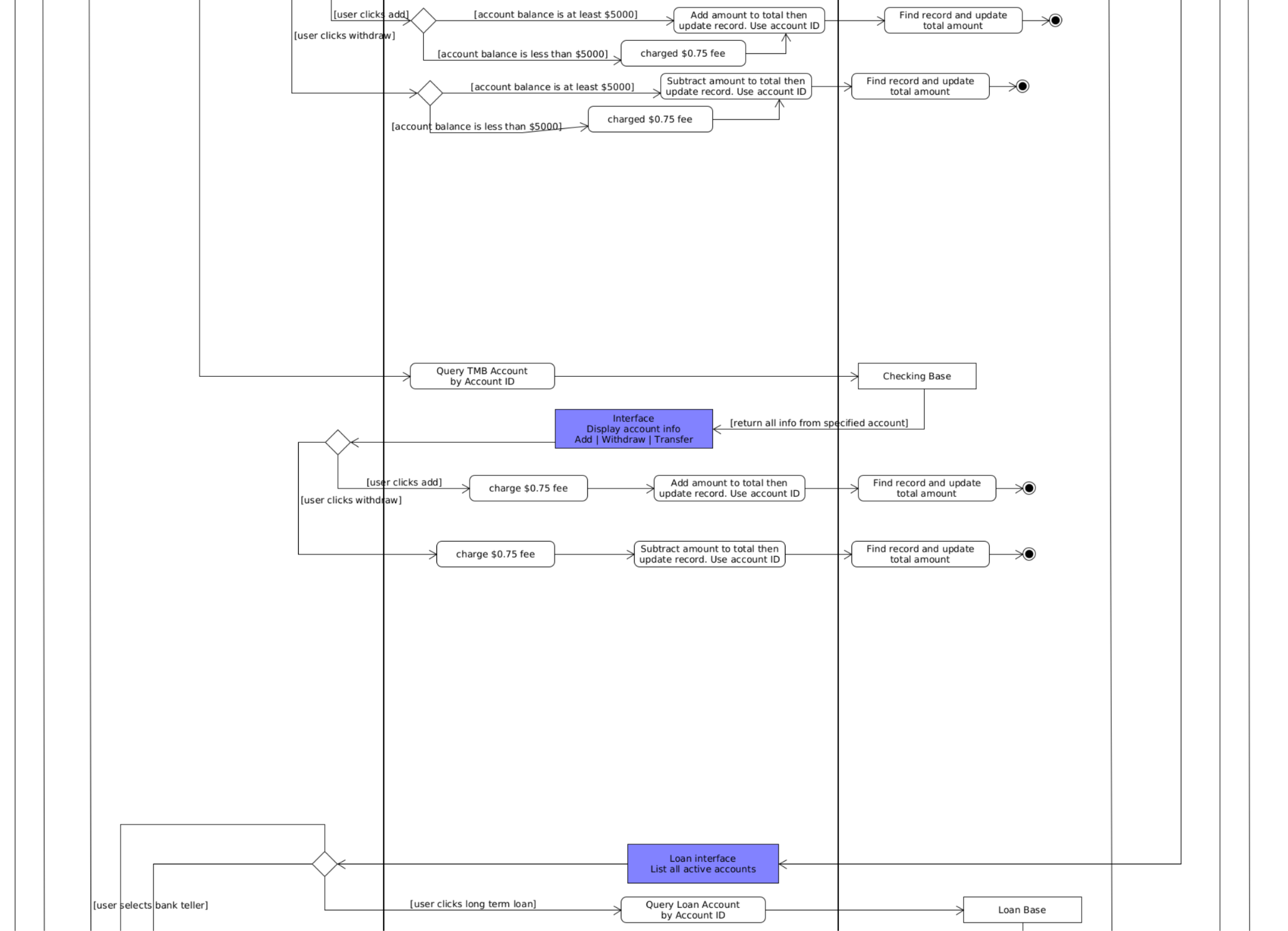


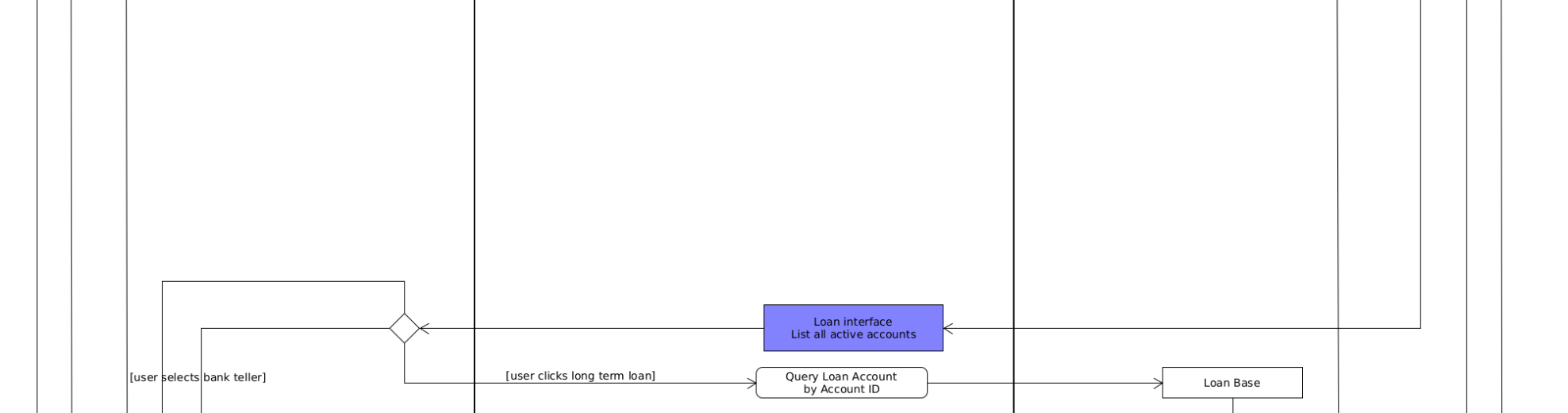
*Customer*

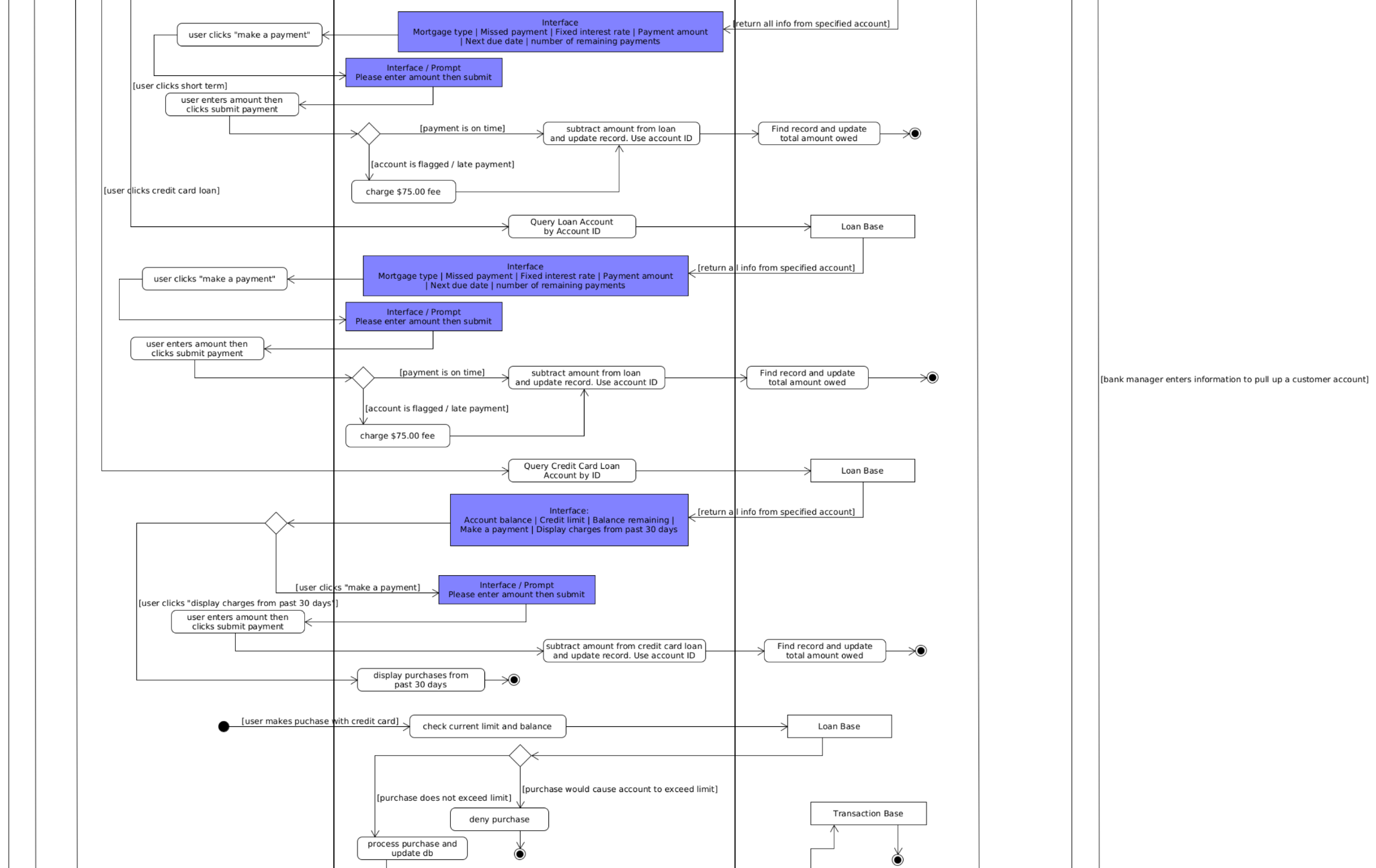


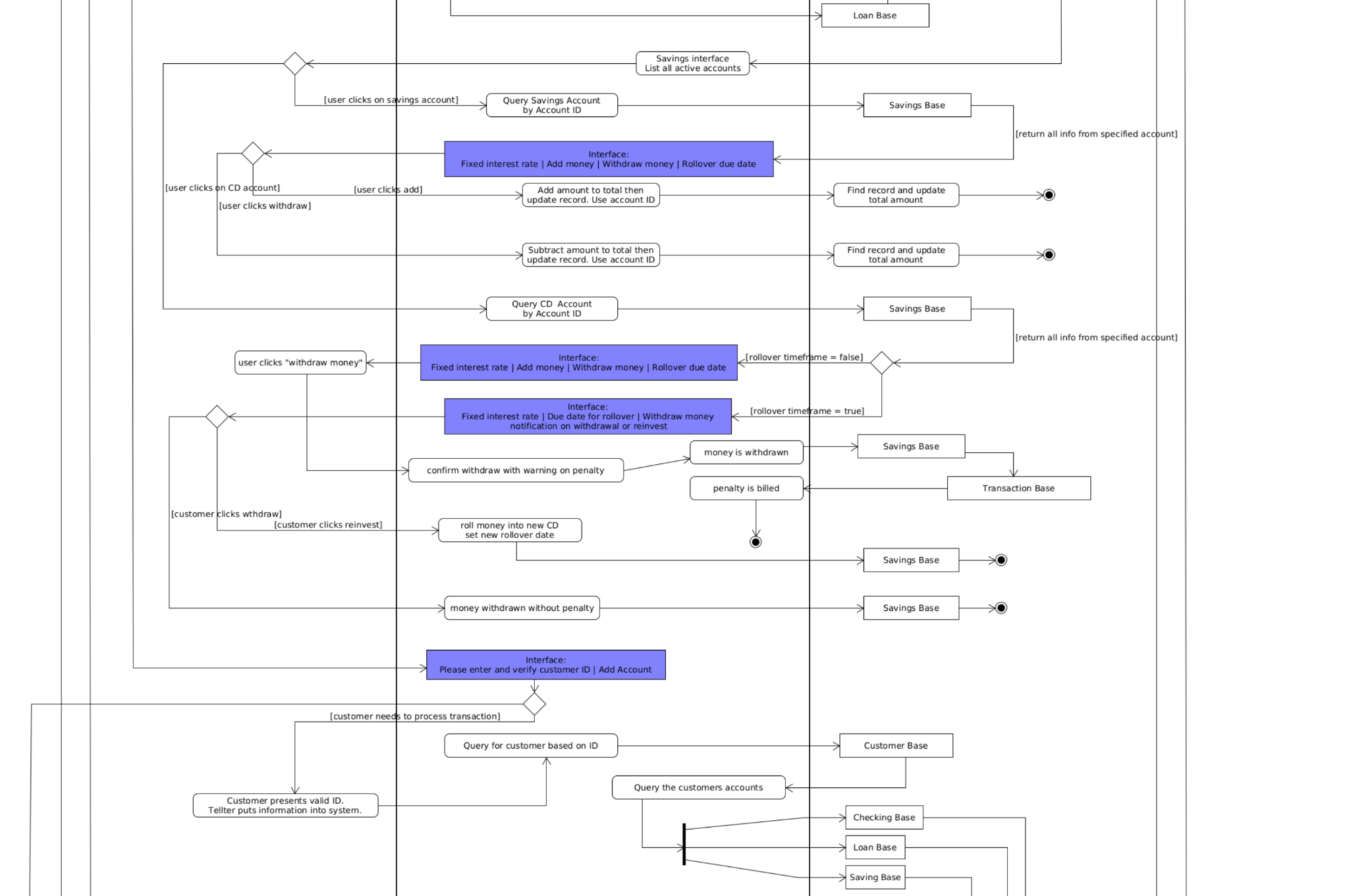
*Activity Diagram*

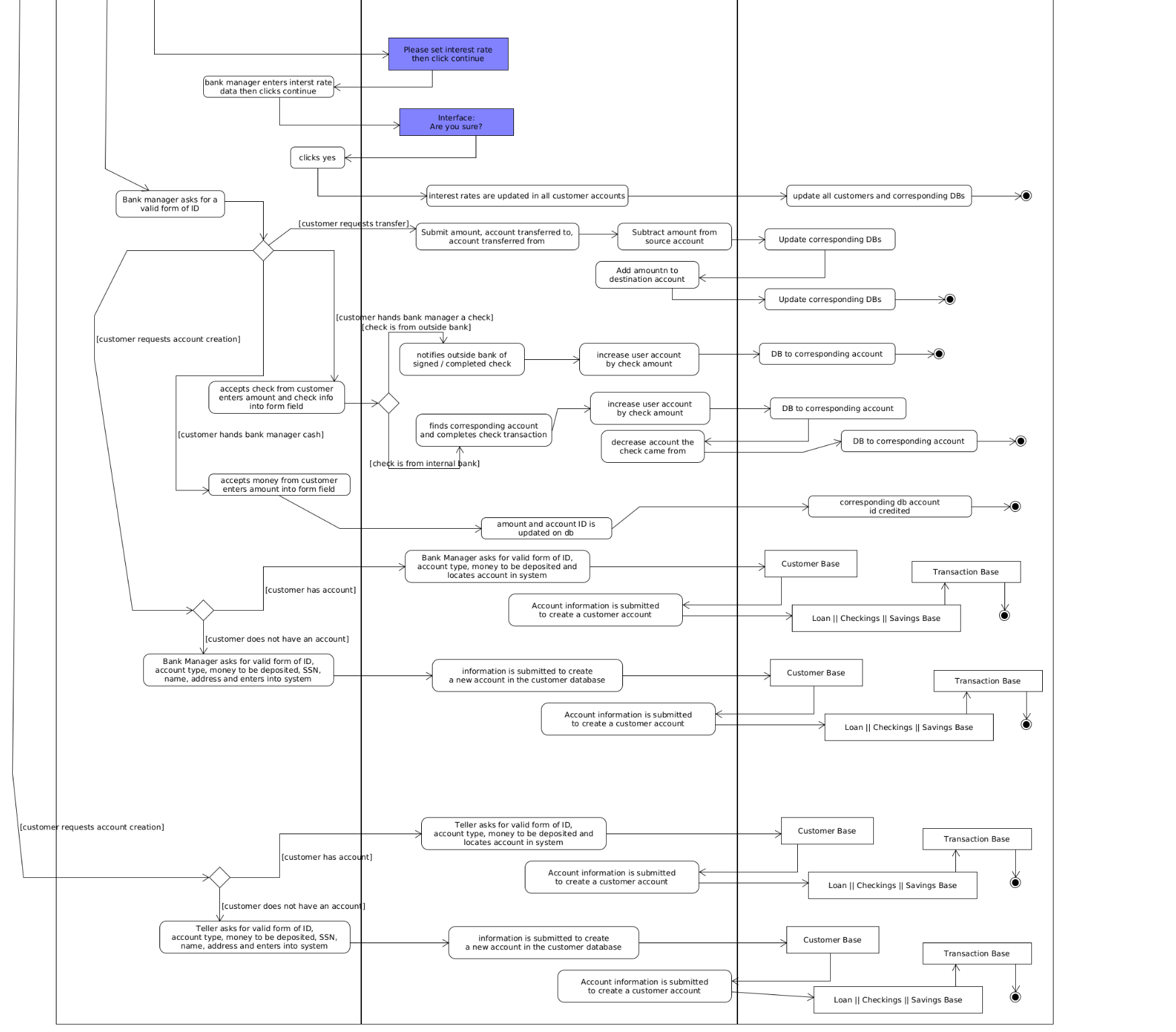
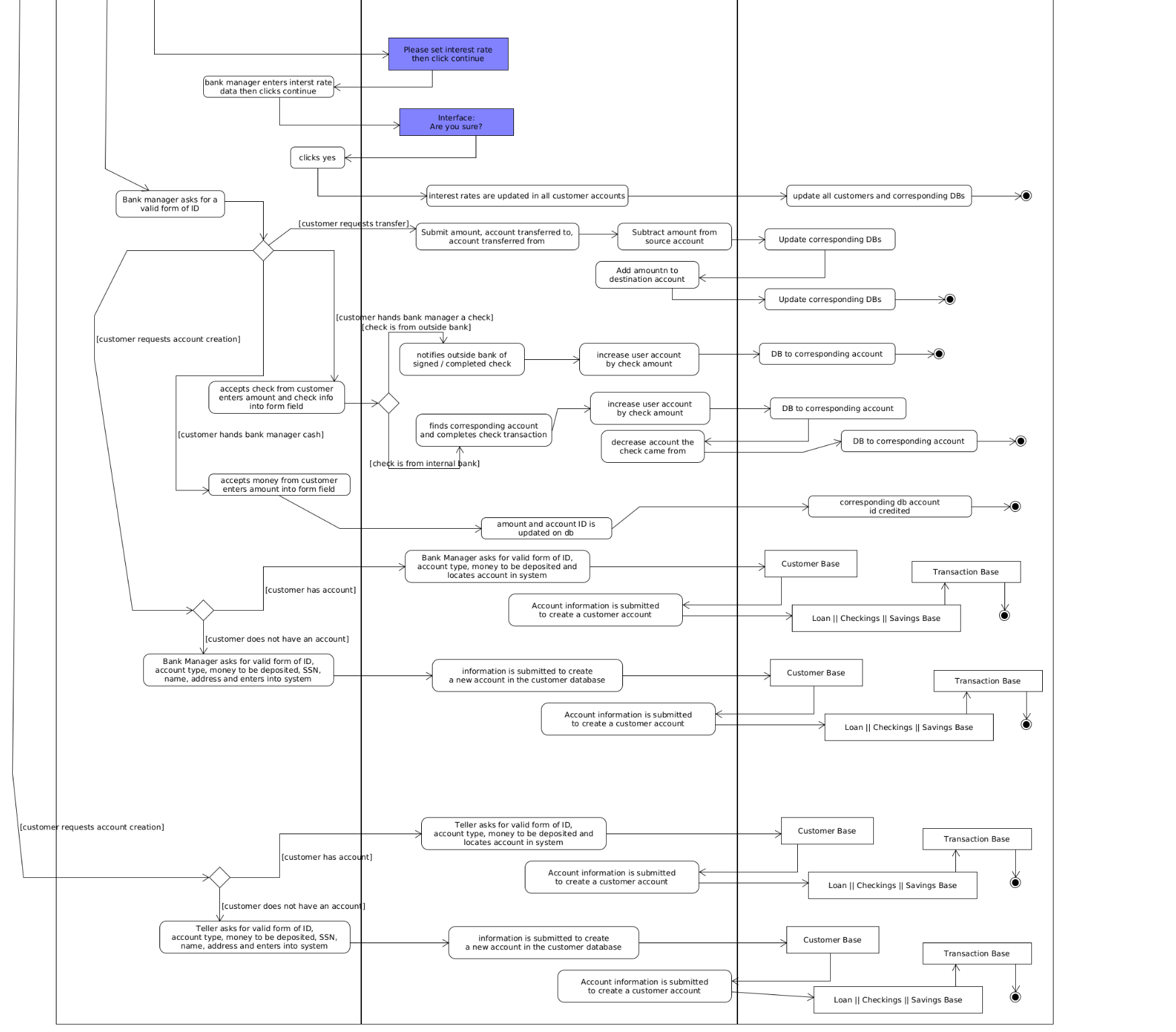
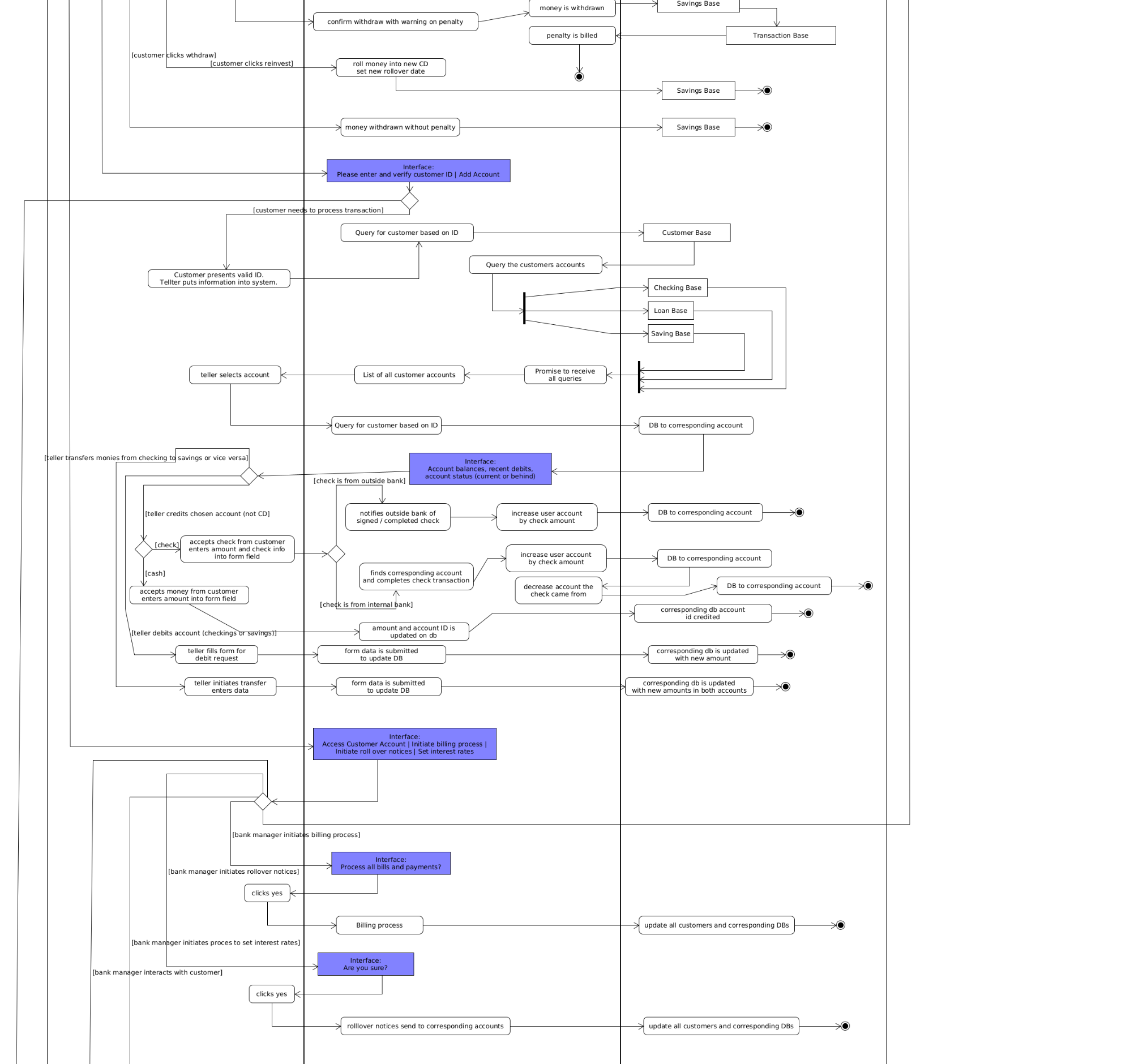


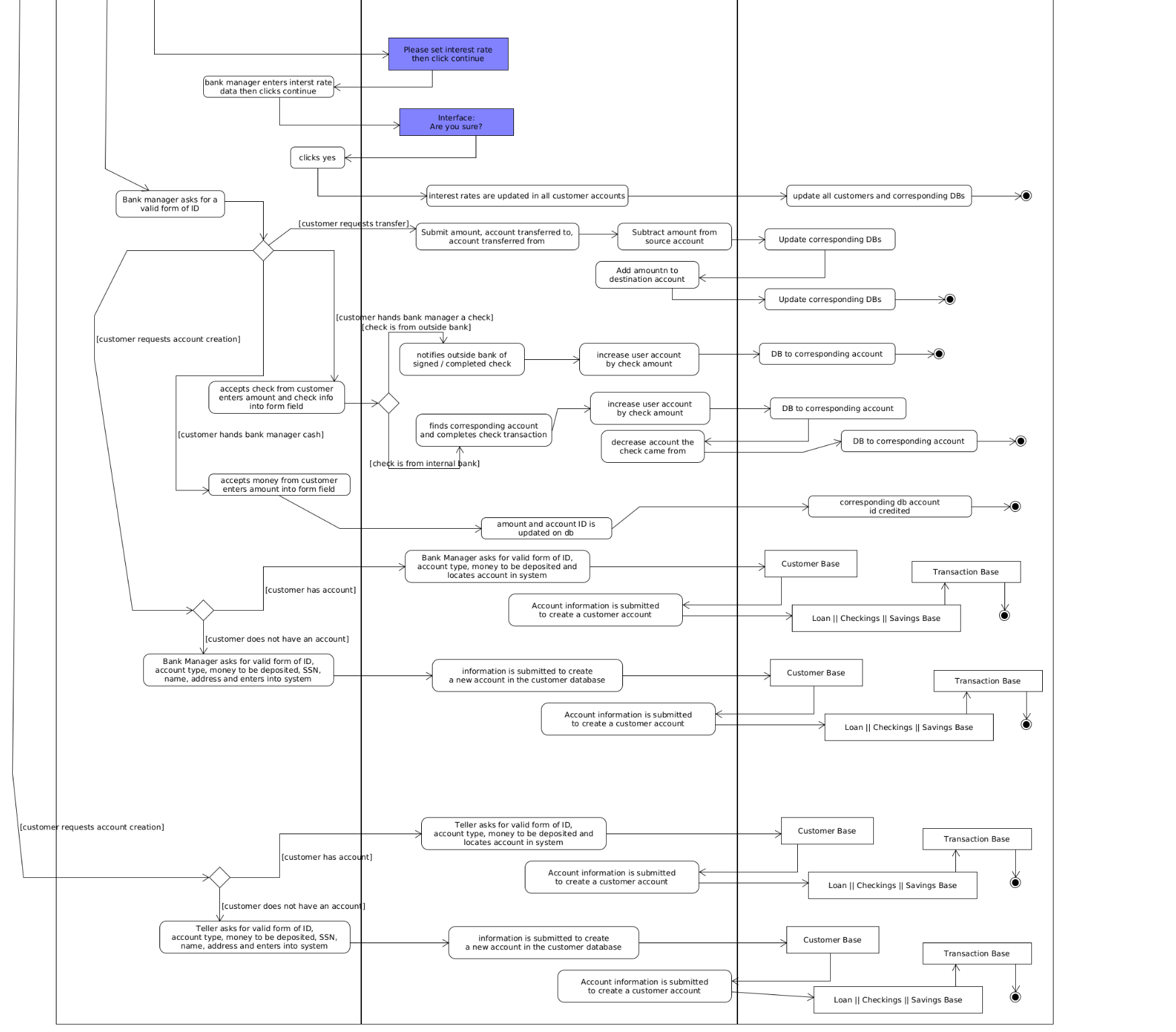






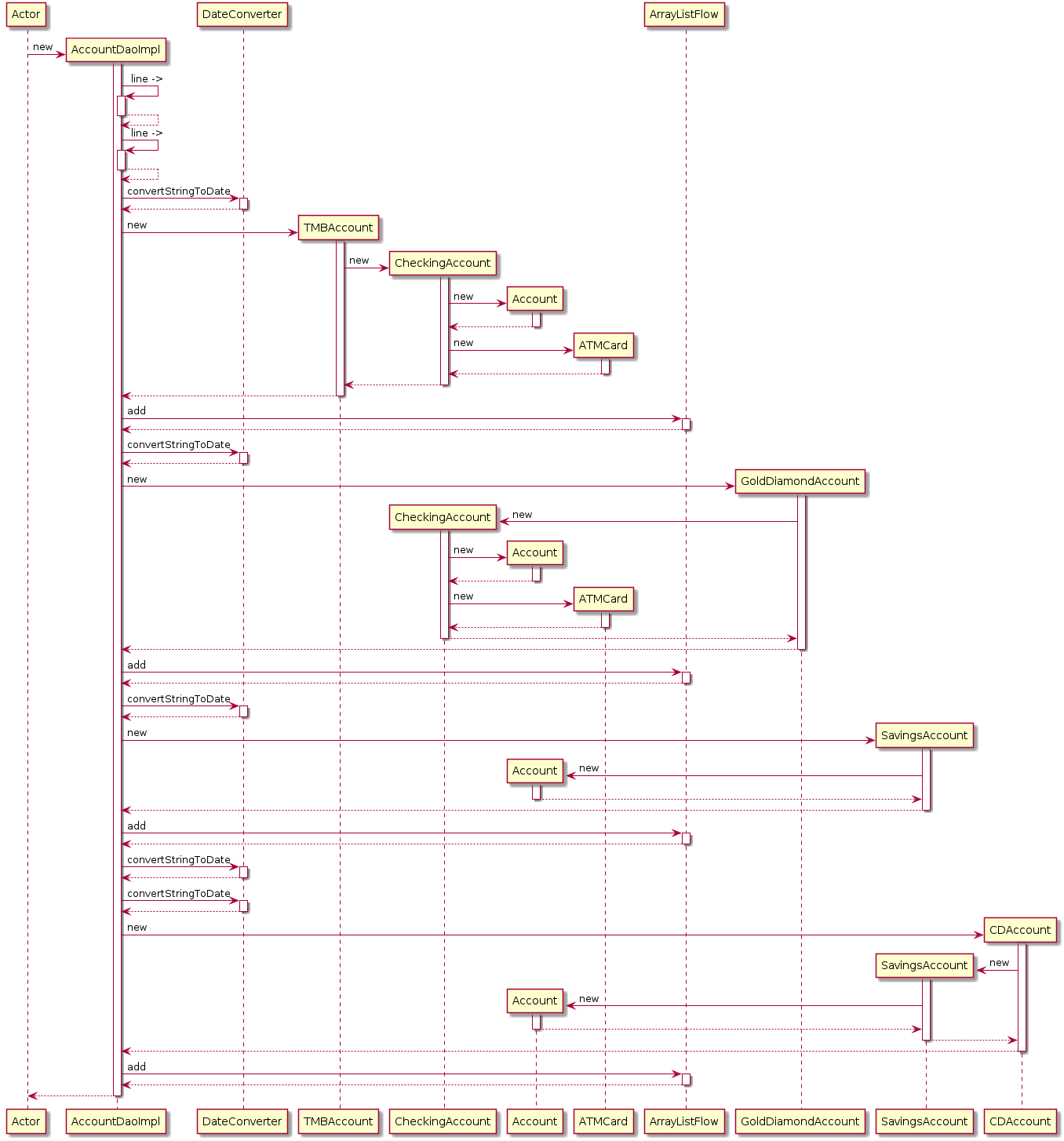




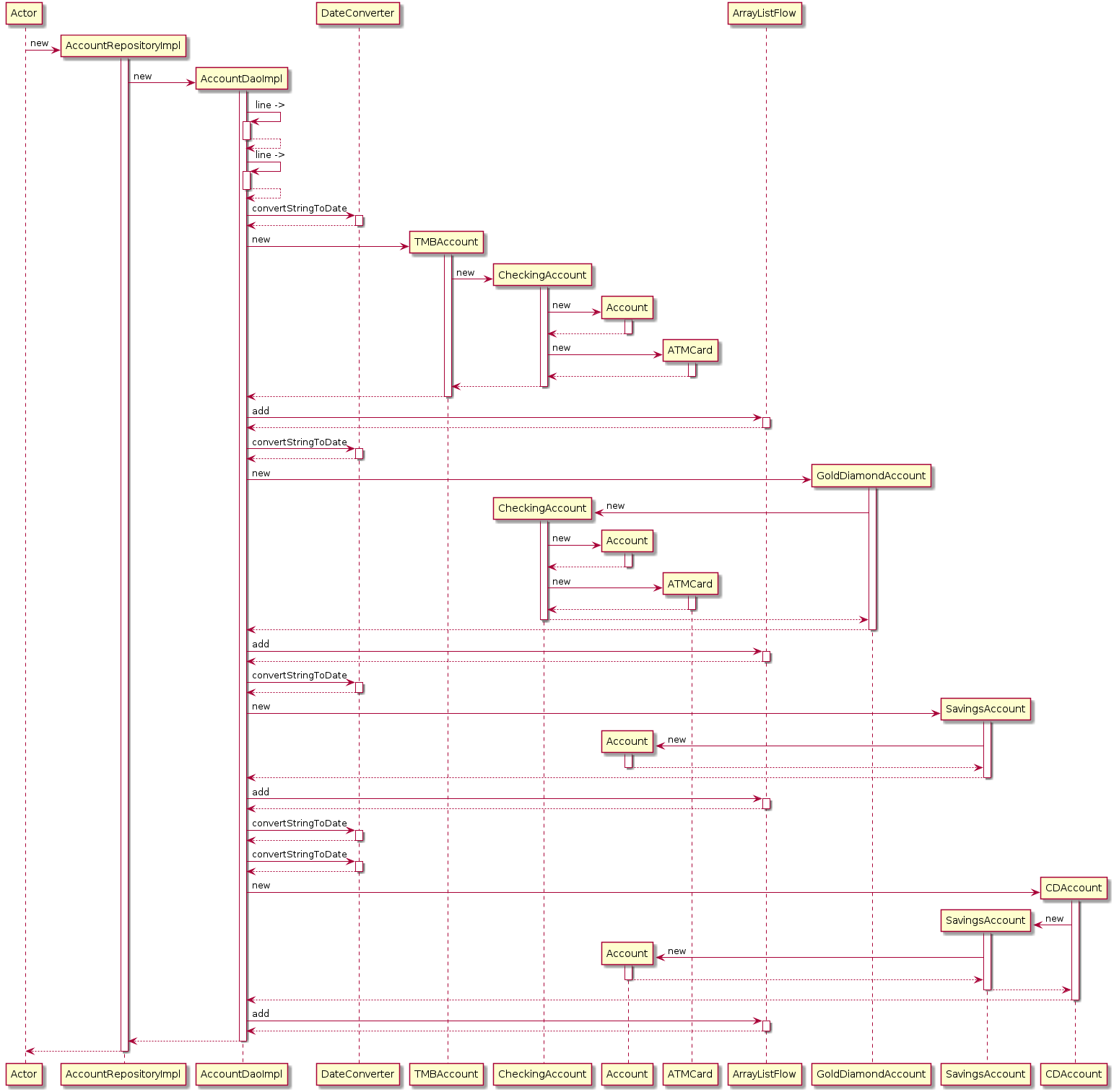


*Sequence Diagram*

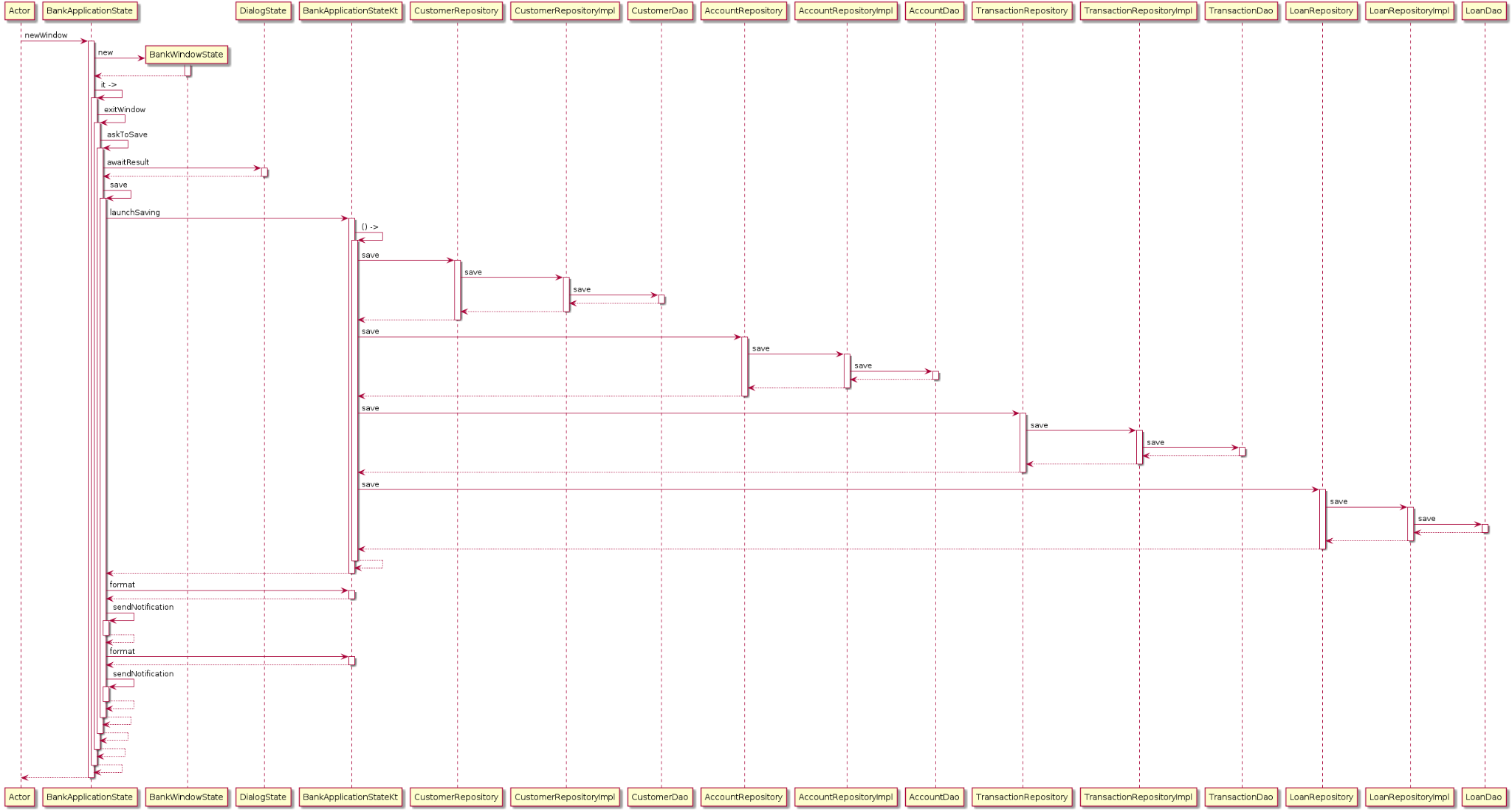
Account Dao Implementation

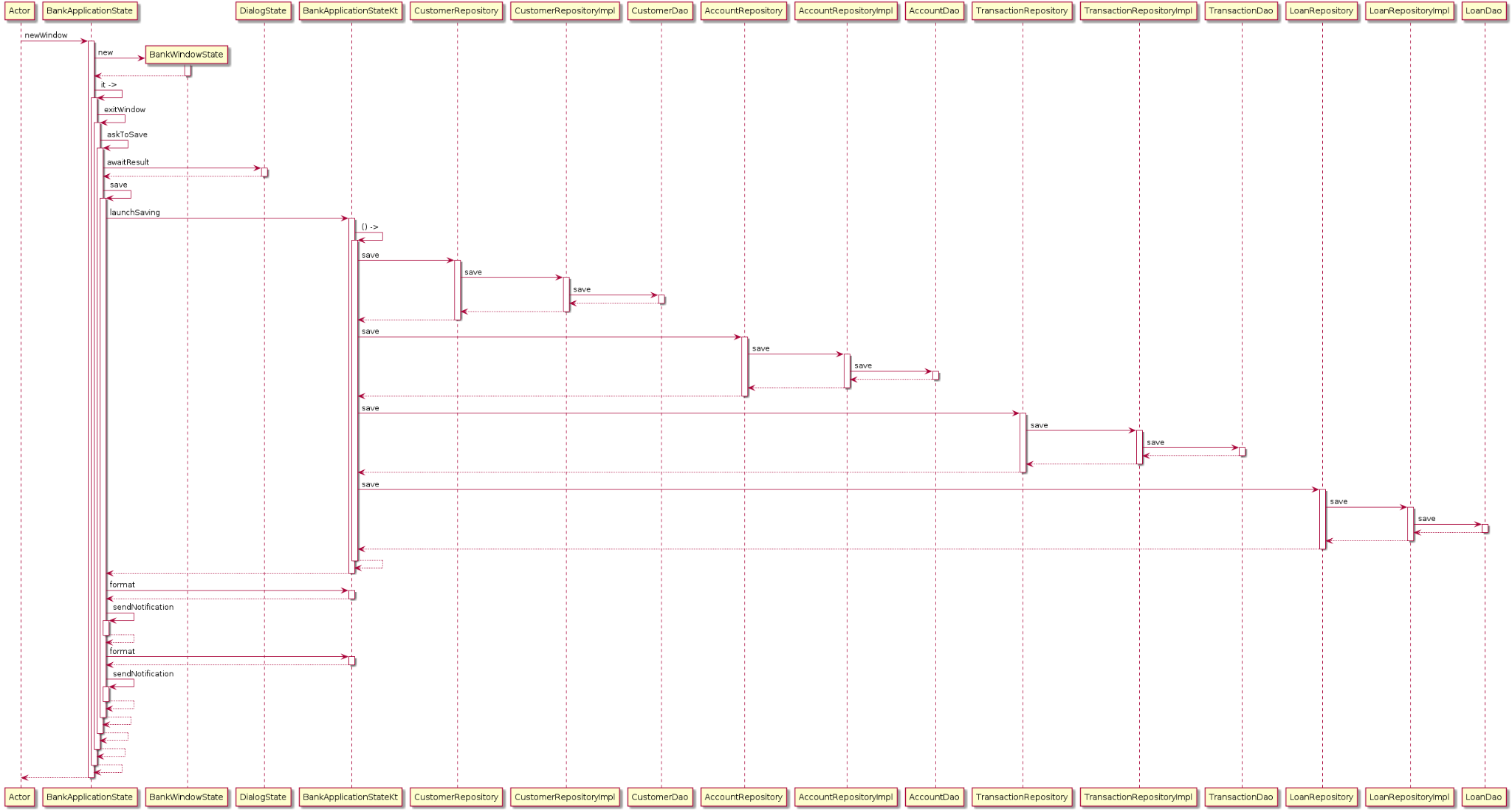


Account Repository Implementation



Bank Application State

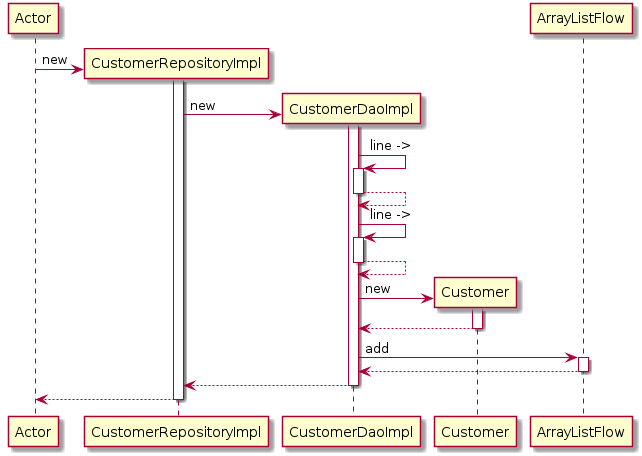




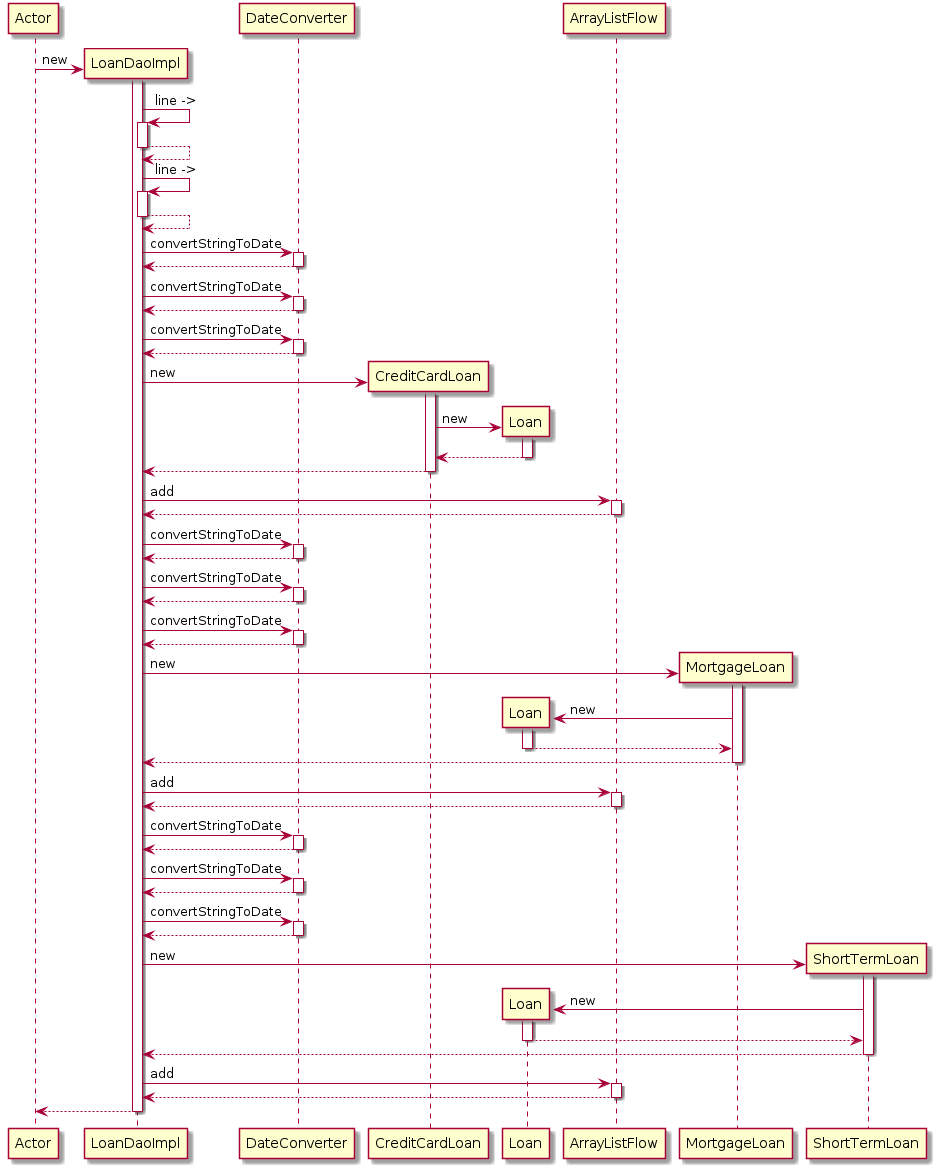
Customer Dao Implementation



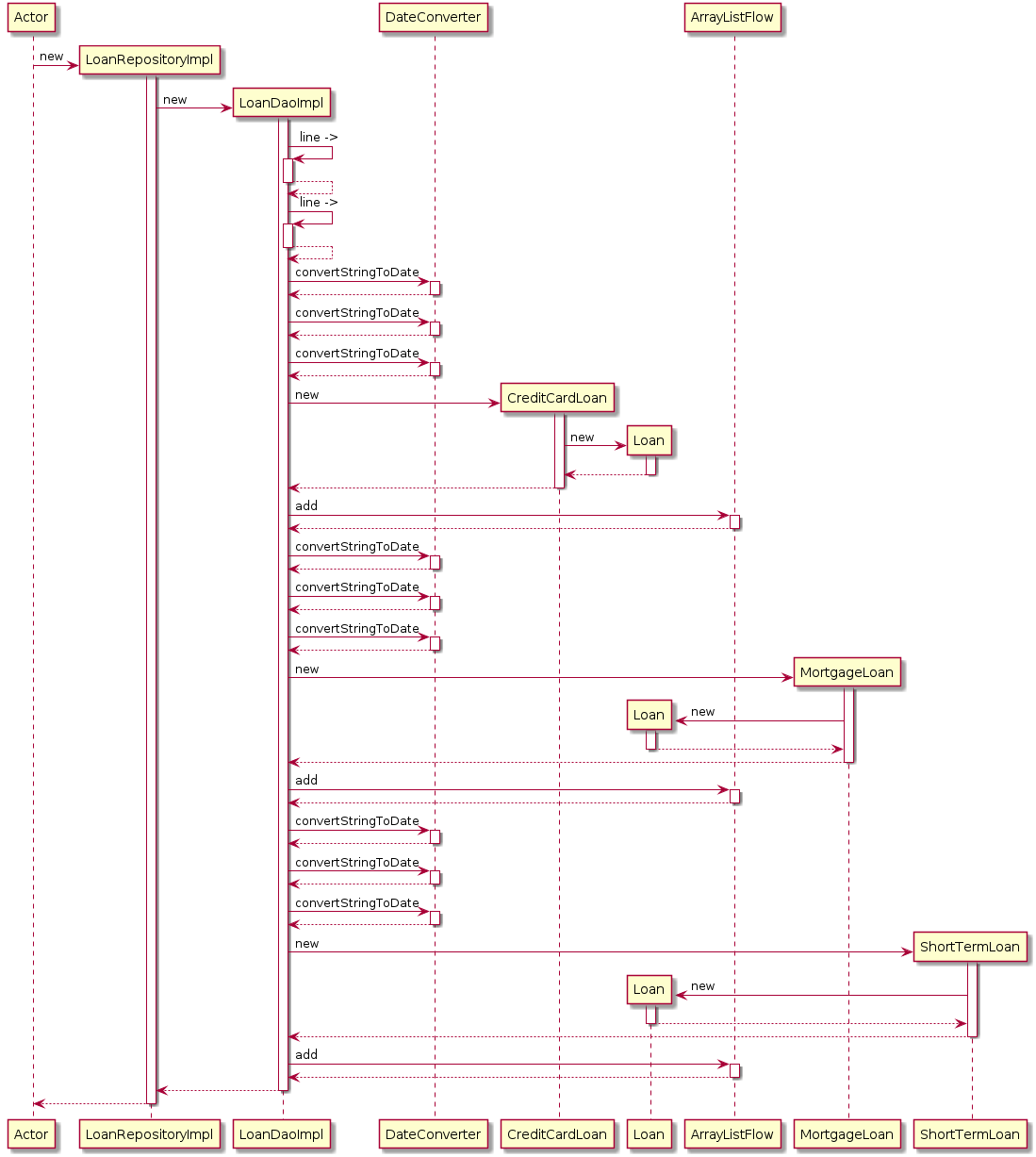
Customer Repository Implementation



Loan Dao Implementation



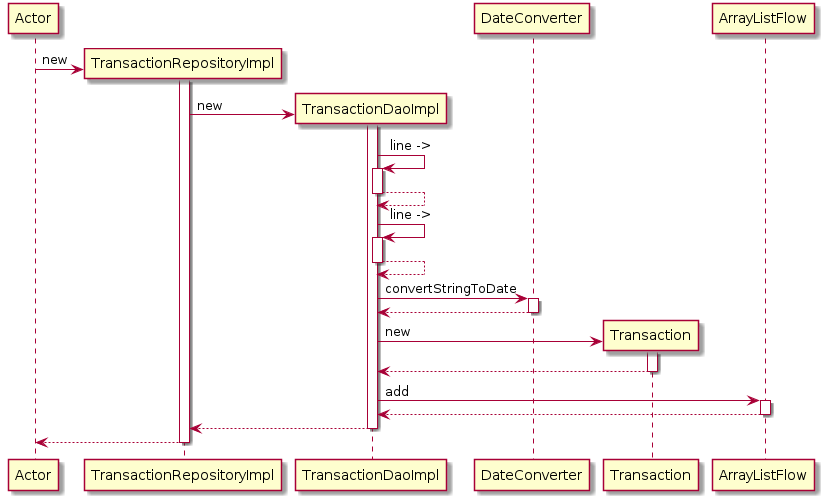
Loan Repository Implementation



Transaction Dao Implementation

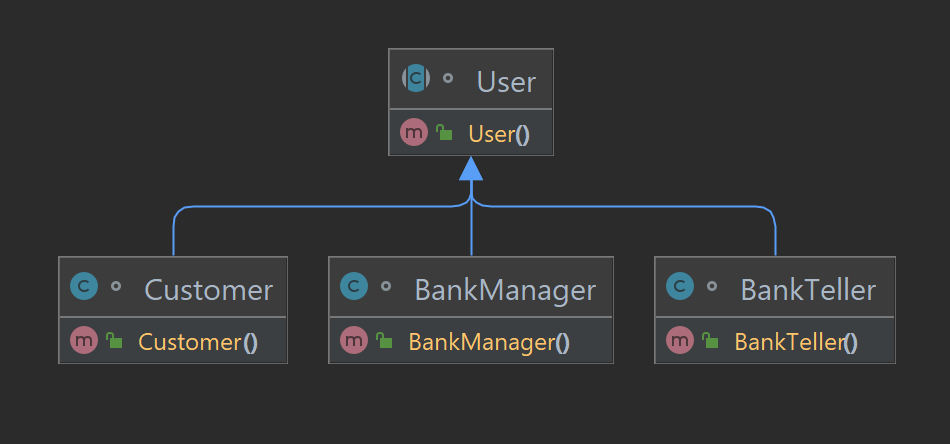


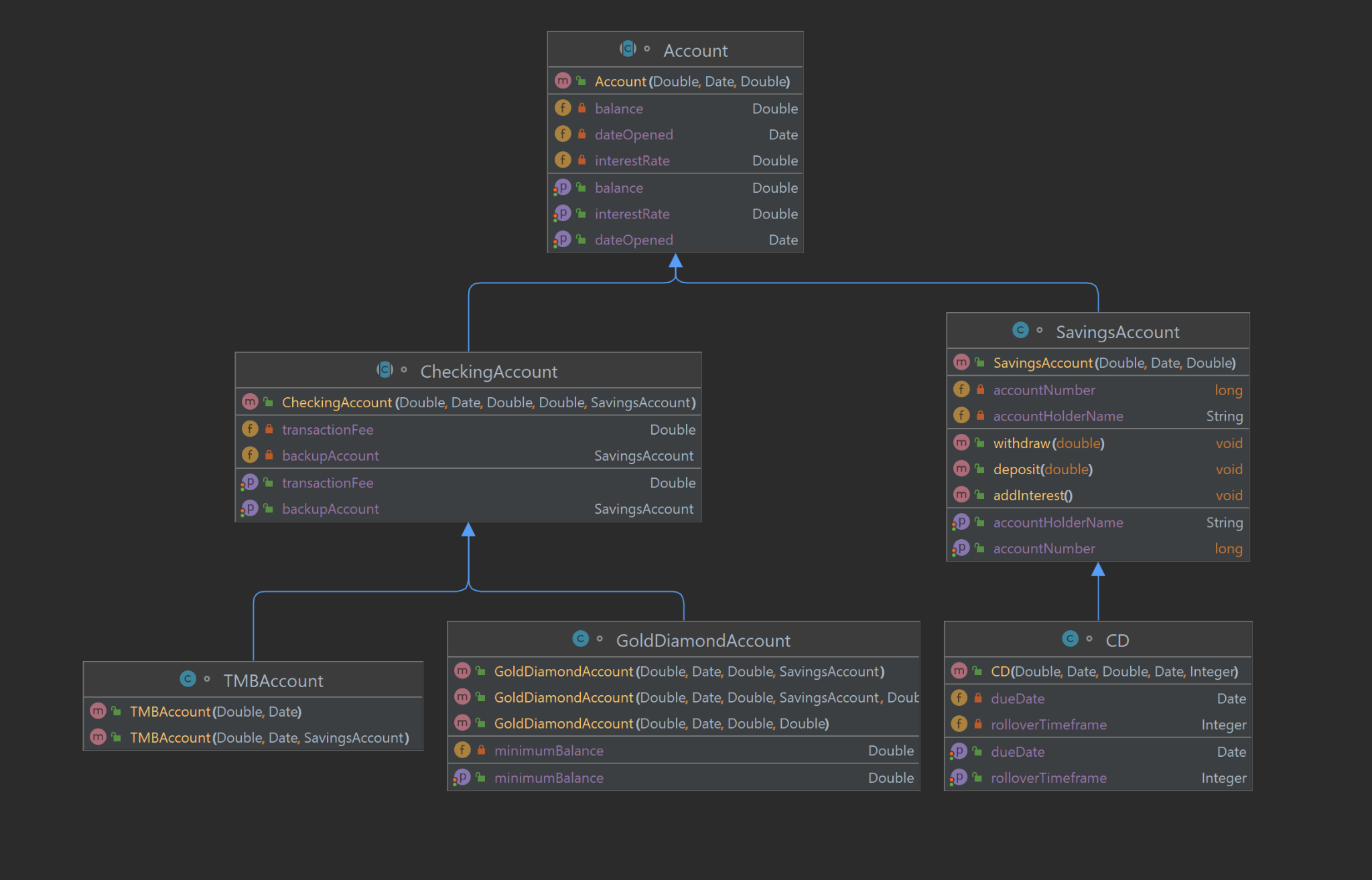
Transaction Repository Implementation

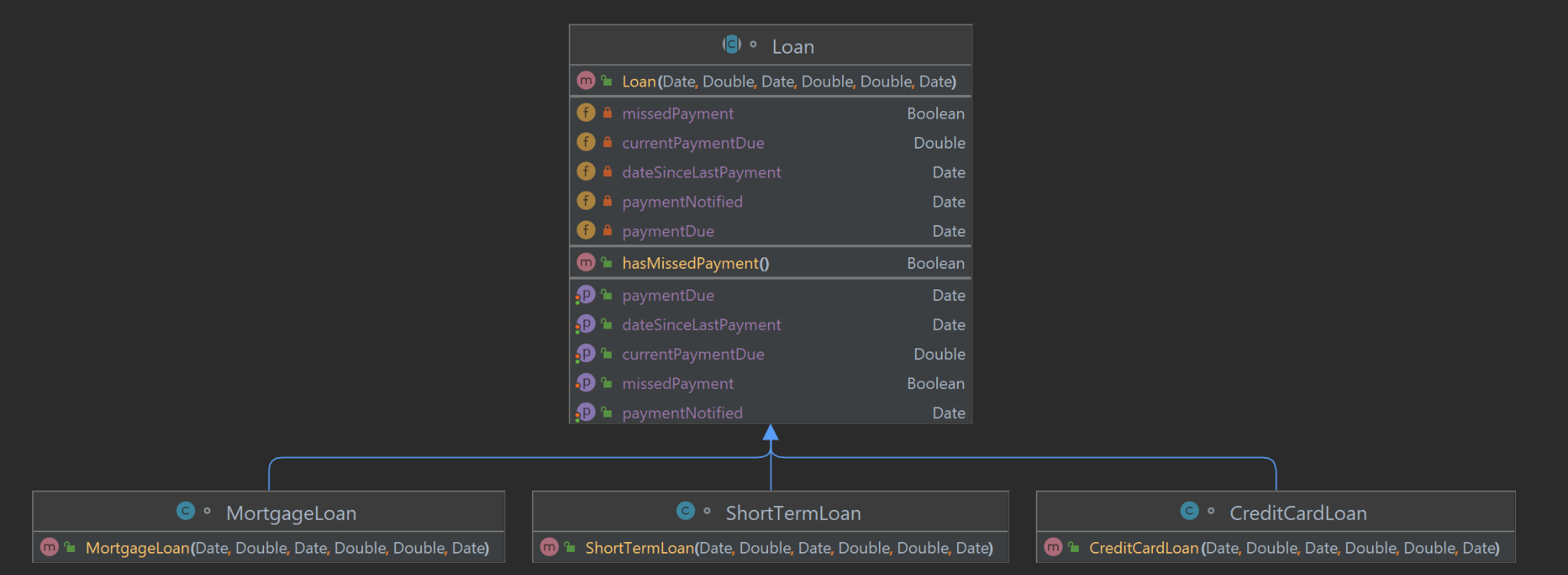


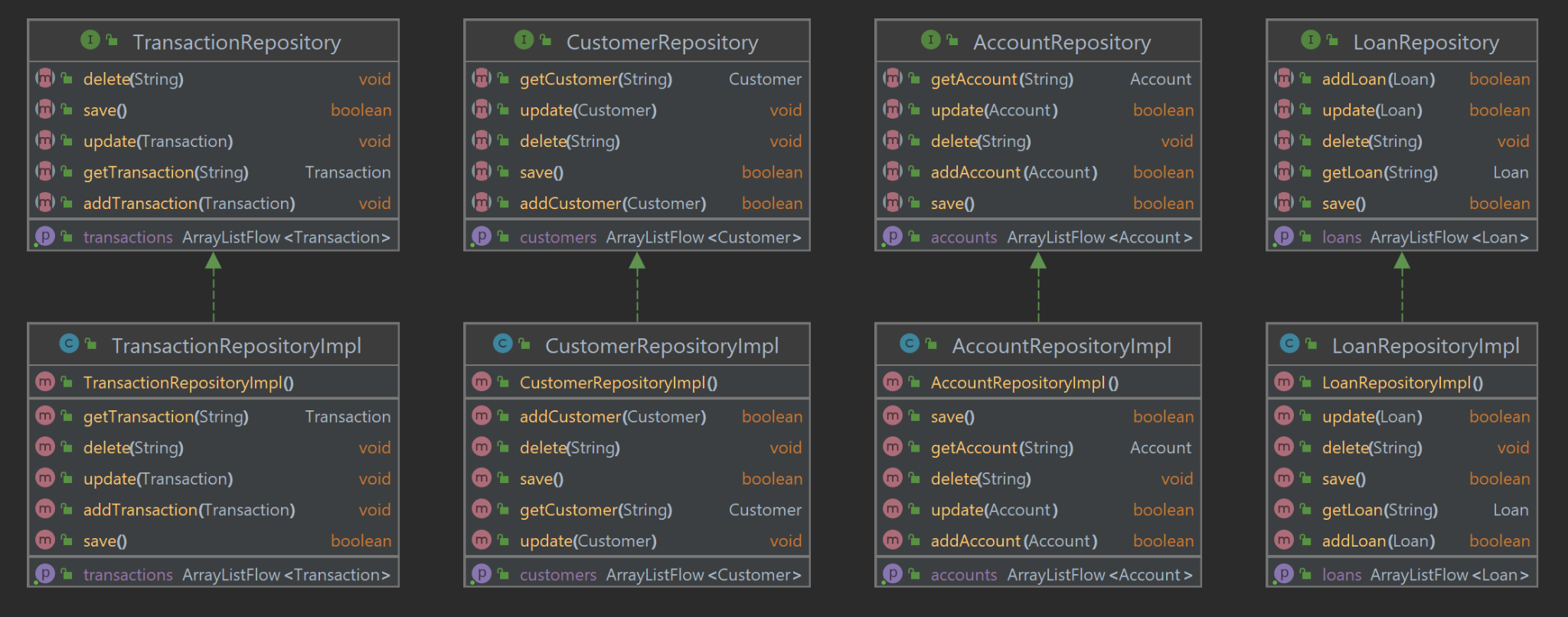
Class Diagrams

Below we have the class diagrams. Here you can easily visualize the polymorphism taking place with the abstract classes User, Account and Loan at the top and then more specific classes catered to more specific applications as you follow down the various charts. The last set of class diagrams describes the relationship between the data access objects (DAOs) and the repositories.



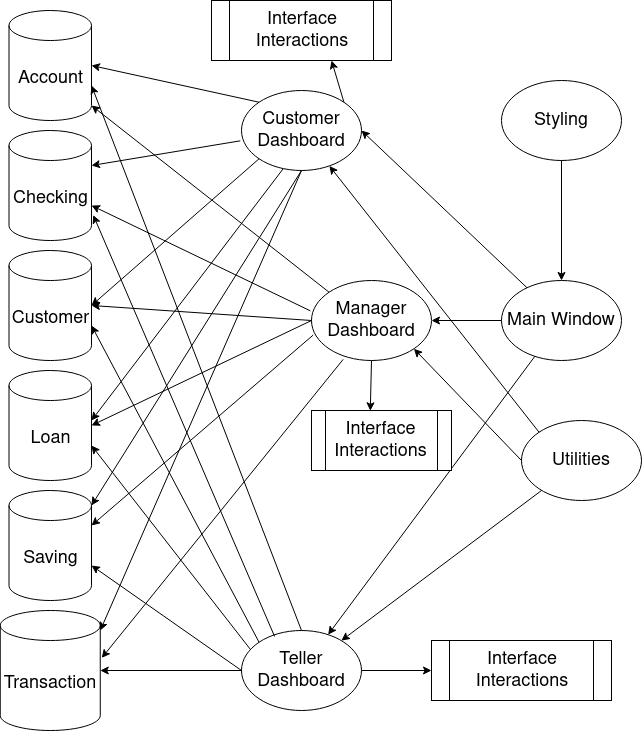






Architecture Diagram

The purpose of the architecture diagram is to provide a solid overall view of the functioning and flow of the system. Although many intermediate functions are implied, the larger overarching functions are described visually.



Functions and Main Drivers

*Main function for entry into the app*  
fun main() = application {  
 CompositionLocalProvider(LocalAppResources provides rememberAppResources()) {  
 BankApplication(rememberApplicationState())  
 }  
}

In the main function, we pass in RememberApplicationState into BankApplication. First lets look at some basic variables in functions listed in the BankApplicationState.kt:

*BankApplicationState.kt*

class BankApplicationState

*// The state of our tray, used for sending notifications to OS*  
val tray = TrayState()  
  
*// List of window states currently open*  
private val \_windows = *mutableStateListOf*<BankWindowState>()  
val windows: List<BankWindowState> get() = \_windows  
  
*// A dialog that is created when app is closing*  
*// Used for asking the user if they want to save*  
private val exitDialog = DialogState<AlertDialogResult>()  
  
*// A repository to pull and push customers to*  
val customerRepository: CustomerRepository = CustomerRepositoryImpl()  
  
*// A repository to pull and push accounts to*  
val accountRepository: AccountRepository = AccountRepositoryImpl()  
  
*// A repository to pull and push transactions to*  
val transactionRepository: TransactionRepository = TransactionRepositoryImpl()  
  
val loanRepository: LoanRepository = LoanRepositoryImpl()

*// A function to create a window state and add it to \_windows*  
fun newWindow()

*// A function to close a window when it asks to be closed*  
*// If it is the last window it will prompt the user to save*  
private suspend fun exitWindow(window: BankWindowState)

*// Function for sending notifications to the tray*  
*// TODO: Make notifications persist after app closes*  
private fun sendNotification(notification: Notification)

*// Job for concurrently saving our csv data*  
private var saveJob: Job? = null  
  
*// Function to tell the job to save our data*  
*// If this function fails the user will be notified and the window won't close*  
private suspend fun save(): Boolean {

*// Function to pop up the exit dialog and handle the user's selection*  
*// If the user selects yes, the app will attempt to save*  
*// If the save is successful, the app will close. If not, the app will notify the user and stay open*  
private suspend fun askToSave(): Boolean {

*/\*\**  
 *\* This class represents the different notifications our app can send*  
 *\* This could be converted to an enum but sealed classes allow optional parameters for flexibility*  
 *\*/*  
sealed class BankApplicationNotification {

*/\*\**  
 *\* This function is an extension function used to convert the sealed class objects to their corresponding notifications*  
 *\*/*  
fun BankApplicationNotification.format() = when (this) {

*/\*\**  
 *\* This class holds the state for any dialog boxes we pop up to the user*  
 *\*/*  
class DialogState<T> {

*/\*\**  
 *\* Function to launch a coroutine that attempts to save our data to disk*  
 *\*/*  
@OptIn(DelicateCoroutinesApi::class)

Now Let’s look at the BankApplication.kt itself. Since the file is small, we went ahead and included the whole thing:

*BankApplication.kt*

*/\*\*  
 \* This composable is responsible for creating windows for our list of window states  
 \*/*

@Composablefun ApplicationScope.BankApplication(state: BankApplicationState) { *AppTheme* **{**for (window in state.windows) { *key*(window) **{***BankWindow*(window) **}**} if (state.windows.*isNotEmpty*()) { *ApplicationTray*(state) } **}**}

*/\*\**  
 *\* This composable is responsible for keeping track of our tray icon if we have one*  
 *\* It is also responsible for handling notifications*  
 *\*/*  
@Composable  
private fun ApplicationScope.ApplicationTray(state: BankApplicationState) {  
 *Tray*(  
 *LocalAppResources*.current.icon,  
 state = state.tray,  
 tooltip = "Bank App",  
 )  
}

As you can see, the extent of the program primarily rests in the BankApplicaiton implementing the BankApplicationState. In our Bank Application, we import the BankWindow in which we access the initialized BankWindowState. This is where we begin to experience the implementation of functionality and interactions with the user.

*BankWindow.kt*

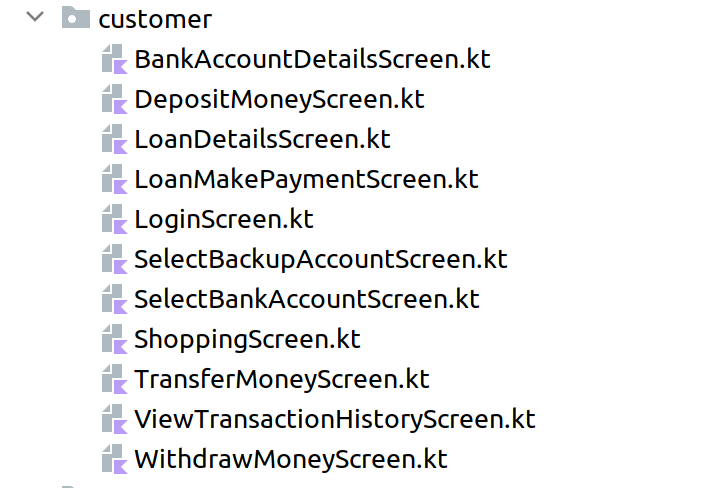
*/\*\*  
 \* This composable creates a window starting at our entry screen  
 \*/*@OptIn(ExperimentalDecomposeApi::class)@Composablefun BankWindow(state: BankWindowState) {

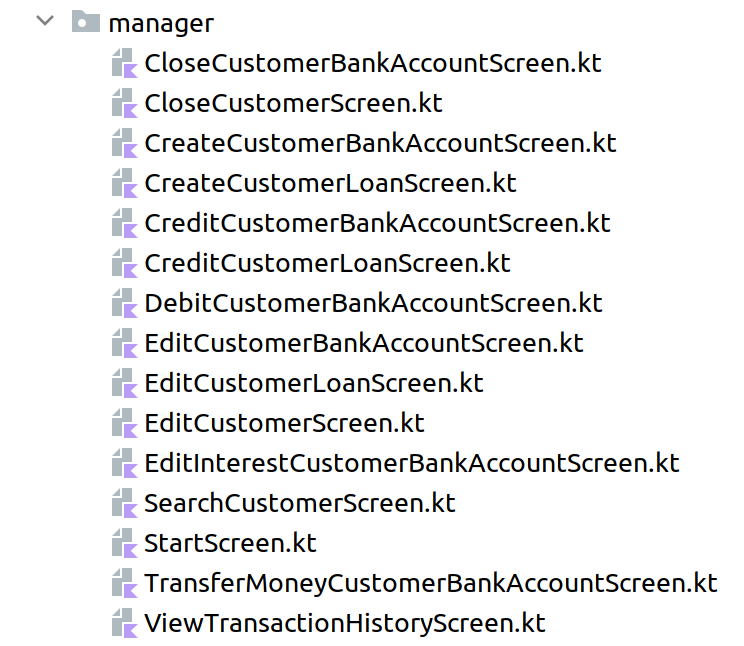
We also have a BankWindowState.kt to keep track of values in the BankWindow.

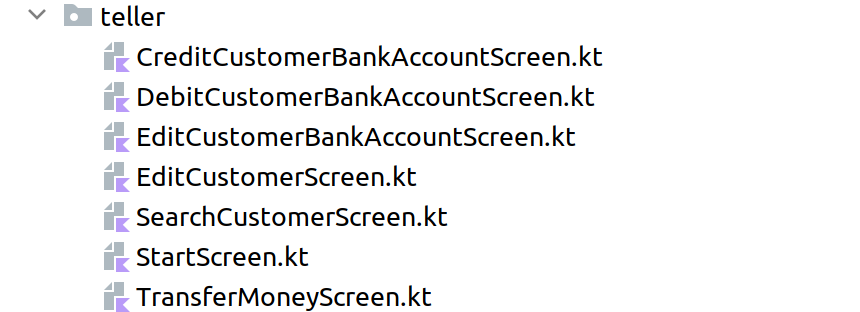
*BankWindowState.kt*

*/\*\*  
 \* This class holds our state for each window  
 \*/*class BankWindowState(

As you can see, we have a pattern of interactions between windows and states. In this same sense, we have different windows for users that are popped on and off of a stack that is saved as a state in the BankWindowState.kt. This allows us to scope in and out of the series of actions that can visually drill down into more detailed components of the application while also having the ability to back track to higher level scopes of the interface. Now that the main functionality of the interfaces has been explained, it would be useful to be aware and understand of the different windows that exist.







In addition to these screens/interfaces, we also have a CreateAccountScreen.kt and a LoginScreen.kt.

Logic/Java

It is important to understand the logic behind the interfaces in our program. First let’s give an overview on the view/model system for the Banking System Appliction:

In our banking system, the Model-View-ViewModel (MVVM) design pattern is employed to ensure a clean separation between the user interface (UI), application logic, and data management. This section explains the design concept and the roles of each component in the MVVM pattern.

Models: These classes represent the data structures, such as Account, Transaction, or Customer, which hold the actual data and define relationships between the different entities.

Data Access Objects (DAOs): Responsible for handling interactions with the data storage (e.g., a database), DAOs provide methods to create, read, update, and delete (CRUD) data in the storage.

Repositories: By managing data access through DAOs and providing a higher level of abstraction for the application, Repositories ensure the application code remains focused on the application logic without needing to know specifics about data storage.

ViewModels: These classes manage and transform data from Models and Repositories to make it available to the View (UI). ViewModels handle user input, delegate tasks to appropriate components (like Repositories), and contain properties and commands bound to UI elements for easy UI updates as data changes.

View (UI): Comprising UI elements like buttons, text boxes, and lists, the View presents data to the user and captures user inputs. It is bound to ViewModel properties and commands.

In our banking system, a ViewModel is responsible for tasks such as displaying a list of accounts, retrieving account details, and updating account balances when deposits or withdrawals are made. The ViewModel interacts with Repositories to fetch account data and processes user input (e.g., a deposit amount). It then delegates data storage updates back to the Repositories through DAOs. This approach ensures that the UI remains focused on presentation and user interaction, while the ViewModel manages data and application logic.

Lastly, let’s narrow down into a detail list of each class and its functions within the various files of the logic operations in the program.

## Account Class

The Account class is an abstract class that represents a generic bank account. It implements the CSV interface for CSV conversion and the Comparable interface for comparing accounts based on their opening dates.

### Constructor

public Account(@NotNull String accountNumber, @NotNull String customerSSN, @NotNull Double balance, @NotNull LocalDate dateOpened, @Nullable Double interestRate)

* Creates a new Account object with the provided account number, customer SSN, balance, date opened, and interest rate (which can be null for basic checking accounts).

### Methods

* compareTo(Account o): int: Compares two accounts based on their opening dates and returns an integer value indicating their relative order.
* deposit(double amount): void: Deposits a given amount into the account. If the amount is positive, it updates the balance and prints a success message with the new balance. If the amount is not positive, it prints an error message.
* withdraw(double amount): void: Withdraws a given amount from the account. If the amount is positive and there are sufficient funds, it updates the balance and prints a success message with the new balance. If the amount is not positive or there are insufficient funds, it prints an error message.
* getAccountNumber(): @NotNull String: Returns the account number.
* setAccountNumber(@NotNull String accountNumber): void: Sets the account number.
* getCustomerSSN(): @NotNull String: Returns the customer SSN.
* setCustomerSSN(@NotNull String customerSSN): void: Sets the customer SSN.
* getBalance(): @NotNull Double: Returns the current balance.
* setBalance(@NotNull Double balance): void: Sets the current balance.
* getDateOpened(): @NotNull LocalDate: Returns the date the account was opened.
* setDateOpened(@NotNull LocalDate dateOpened): void: Sets the date the account was opened.
* getInterestRate(): @Nullable Double: Returns the current interest rate.
* setInterestRate(@Nullable Double interestRate): void: Sets the current interest rate.
* convertToCSV(): String[]: Converts the account object to a CSV string array containing the account number, customer SSN, balance, and date opened.

## Class Documentation:

### ATMCard

#### Summary

The ATMCard class represents an ATM card associated with a CheckingAccount in a larger Java program. It contains a reference to a CheckingAccount object and provides methods to access and manipulate the associated account.

#### Code Documentation

##### Fields

* account: A CheckingAccount object representing the checking account associated with the ATM card. It is annotated with @NotNull, indicating that it cannot be null.

##### Constructors

* ATMCard(CheckingAccount account): A constructor that creates an ATMCard object with the provided CheckingAccount object. The account parameter is annotated with @NotNull, indicating that it cannot be null.

##### Methods

* getAccount(): CheckingAccount: A method that returns the CheckingAccount object associated with the ATM card.

#### Function Overview

The ATMCard class serves as a representation of an ATM card associated with a CheckingAccount. It allows access to the associated checking account through the getAccount() method, which returns the CheckingAccount object. The ATMCard class can be used to perform operations related to the checking account, such as checking the balance, making deposits, and withdrawals, depending on the functionality provided by the CheckingAccount class.

### CDAccount

#### Summary

The CDAccount class represents a certificate of deposit (CD) savings account in a bank system. It is a subclass of SavingsAccount and inherits its properties and methods. The CDAccount class adds a due date field to represent the date when the CD is due to complete, and provides methods to access and manipulate this due date.

#### Code Documentation

##### Fields

* dueDate: A LocalDate object representing the date when the CD is due to complete. It is annotated with @NotNull, indicating that it cannot be null.

##### Constructors

* CDAccount(String accountNumber, String customerSSN, Double balance, LocalDate dateOpened, Double interestRate, LocalDate dueDate): A constructor that creates a CDAccount object with the provided account number, customer SSN, balance, date opened, interest rate, and due date. The account number, customer SSN, balance, date opened, and interest rate parameters are not null and must be provided.

##### Methods

* getDueDate(): LocalDate: A method that returns the due date of the CD account.
* setDueDate(LocalDate dueDate): A method that sets the due date of the CD account.
* convertToCSV(): String[]: An overridden method that converts the CDAccount object to a CSV (Comma Separated Values) format. It returns an array of strings containing the values of the account number, customer SSN, balance, date opened, account type, interest rate, and due date, which can be used for data serialization or storage.

#### Function Overview

The CDAccount class extends the SavingsAccount class and represents a CD savings account in a bank system. It provides methods to access and manipulate the due date of the CD account, and also overrides the convertToCSV() method to convert the object to a CSV format. The CDAccount class can be used to perform operations related to CD savings accounts, such as getting the due date, setting the due date, and converting the object to a CSV format for storage or serialization purposes.

### CheckingAccount

#### Summary

The CheckingAccount class represents a checking account in a bank system. It is an abstract class that extends the Account class and provides additional fields and methods specific to checking accounts, such as transaction fee, backup account, overdrafts this month, and ATM card.

#### Code Documentation

##### Fields

* transactionFee: A Double value representing the transaction fee associated with the checking account. It is annotated with @NotNull, indicating that it cannot be null.
* backupAccount: A reference to a SavingsAccount object that is designated as the backup account for the checking account. It is annotated with @Nullable, indicating that it can be null.
* overdraftsThisMonth: An Integer value representing the number of overdrafts that have occurred in the current month. It is annotated with @NotNull, indicating that it cannot be null.
* atmCard: An ATMCard object representing the ATM card associated with the checking account. It is annotated with @Nullable, indicating that it can be null.

##### Constructors

* CheckingAccount(String accountNumber, String customerSSN, Double balance, LocalDate dateOpened, Double interestRate, Double transactionFee, SavingsAccount backupAccount, Integer overdraftsThisMonth, Boolean hasATMCard): A constructor that creates a CheckingAccount object with the provided account number, customer SSN, balance, date opened, interest rate, transaction fee, backup account, overdrafts this month, and flag indicating if the account has an ATM card. The account number, customer SSN, balance, date opened, transaction fee, and overdrafts this month parameters are not null and must be provided. If the hasATMCard flag is true, an ATMCard object is created and associated with the checking account.

##### Methods

* getTransactionFee(): Double: A method that returns the transaction fee associated with the checking account.
* setTransactionFee(Double transactionFee): A method that sets the transaction fee for the checking account.
* getBackupAccount(): SavingsAccount: A method that returns the backup account associated with the checking account.
* setBackupAccount(SavingsAccount backupAccount): A method that sets the backup account for the checking account.
* getOverdraftsThisMonth(): Integer: A method that returns the number of overdrafts that have occurred in the current month for the checking account.
* setOverdraftsThisMonth(Integer overdraftsThisMonth): A method that sets the number of overdrafts for the checking account.
* getAtmCard(): ATMCard: A method that returns the ATM card associated with the checking account.
* setAtmCard(ATMCard atmCard): A method that sets the ATM card for the checking account.

### GoldDiamondAccount Function Overview

The CheckingAccount class extends the Account class and represents a checking account in a bank system. It provides methods to access and manipulate fields such as transaction fee, backup account, overdrafts this month, and ATM card associated with the checking account. The CheckingAccount class can be used to perform operations related to checking accounts, such as getting transaction fee, setting backup account, getting overdrafts this month, and setting ATM card, among others.

This code defines a class GoldDiamondAccount which extends the CheckingAccount class. It represents a type of checking account that has additional features specific to a "Gold" or "Diamond" level of account.

The GoldDiamondAccount class has the following attributes:

* minimumBalance: A Double value representing the minimum balance required to avoid transaction fees for this account. It is initialized with a default value of 5000.0, but can be set to a different value using the setMinimumBalance() method.

The GoldDiamondAccount class has the following constructor:

* GoldDiamondAccount: A constructor that takes in several parameters including accountNumber, customerSSN, balance, dateOpened, interestRate, backupAccount, hasATMCard, and overdraftsThisMonth. It calls the superclass CheckingAccount constructor with appropriate values, including a transaction fee of 0.0 if the balance is greater than or equal to the default minimum balance, or 0.75 otherwise. It also sets the minimumBalance attribute to the default minimum balance.

The GoldDiamondAccount class also has an overridden method convertToCSV() that returns an array of String values representing the object's attributes, including the additional attributes specific to GoldDiamondAccount class.

Note: There is a TODO comment in the code mentioning the need for a factory for GoldDiamondAccount due to different combinations. Depending on the specific requirements of the application, a factory method or a separate factory class may need to be implemented to create GoldDiamondAccount objects with different combinations of attributes.

### SavingsAccount Function Overview

This code defines a class SavingsAccount which extends the Account class. It represents a type of savings account.

The SavingsAccount class has the following constructor:

* SavingsAccount: A constructor that takes in several parameters including accountNumber, customerSSN, balance, dateOpened, and interestRate. It calls the superclass Account constructor with the provided values for accountNumber, customerSSN, balance, dateOpened, and interestRate.

The SavingsAccount class also has an overridden method convertToCSV() that returns an array of String values representing the object's attributes, including the additional attributes specific to the SavingsAccount class. In this case, it includes a type identifier "S" and the interest rate of the savings account.

Note: There is a TODO comment in the code mentioning the need to implement business logic of savings account. This may refer to additional methods or functionality that needs to be implemented for the SavingsAccount class to fully represent the business requirements of the application. Depending on the specific requirements of the application, additional methods or logic may need to be added to handle savings account-related operations such as interest calculations, withdrawal limits, and account restrictions, among others.

### TMBAccount Function Overview

This code defines a class TMBAccount which extends the CheckingAccount class, representing a type of checking account with specific features.

The TMBAccount class has the following constructors:

* TMBAccount: A constructor that takes in several parameters including accountNumber, customerSSN, balance, dateOpened, hasATMCard, backupAccount, and overdraftsThisMonth. It calls the superclass CheckingAccount constructor with the provided values for accountNumber, customerSSN, balance, dateOpened, null for interestRate, a default transaction fee of 0.75, backupAccount, overdraftsThisMonth, and hasATMCard.
* TMBAccount: An overloaded constructor that takes in similar parameters as the first constructor, except it does not require a backupAccount. It calls the superclass CheckingAccount constructor with the provided values, passing null for backupAccount.

The TMBAccount class also has an overridden method convertToCSV() that returns an array of String values representing the object's attributes, including the additional attributes specific to the TMBAccount class. In this case, it includes a type identifier "TMB", the accountNumber of the backupAccount, a boolean value indicating if the account has an ATM card, and the number of overdrafts made in the current month.

Note: It's worth mentioning that in the code, TMBAccount is extending CheckingAccount and using its methods and attributes. However, it's not clear from the code snippet what methods or functionality CheckingAccount provides, and if TMBAccount is implementing the correct business logic based on the application's requirements. Depending on the specific requirements of the application, additional methods or logic may need to be added or overridden in the TMBAccount class to fully represent the business requirements of the application.

## Loan Class

The Loan class is an abstract class that represents a loan in a larger Java program. It implements the CSV interface and extends the Comparable<Loan> interface, allowing for CSV conversion and comparison between loan objects.

### Class Summary

* Package: edu.missouriwestern.csc406team1.database.model.loan
* Extends: Comparable<Loan>
* Implements: CSV
* Access Modifiers: public
* Abstract: Yes

### Class Variables

* accountNumber (String): The account number associated with the loan. Not null.
* customerSSN (String): The SSN (Social Security Number) of the customer owning the loan. Not null.
* datePaymentDue (LocalDate): The date when the next payment is due. Not null.
* paymentNotified (LocalDate): The date when notification is sent for the next payment. Nullable.
* currentPaymentDue (Double): The amount due for the next payment. Not null.
* dateSinceLastPayment (LocalDate): The date when the last payment was received. Not null.
* missedPayment (Boolean): A flag indicating if a payment was missed. Not null.
* interestRate (Double): The interest rate of the loan. Not null.
* balance (Double): The current balance of the loan. Not null.
* dateOpened (LocalDate): The date when the loan was opened with the bank. Not null.

### Constructors

* Loan(String accountNumber, String customerSSN, Double balance, Double interestRate, LocalDate datePaymentDue, LocalDate paymentNotified, Double currentPaymentDue, LocalDate dateSinceLastPayment, Boolean missedPayment): Creates a Loan object with the provided parameters. The order of the parameters should follow the order of CSV data for ease of use.

### Methods

* getAccountNumber(): String: Returns the account number associated with the loan.
* setAccountNumber(String accountNumber): void: Sets the account number of the loan.
* getDatePaymentDue(): LocalDate: Returns the date when the next payment is due.
* setDatePaymentDue(LocalDate datePaymentDue): void: Sets the date when the next payment is due.
* getPaymentNotified(): LocalDate: Returns the date when notification is sent for the next payment.
* setPaymentNotified(LocalDate paymentNotified): void: Sets the date when notification is sent for the next payment.
* getCurrentPaymentDue(): Double: Returns the amount due for the next payment.
* setCurrentPaymentDue(Double currentPaymentDue): void: Sets the amount due for the next payment.
* getDateSinceLastPayment(): LocalDate: Returns the date when the last payment was received.
* setDateSinceLastPayment(LocalDate dateSinceLastPayment): void: Sets the date when the last payment was received.
* hasMissedPayment(): Boolean: Returns a boolean indicating if a payment was missed.
* setMissedPayment(Boolean missedPayment): void: Sets the flag indicating if a payment was missed.
* convertToCSV(): String[]: Implements the convertToCSV method from the CSV interface, which converts the Loan object to a string array of CSV data. The array contains the account number, customer SSN, balance, and date opened.

Note: This class is abstract and cannot be instantiated directly. It serves as a base class for other types of loans that extend from it, allowing customization of functionality in the child classes.

That concludes the documentation for the Loan class.

### Function Overviews

#### CreditCardLoan

The CreditCardLoan class extends the Loan class and represents a specific type of loan that is a credit card loan. It has an additional attribute creditLimit which represents the credit limit of the credit card loan. The constructor of CreditCardLoan takes the same parameters as the constructor of Loan along with the creditLimit parameter, and calls the super constructor of Loan to initialize the common attributes.

The CreditCardLoan class can now implement its own business logic based on the requirements of credit card loans. This may include methods for calculating interest, handling payments, updating credit limit, and any other specific functionality related to credit card loans. Additionally, any additional attributes or methods specific to credit card loans can be added to this class as needed.

#### MortgageLoan

The MortgageLoan class extends the Loan class and represents a specific type of loan that is a mortgage loan. It does not have any additional attributes beyond those inherited from the Loan class. The constructor of MortgageLoan takes the same parameters as the constructor of Loan and calls the super constructor of Loan to initialize the common attributes.

The MortgageLoan class can now implement its own business logic based on the requirements of mortgage loans. This may include methods for calculating interest, handling payments, and any other specific functionality related to mortgage loans. Additionally, any additional attributes or methods specific to mortgage loans can be added to this class as needed.

#### ShortTermLoan

The ShortTermLoan class extends the Loan class and represents a specific type of loan that is a short-term loan. It does not have any additional attributes beyond those inherited from the Loan class. The constructor of ShortTermLoan takes the same parameters as the constructor of Loan and calls the super constructor of Loan to initialize the common attributes.

The ShortTermLoan class can now implement its own business logic based on the requirements of short-term loans. This may include methods for calculating interest, handling payments, and any other specific functionality related to short-term loans. Additionally, any additional attributes or methods specific to short-term loans can be added to this class as needed.

## Customer Class

The Customer class is a model for representing a customer object. It implements the Comparable and CSV interfaces to enable sorting and writing to disk in CSV format.

### Class Summary

* Package: edu.missouriwestern.csc406team1.database.model
* Implemented Interfaces: Comparable<Customer>, CSV

### Fields

* ssn: A String representing the social security number of the customer. It is annotated with @NotNull to indicate that it cannot be null.
* address: A String representing the address of the customer. It is annotated with @NotNull to indicate that it cannot be null.
* city: A String representing the city of the customer. It is annotated with @NotNull to indicate that it cannot be null.
* state: A String representing the state of the customer. It is annotated with @NotNull to indicate that it cannot be null.
* zipcode: A String representing the zip code of the customer. It is annotated with @NotNull to indicate that it cannot be null.
* firstname: A String representing the first name of the customer. It is annotated with @NotNull to indicate that it cannot be null.
* lastname: A String representing the last name of the customer. It is annotated with @NotNull to indicate that it cannot be null.

### Constructor

* Customer(String ssn, String address, String city, String state, String zipcode, String firstname, String lastname): A constructor that takes in the required parameters of a customer object, including social security number, address, city, state, zip code, first name, and last name, and initializes the corresponding fields.

### Methods

* String getSsn(): A getter method that returns the social security number of the customer.
* String getAddress(): A getter method that returns the address of the customer.
* void setAddress(String address): A setter method that sets the address of the customer.
* String getCity(): A getter method that returns the city of the customer.
* void setCity(String city): A setter method that sets the city of the customer.
* String getState(): A getter method that returns the state of the customer.
* void setState(String state): A setter method that sets the state of the customer.
* String getZipcode(): A getter method that returns the zip code of the customer.
* void setZipcode(String zipcode): A setter method that sets the zip code of the customer.
* String getFirstname(): A getter method that returns the first name of the customer.
* void setFirstname(String firstname): A setter method that sets the first name of the customer.
* String getLastname(): A getter method that returns the last name of the customer.
* void setLastname(String lastname): A setter method that sets the last name of the customer.
* int compareTo(Customer o): An overridden method from the Comparable interface that compares this customer object with another customer object based on their first names. It returns a negative integer, zero, or a positive integer depending on whether this object is less than, equal to, or greater than the specified object.
* String[] convertToCSV(): An overridden method from the CSV interface that converts the customer object to an array of strings in CSV format, with each field as a separate element in the array.

Note: The @NotNull annotations indicate that the corresponding fields or parameters cannot be null, and any attempt to set them as null will result in a NullPointerException.

## Transaction Class

The Transaction class is a part of the larger Java program and is located in the edu.missouriwestern.csc406team1.database.model package. It represents a transaction with various attributes such as transaction ID, credit, debit, amount, account ID, new total, date, time, and transaction type. It implements the Comparable and CSV interfaces and provides methods to convert the object to CSV format and compare transactions based on certain criteria.

### Class Signature

public class Transaction implements Comparable<Transaction>, CSV {

### Constructors

* Transaction(String transactionID, Boolean credit, Boolean debit, String transactionType, Double amount, Double newTotal, String accID, LocalDate date, LocalTime time): A constructor that takes in the transaction ID, credit status, debit status, transaction type, amount, new total, account ID, date, and time as parameters and initializes the corresponding attributes of the Transaction object.

### Getters and Setters

* Boolean isCredit(): Returns the credit status of the transaction.
* void setCredit(boolean credit): Sets the credit status of the transaction.
* Boolean isDebit(): Returns the debit status of the transaction.
* void setDebit(Boolean debit): Sets the debit status of the transaction.
* Double getAmount(): Returns the amount of the transaction.
* void setAmount(Double amount): Sets the amount of the transaction.
* String getAccID(): Returns the account ID associated with the transaction.
* void setAccID(String accID): Sets the account ID associated with the transaction.
* Double getNewTotal(): Returns the new total after the transaction.
* void setNewTotal(Double newTotal): Sets the new total after the transaction.
* LocalDate getDate(): Returns the date of the transaction.
* void setDate(LocalDate date): Sets the date of the transaction.
* LocalTime getTime(): Returns the time of the transaction.
* void setTime(LocalTime time): Sets the time of the transaction.
* String getTransactionType(): Returns the type of the transaction.
* void setTransactionType(String transactionType): Sets the type of the transaction.
* void setTransactionID(String transactionID): Sets the ID of the transaction.
* String getTransactionID(): Returns the ID of the transaction.

### Implemented Methods

* String[] convertToCSV(): Implements the CSV interface method and returns an array of strings representing the object in CSV format.
* int compareTo(Transaction o): Implements the Comparable interface method and compares transactions based on certain criteria.

Please note that the convertToCSV() and compareTo() methods currently return a placeholder value of 0 and an empty string array new String[0] respectively, and need to be implemented according to the specific requirements of the larger Java program.

## AccountRepositoryImpl Class

### Summary

The AccountRepositoryImpl class is a Java class that implements the AccountRepository interface. It serves as a repository for managing Account objects in a database. It provides methods for retrieving, adding, updating, and deleting Account objects, as well as saving changes to the database.

### Code Documentation

The AccountRepositoryImpl class is located in the edu.missouriwestern.csc406team1.database package and requires the following imports:

import edu.missouriwestern.csc406team1.ArrayListFlow;

import edu.missouriwestern.csc406team1.database.dao.AccountDao;

import edu.missouriwestern.csc406team1.database.dao.AccountDaoImpl;

import edu.missouriwestern.csc406team1.database.model.account.Account;

#### Class Members

* private final AccountDao accountDao: An instance of the AccountDao interface that is used to interact with the underlying database.

#### Constructors

* public AccountRepositoryImpl(): A constructor that initializes the accountDao field with an instance of AccountDaoImpl.

#### Methods

* @Override public Account getAccount(String accountNumber): Retrieves an Account object from the database based on the provided account number. Returns the retrieved Account object or null if not found.
* @Override public ArrayListFlow<Account> getAccounts(): Retrieves a list of all Account objects from the database. Returns an ArrayListFlow object containing the retrieved Account objects.
* @Override public void addAccount(Account account): Adds an Account object to the database.
* @Override public void update(Account account): Updates an existing Account object in the database.
* @Override public void delete(Account account): Deletes an existing Account object from the database based on the provided Account object.
* @Override public boolean save(): Saves any changes made to the Account objects in the database and returns true if the save operation was successful, false otherwise.

### Overview of Functions

1. getAccount(String accountNumber): Retrieves an Account object from the database based on the provided account number.
2. getAccounts(): Retrieves a list of all Account objects from the database.
3. addAccount(Account account): Adds an Account object to the database.
4. update(Account account): Updates an existing Account object in the database.
5. delete(Account account): Deletes an existing Account object from the database based on the provided Account object.
6. save(): Saves any changes made to the Account objects in the database and returns true if the save operation was successful, false otherwise.

## CustomerRepositoryImpl Class

This class is an implementation of the CustomerRepository interface, which serves as a layer of abstraction between the DAO (Data Access Object) and the business logic. It provides methods for retrieving, adding, updating, and deleting customers in the system, as well as saving customer data to disk.

### Constructors

#### CustomerRepositoryImpl()

* A simple constructor that instantiates the CustomerDao object.

### Methods

#### getCustomer(String ssn): Customer

* Retrieves a customer from the system based on their SSN (Social Security Number).
* Parameters:
  + ssn (String): The SSN of the customer to retrieve.
* Returns:
  + Customer: The retrieved Customer object if found, or null if not found.

#### getCustomers(): ArrayListFlow<Customer>

* Retrieves a custom ArrayListFlow of all customers in the system.
* Returns:
  + ArrayListFlow<Customer>: The ArrayListFlow containing all customers in the system.

#### addCustomer(Customer customer): void

* Adds a customer to the system.
* Parameters:
  + customer (Customer): The Customer object to add.

#### update(Customer customer): void

* Updates an existing customer in the system.
* Parameters:
  + customer (Customer): The Customer object to update.

#### delete(Customer customer): void

* Deletes a customer from the system.
* Parameters:
  + customer (Customer): The Customer object to delete.

#### save(): boolean

* Attempts to save the customers to disk.
* Returns:
  + boolean: true if the customers are successfully saved, false otherwise.

## TransactionRepositoryImpl Class

The TransactionRepositoryImpl class is part of the larger Java program and is located in the edu.missouriwestern.csc406team1.database package. It implements the TransactionRepository interface and serves as a repository for managing transactions in a database.

### Summary

* Package: edu.missouriwestern.csc406team1.database
* Class: TransactionRepositoryImpl
* Interface: TransactionRepository
* Dependencies:
  + edu.missouriwestern.csc406team1.ArrayListFlow
  + edu.missouriwestern.csc406team1.database.dao.TransactionDao
  + edu.missouriwestern.csc406team1.database.dao.TransactionDaoImpl
  + edu.missouriwestern.csc406team1.database.model.Transaction

### Constructors

* TransactionRepositoryImpl(): Creates a new TransactionRepositoryImpl object and initializes the transactionDao field with a new instance of TransactionDaoImpl.

### Methods

* getTransaction(String transactionID): Retrieves a transaction from the repository based on the given transaction ID. Returns a Transaction object.
* getTransactions(): Retrieves all transactions from the repository as an ArrayListFlow object, which is a custom class that extends ArrayList with additional functionality.
* addTransaction(Transaction transaction): Adds a new transaction to the repository.
* update(Transaction transaction): Updates an existing transaction in the repository with the given transaction object.
* delete(Transaction transaction): Deletes a transaction from the repository based on the given transaction object.
* save(): Saves any changes made to the repository and returns a boolean value indicating whether the save operation was successful or not.

Note: All of the above methods delegate their operations to the TransactionDao interface, which is implemented by the TransactionDaoImpl class. The TransactionRepositoryImpl class acts as an intermediary between the application logic and the data access layer, encapsulating the database operations related to transactions.

This class can be used as a repository for managing transactions in a larger Java application that requires interaction with a database. The TransactionRepositoryImpl class provides abstraction and separation of concerns by encapsulating the database operations related to transactions, allowing for modular and organized code. Developers can use the provided methods to perform CRUD (Create, Read, Update, Delete) operations on transactions in the database, making it a useful tool for managing transactions in a Java application.