**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.

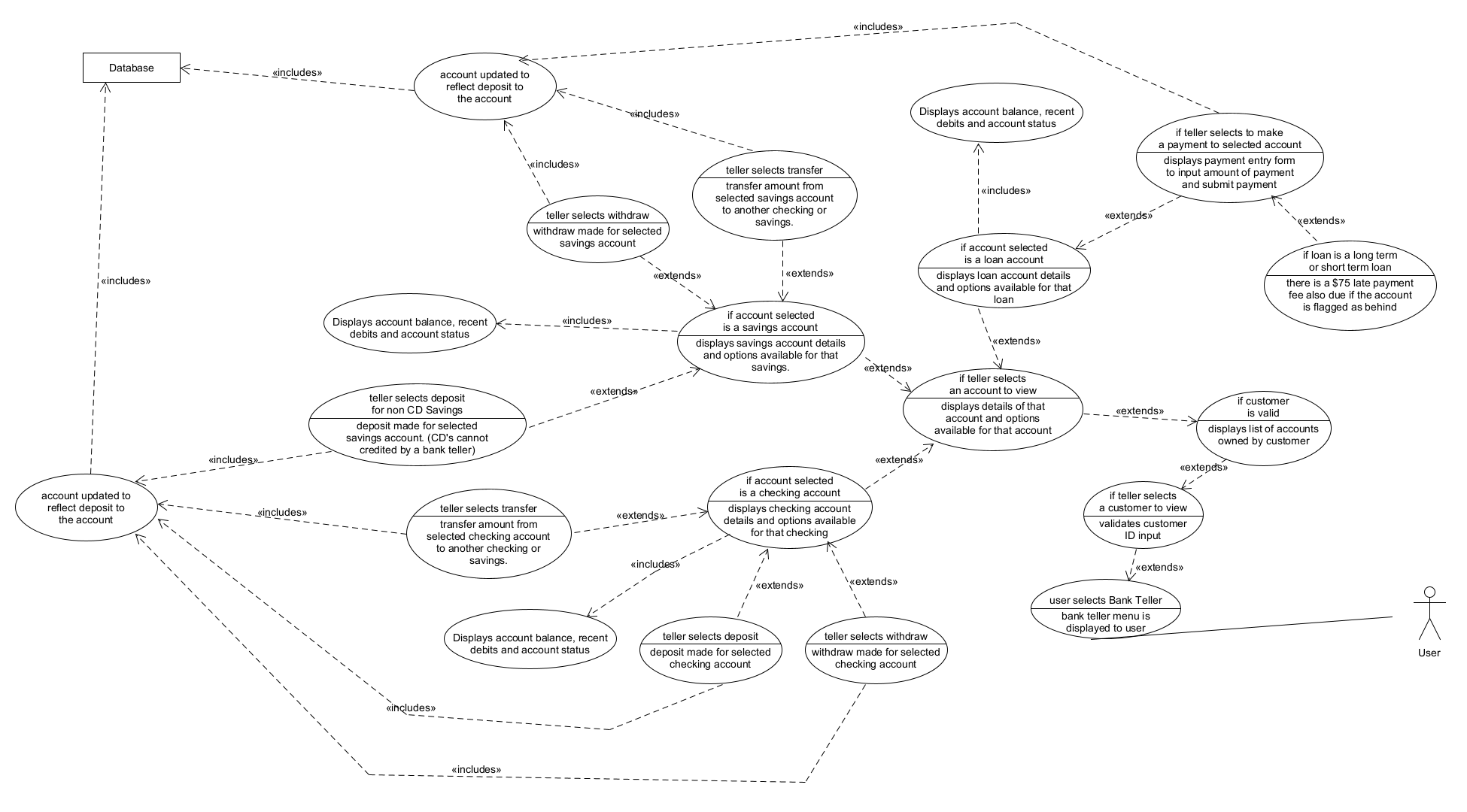
Lastly, it would be helpful to have an understanding of the basic operations and initial interfaces of the program and the purpose from a user perspective. The purpose of this program is to mock a banking system application that maintains user accounts and transactions at a bank. The application is built 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

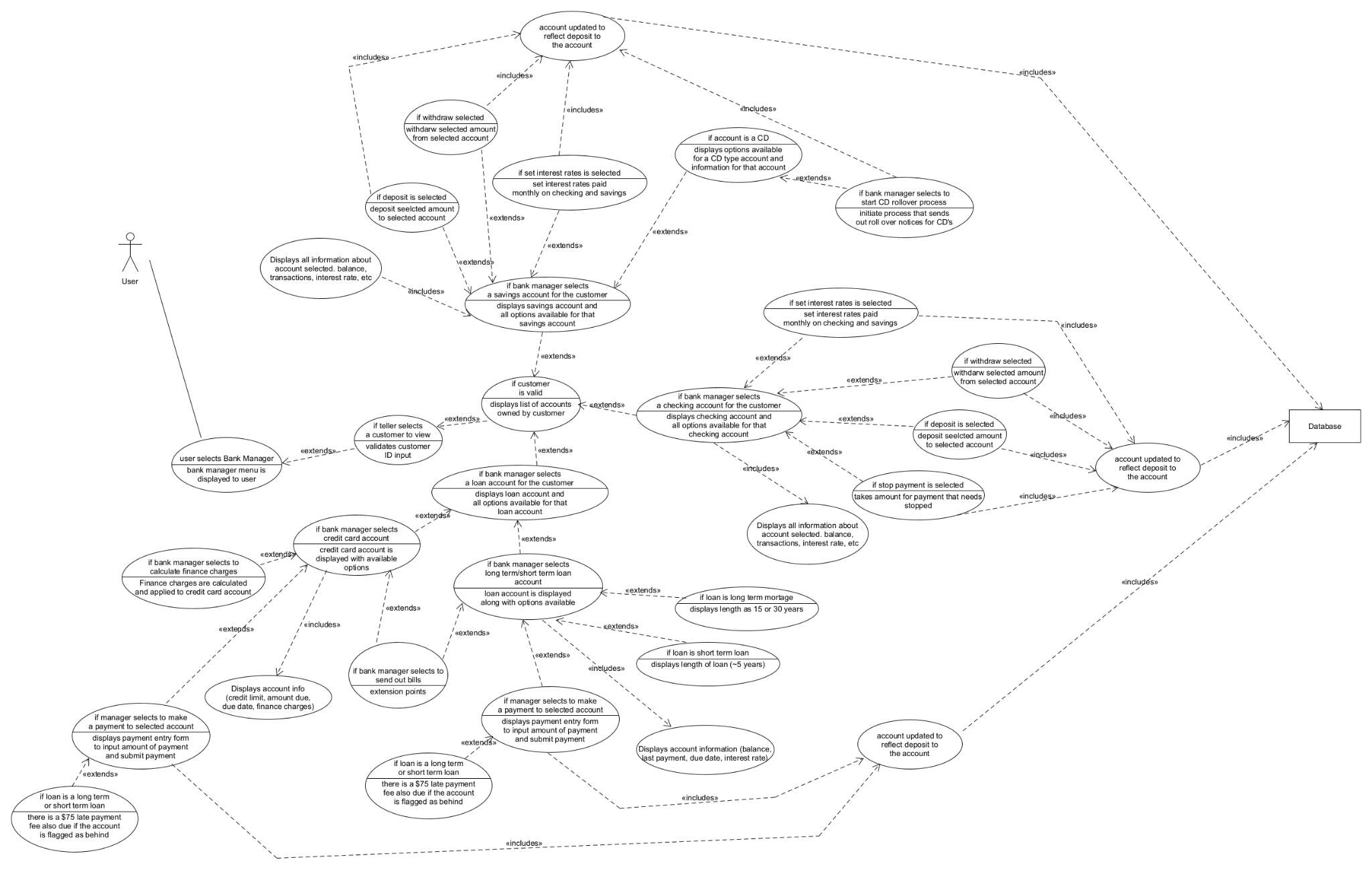
Below are 3 design diagrams that may be utilized to conceptualize the process of the interfaces and functionality of the program. First there is a use case diagram, then an activity diagram, and lastly a set of sequence diagrams detailing both the user repositories as well as the user data access objects.

*Use Case Diagram*

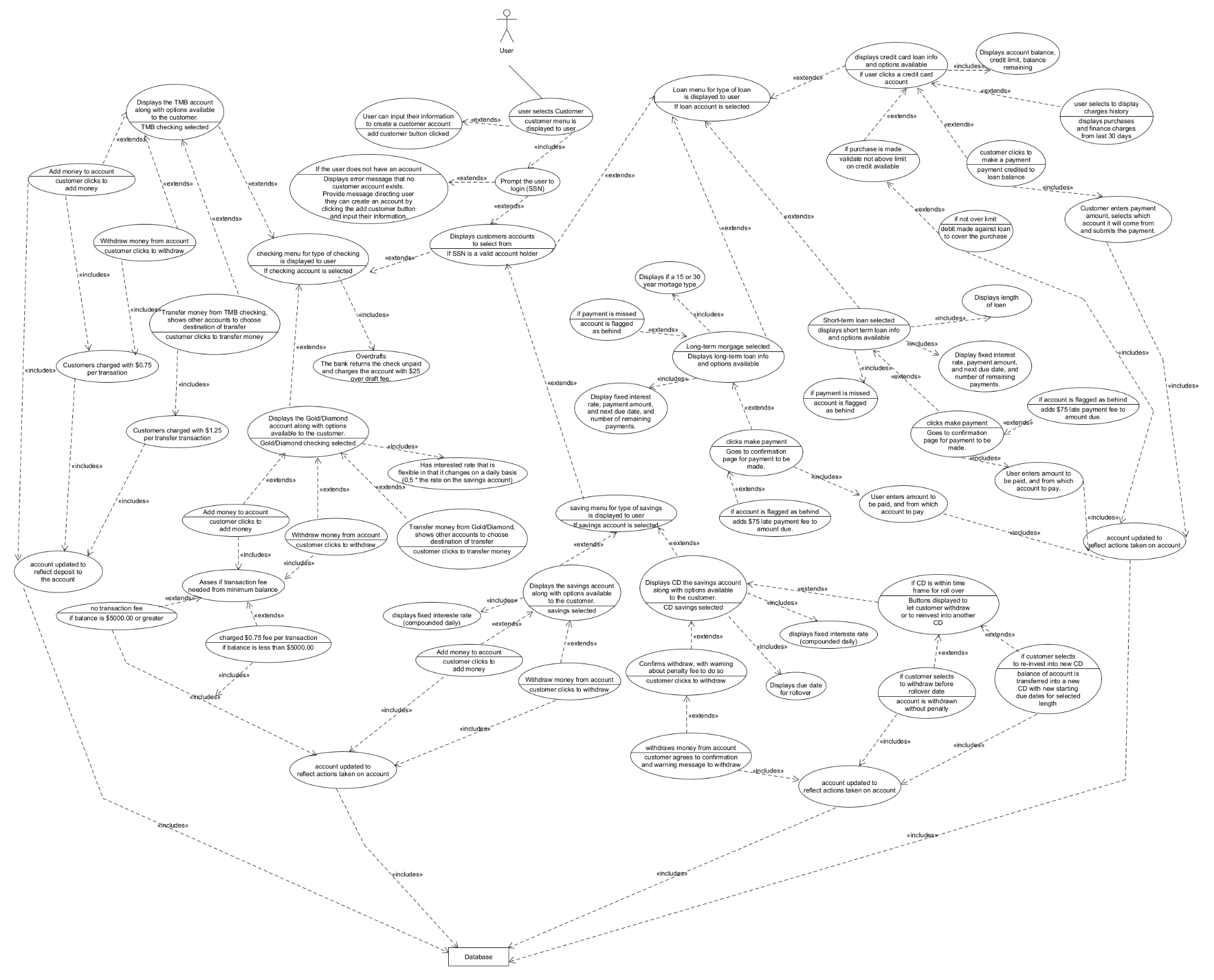
*Bank Teller*



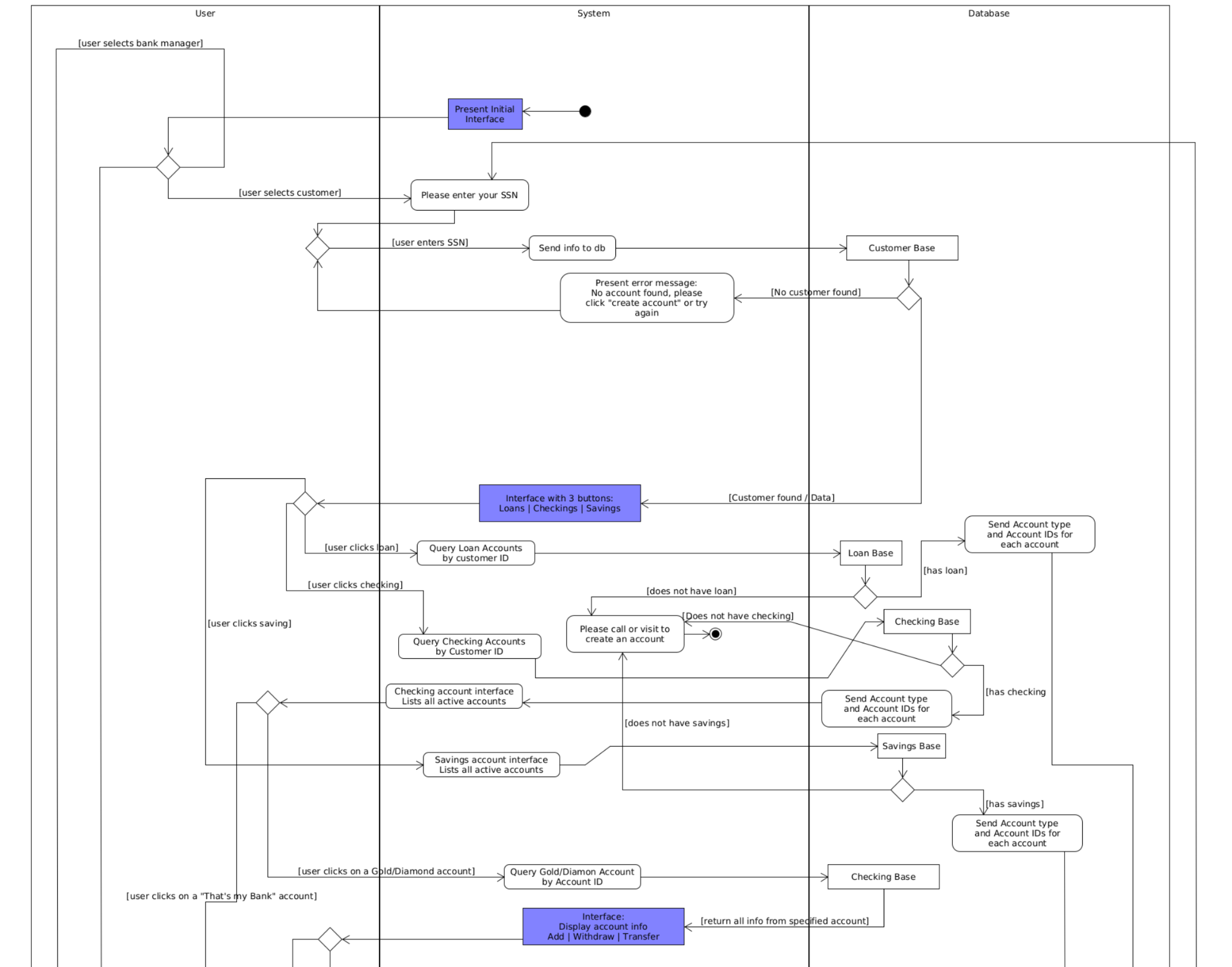
*Bank Manager*

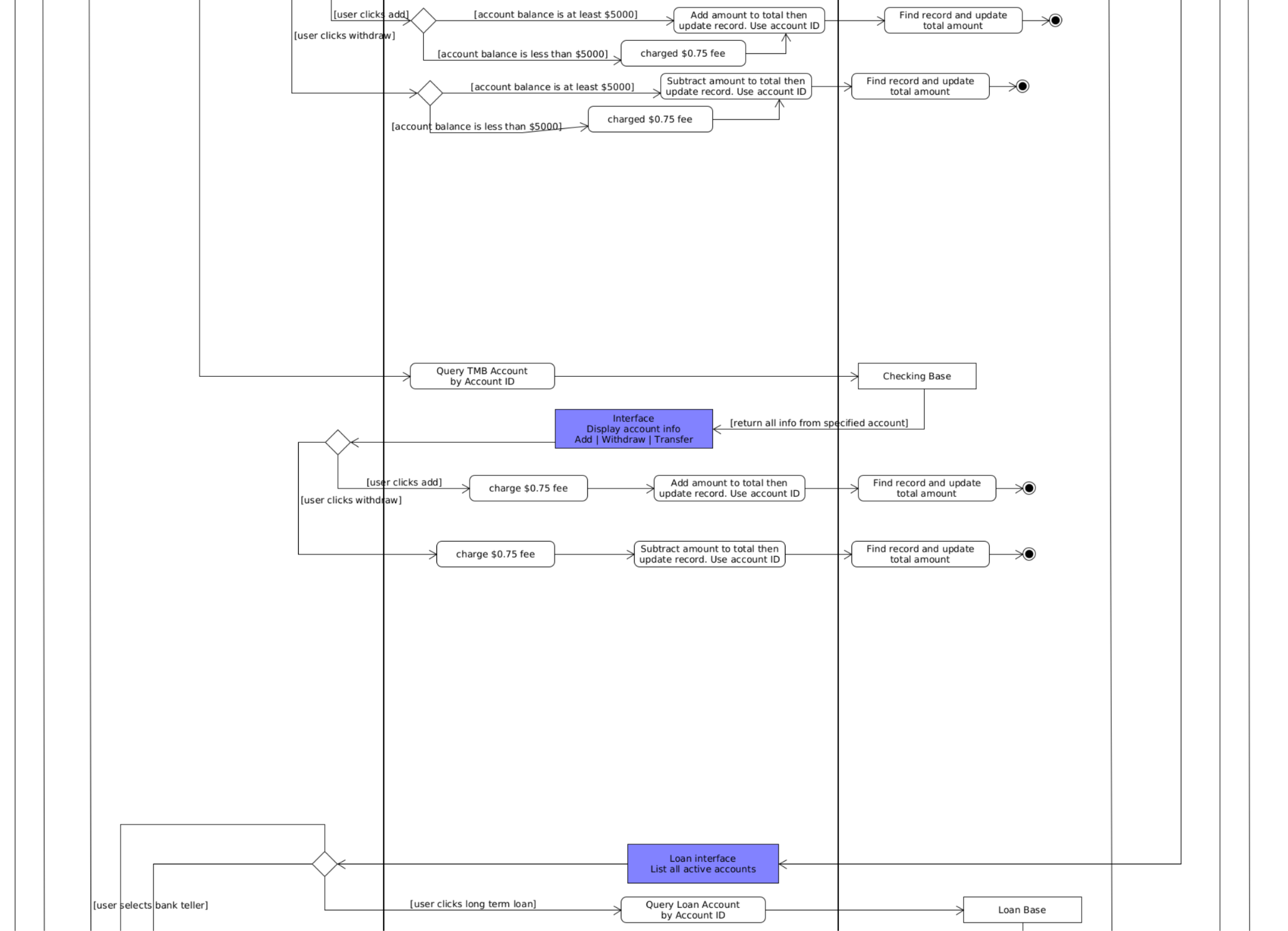


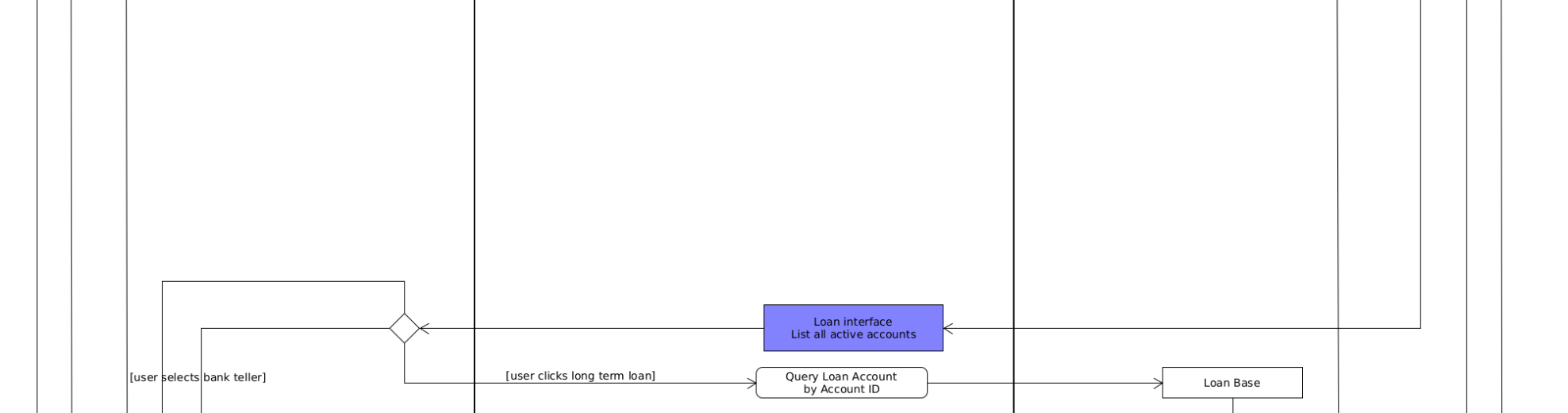
*Customer*

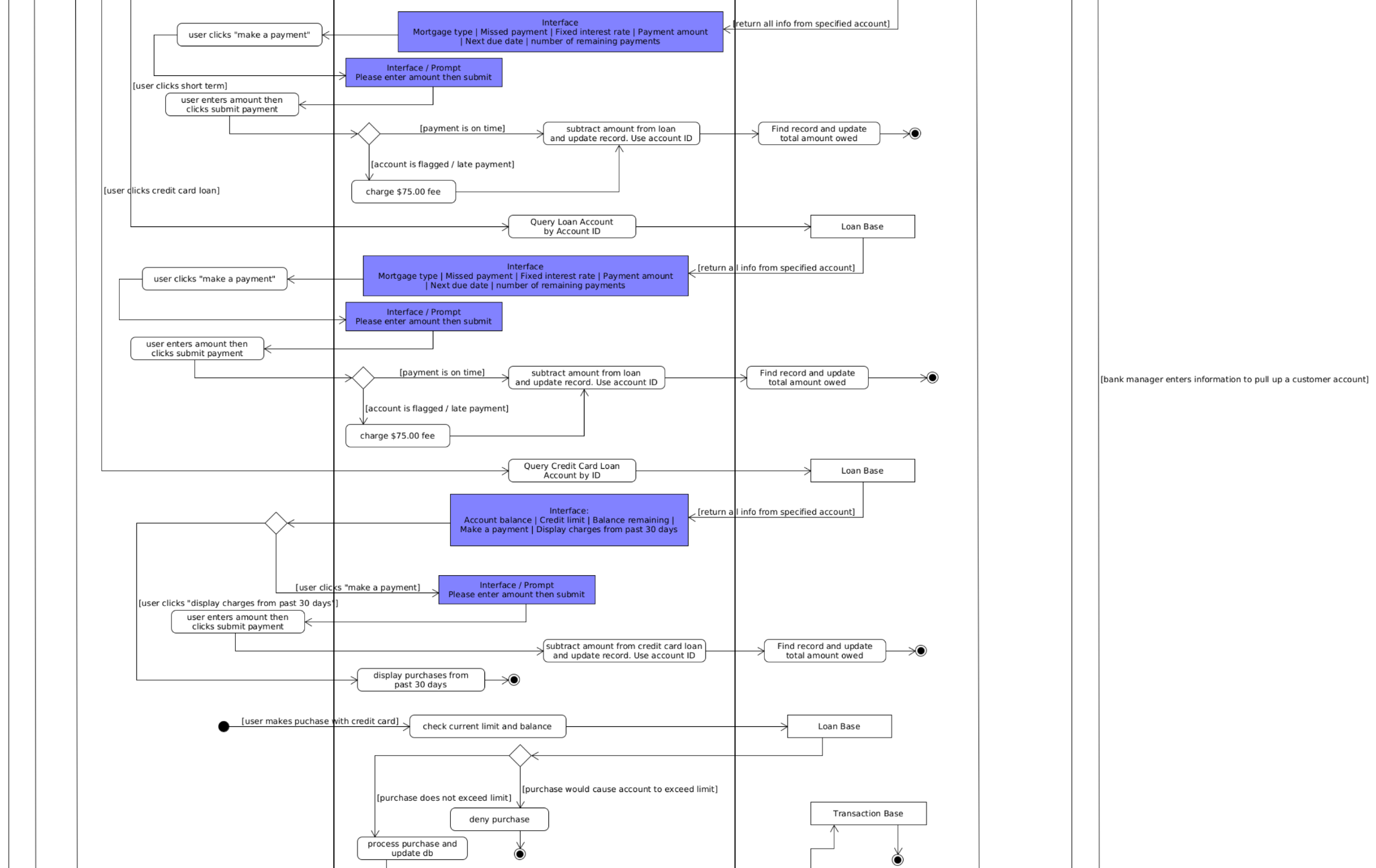


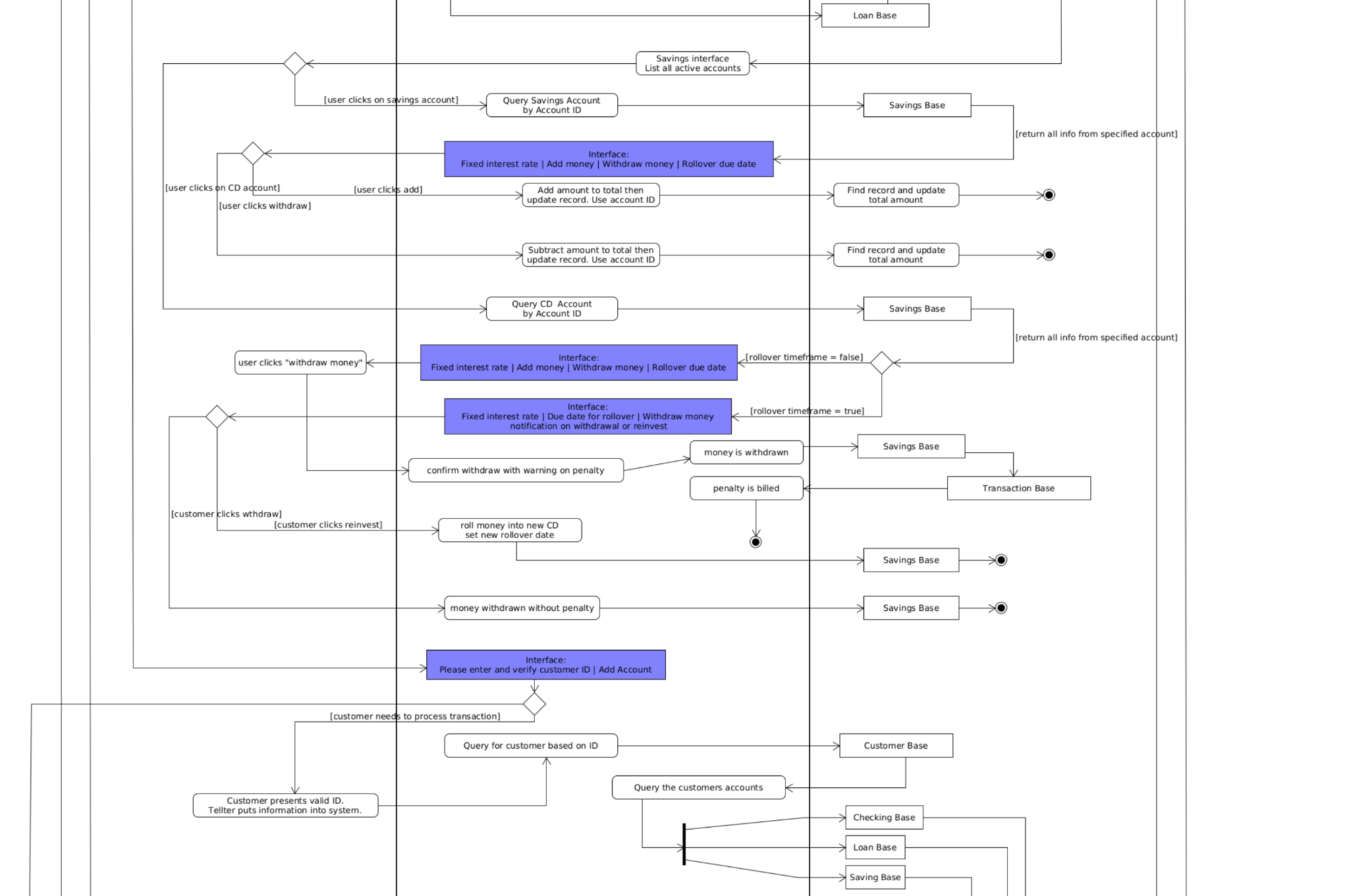
*Activity Diagram*

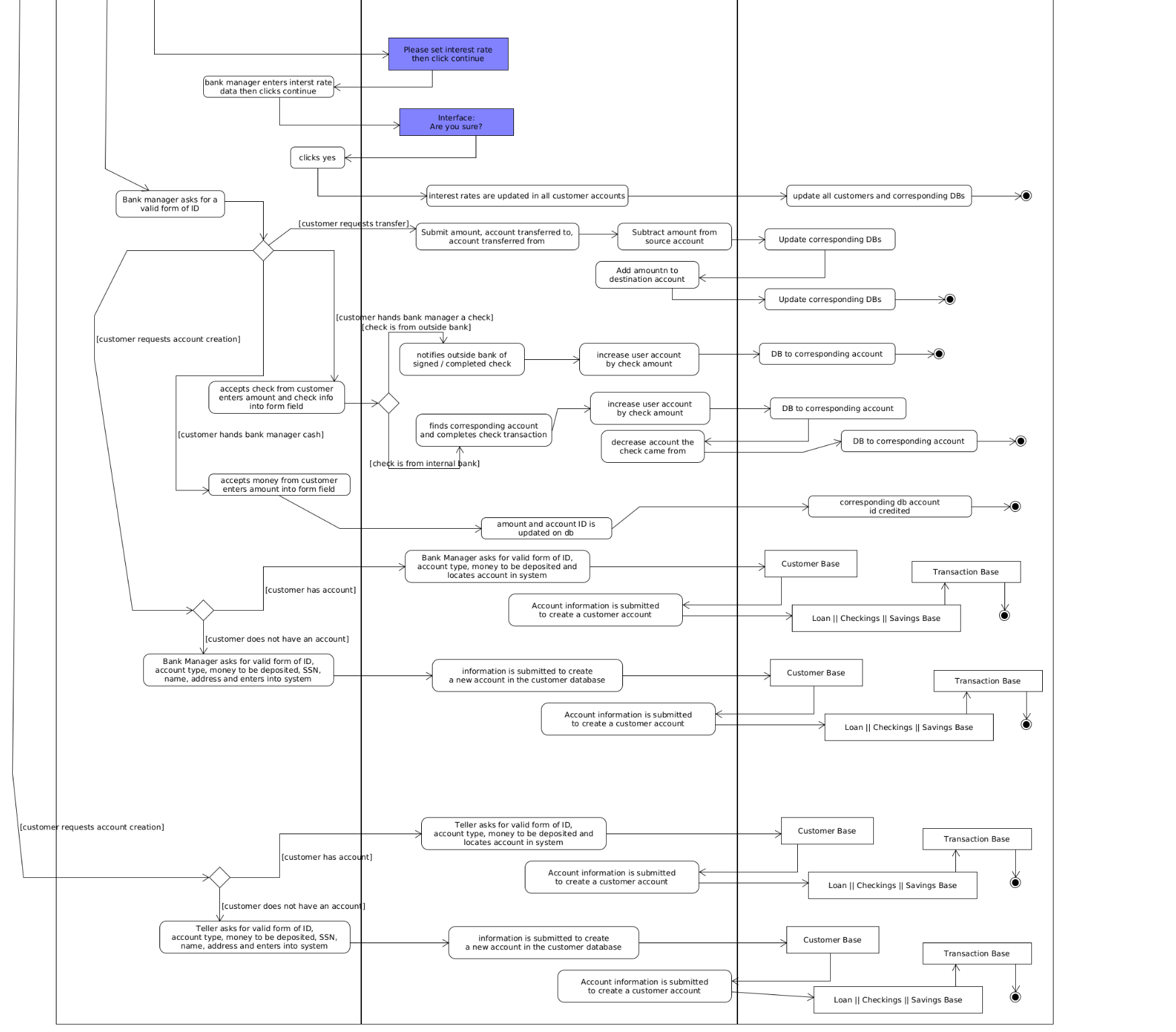
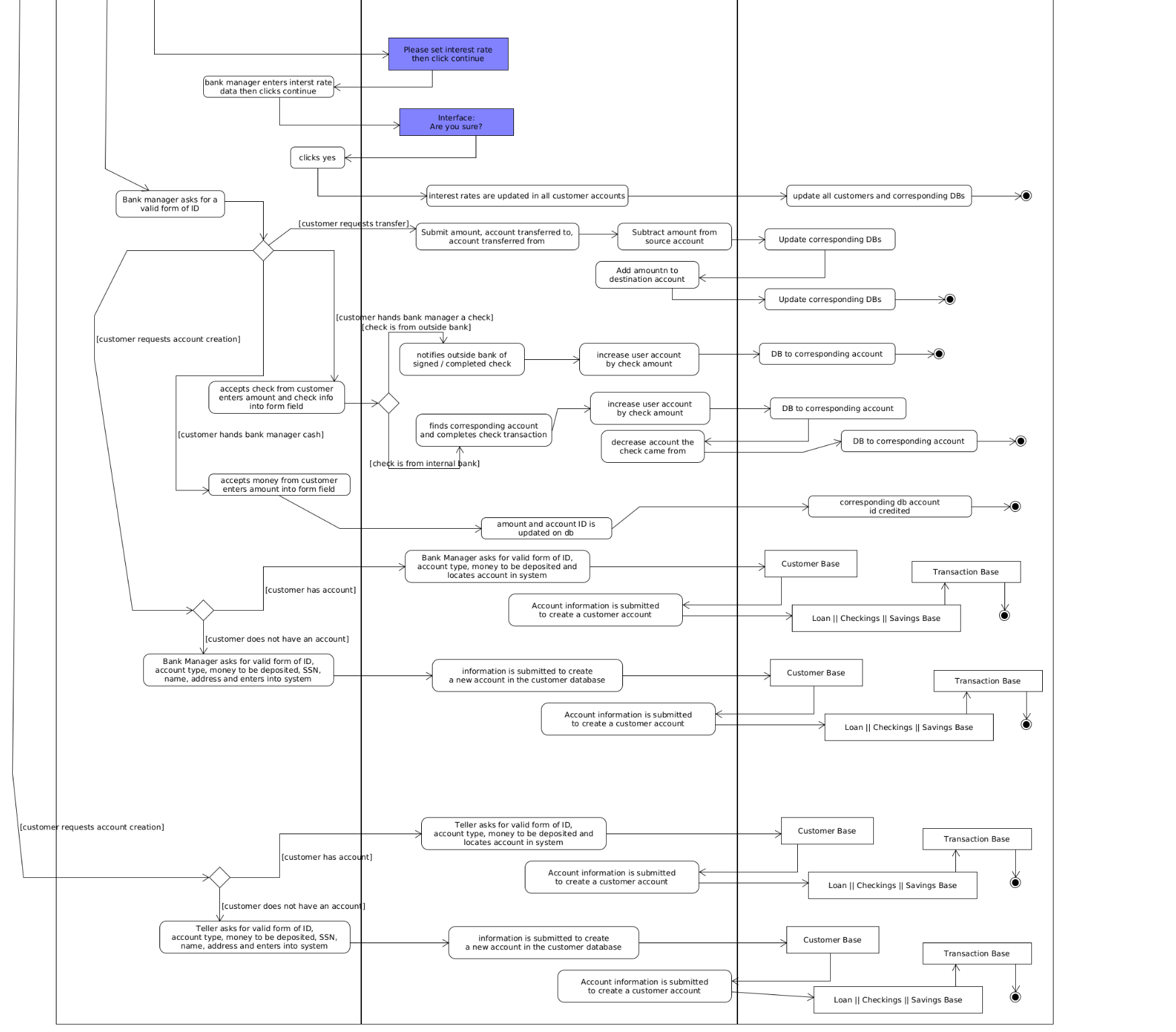
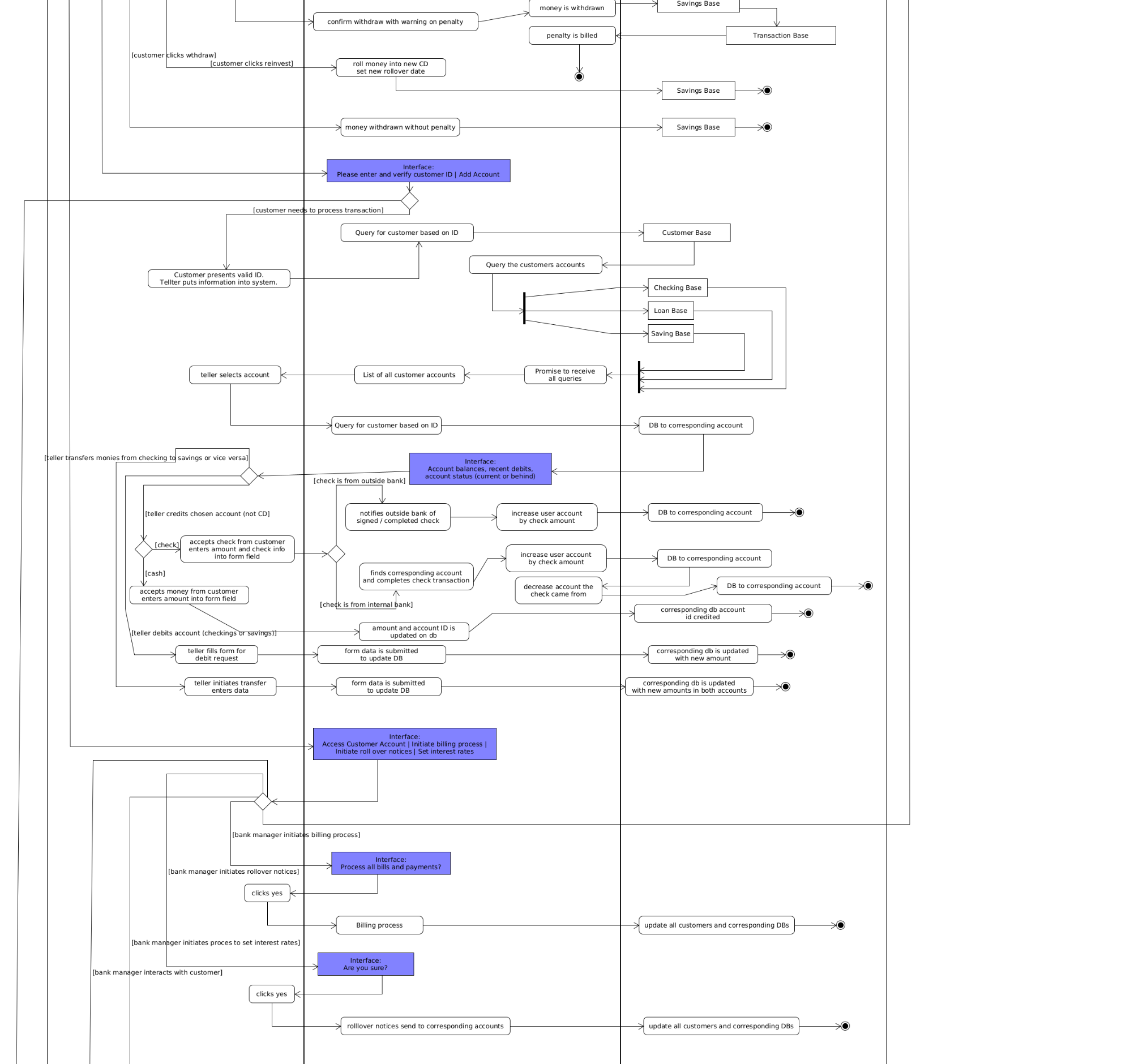


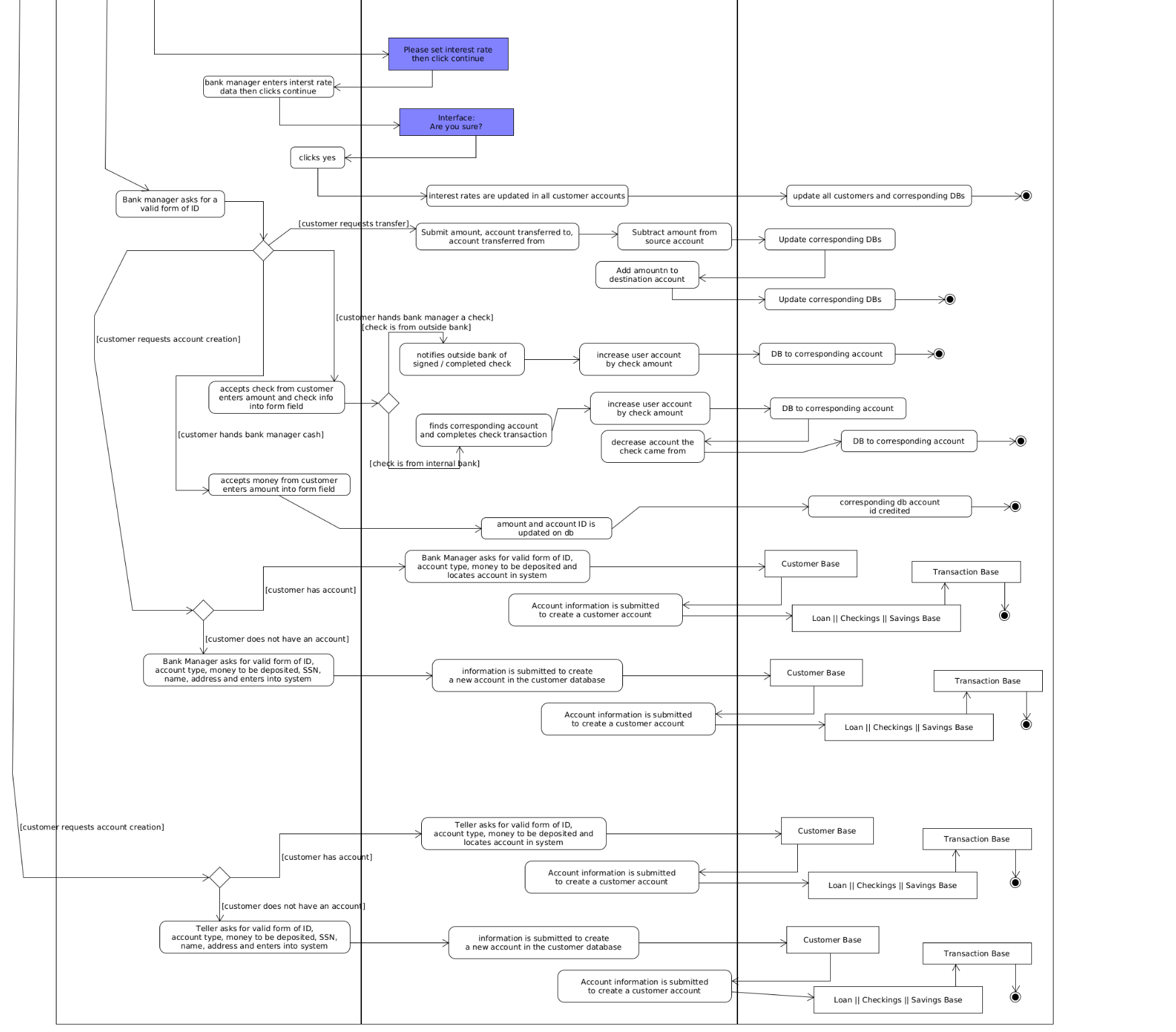






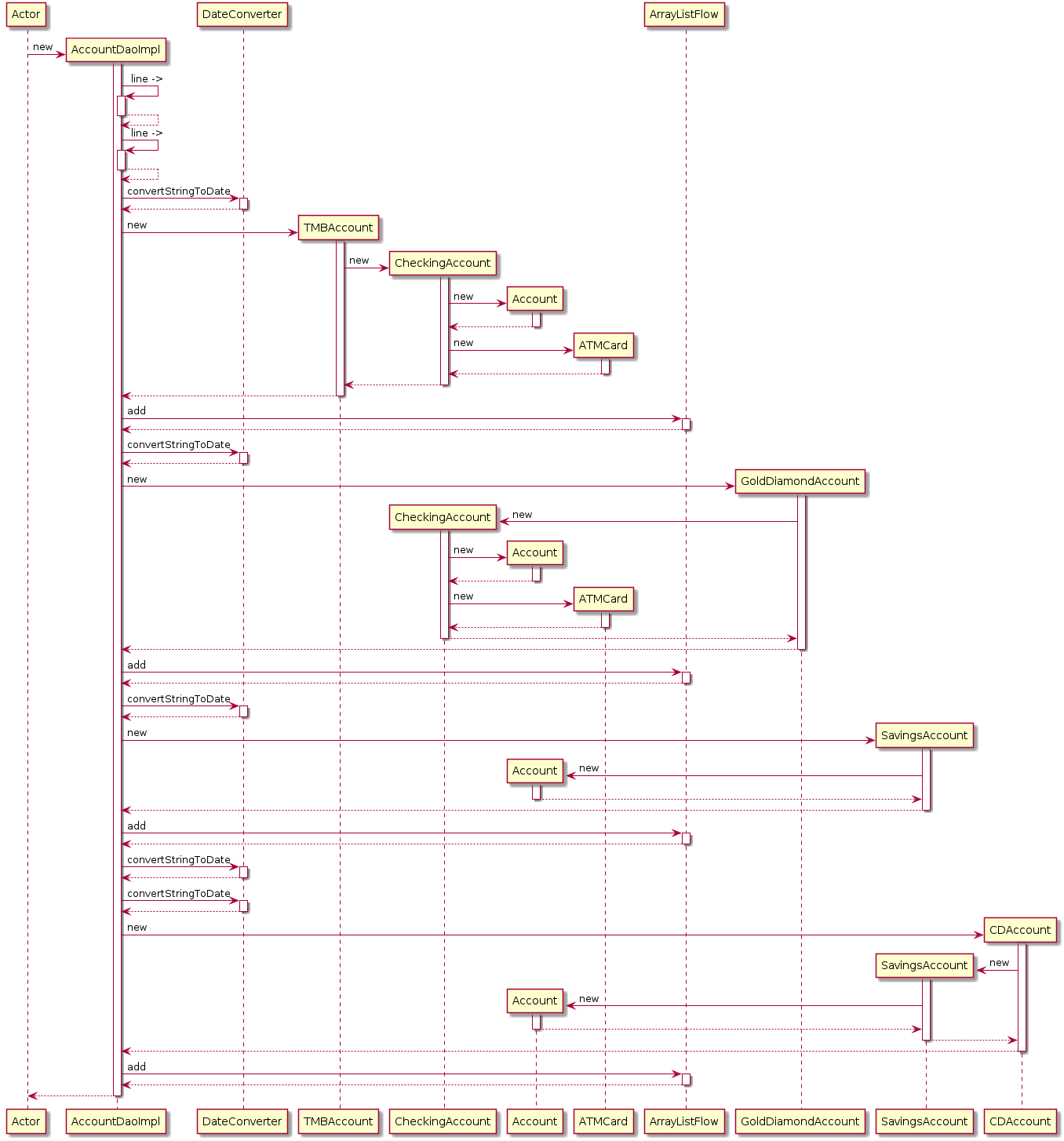




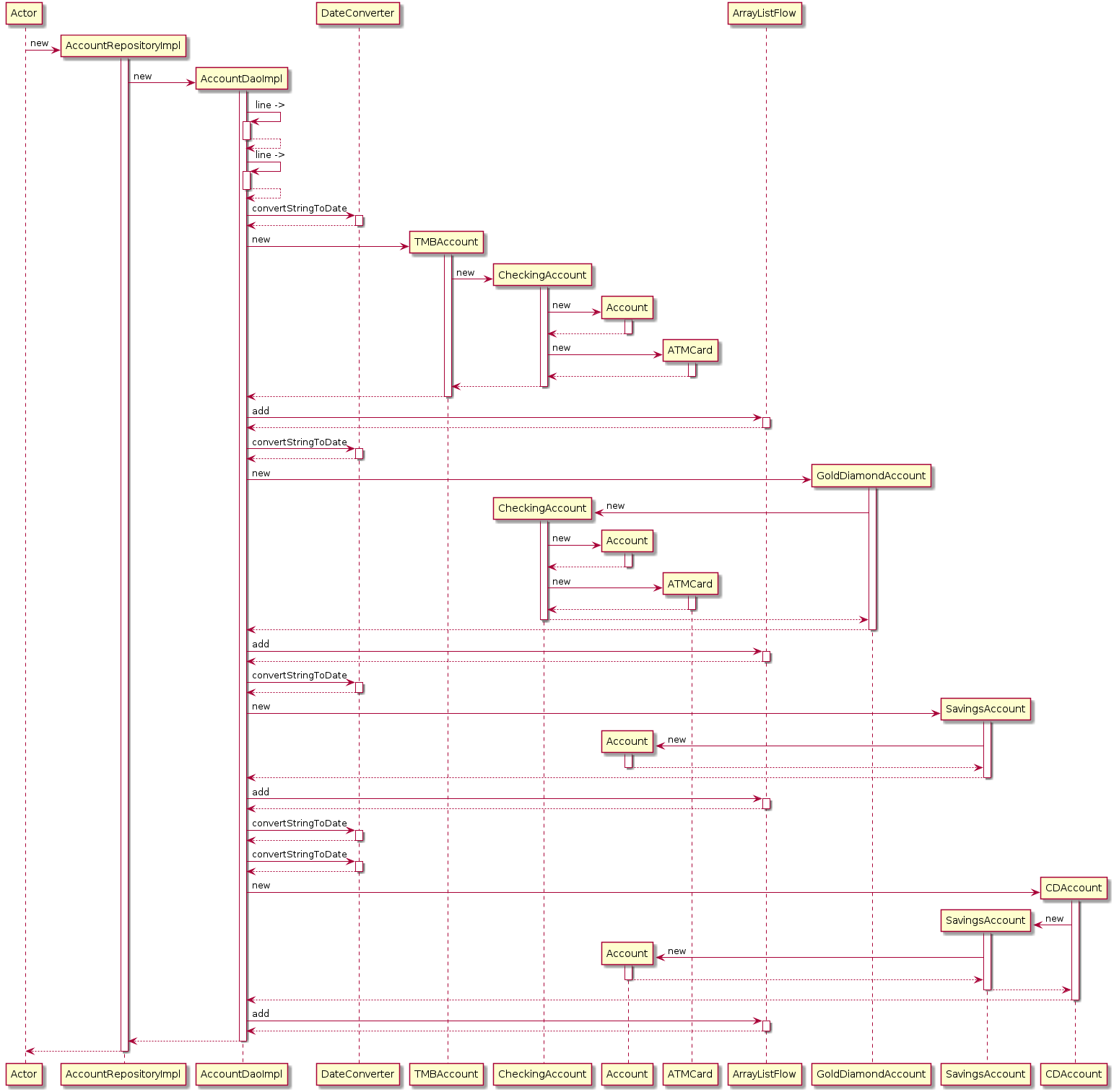


*Sequence Diagram*

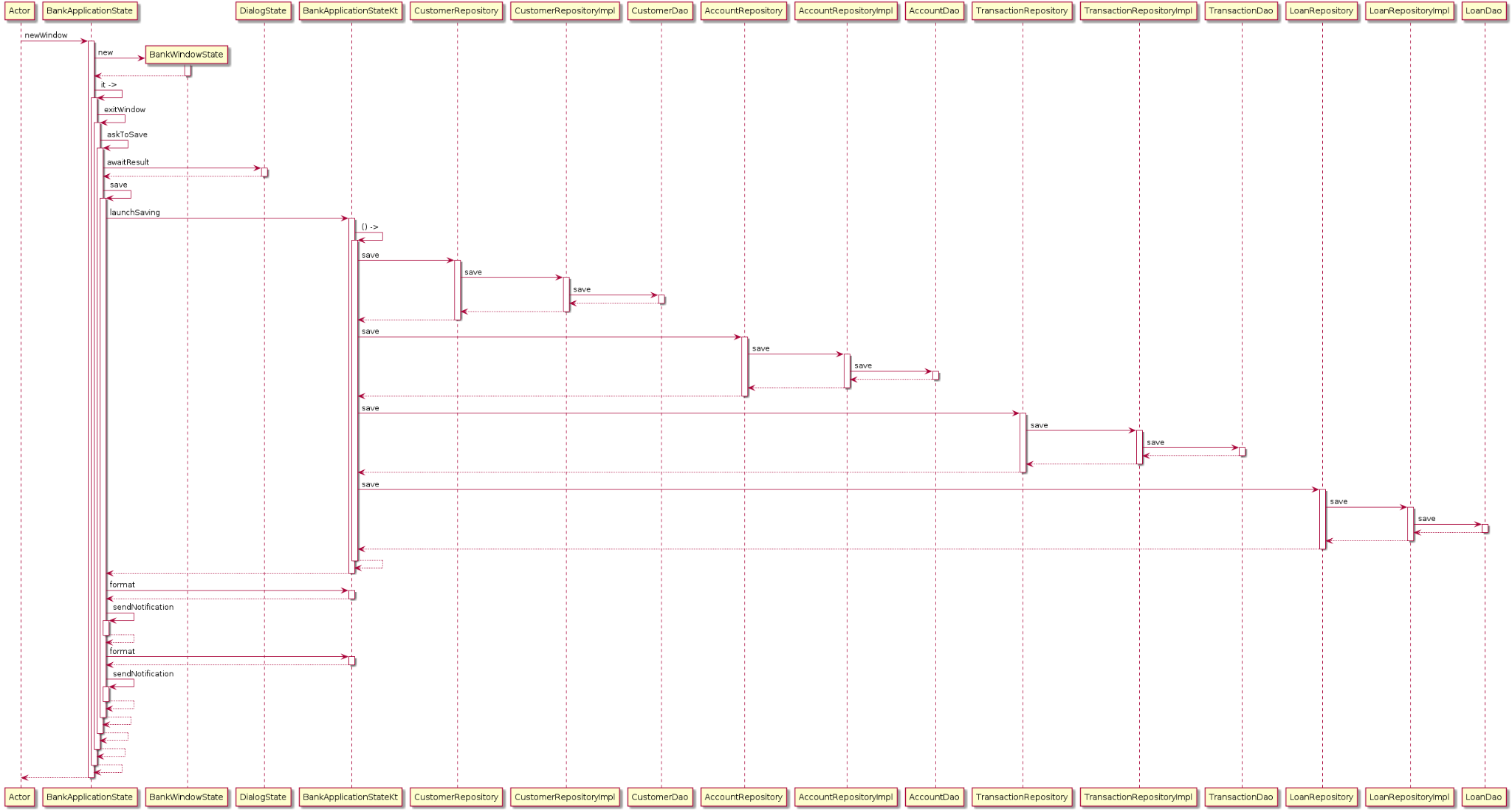
Account Dao Implementation

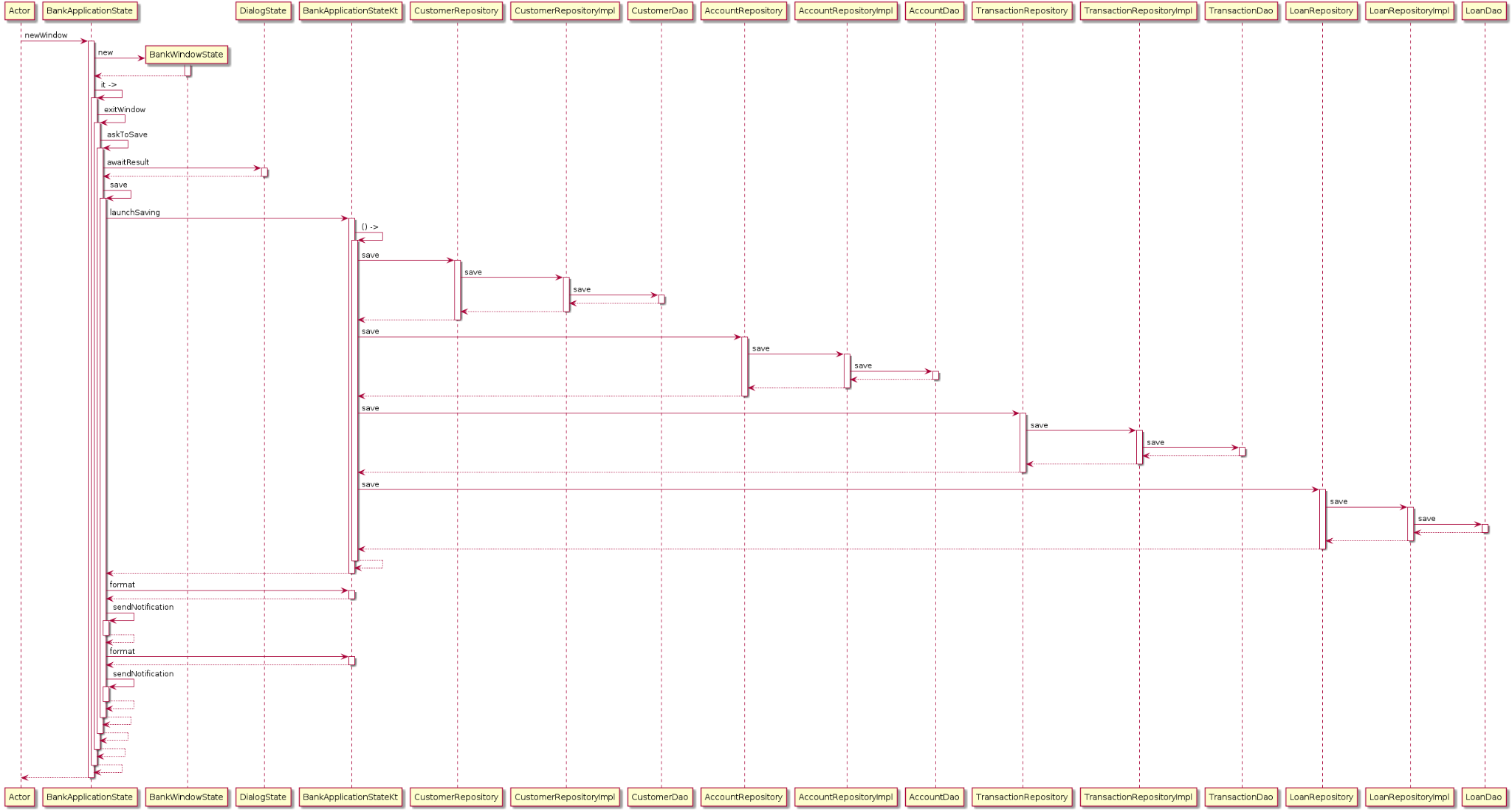


Account Repository Implementation



Bank Application State

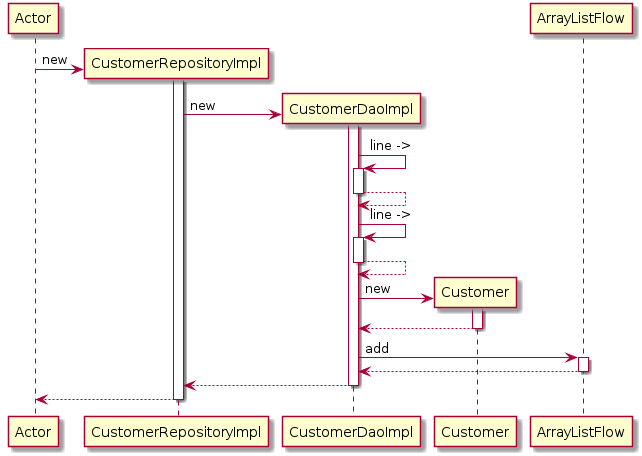




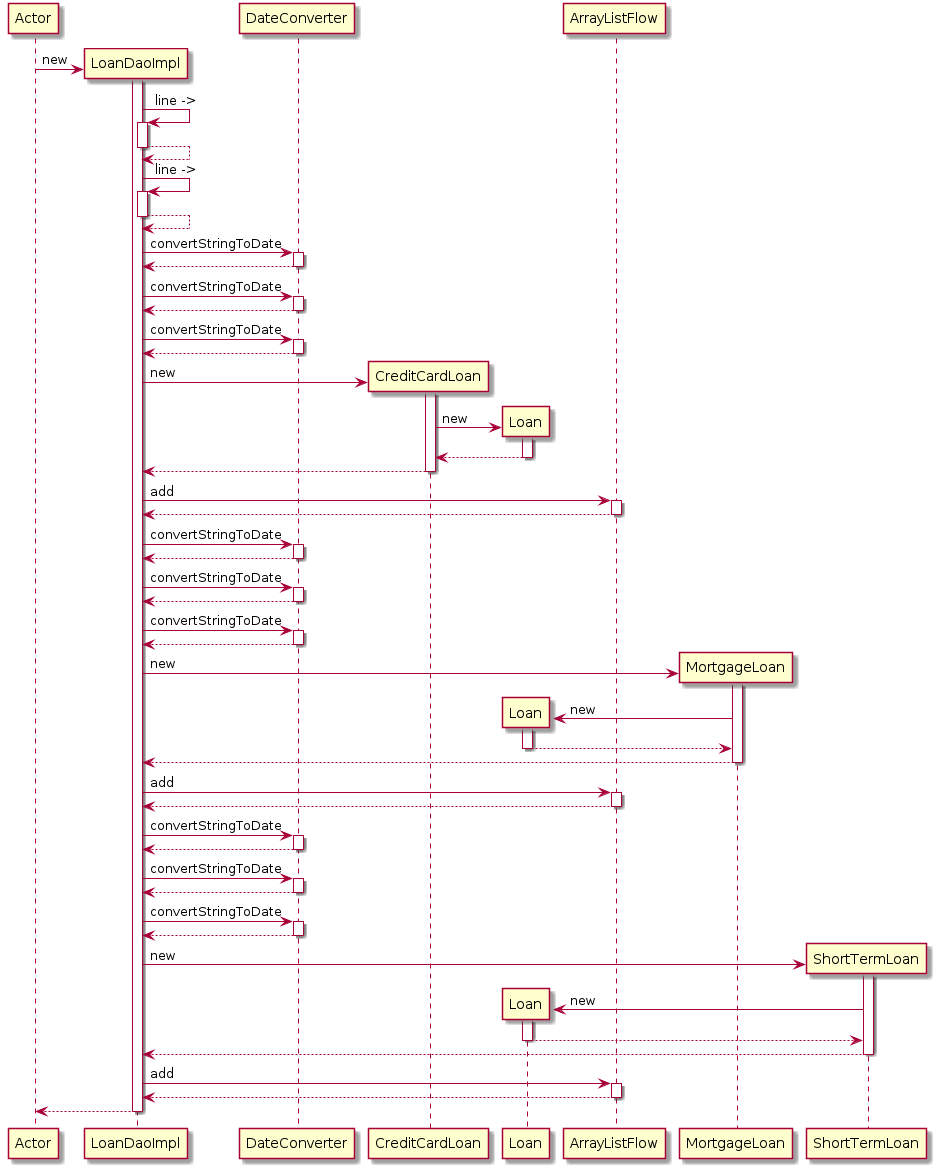
Customer Dao Implementation



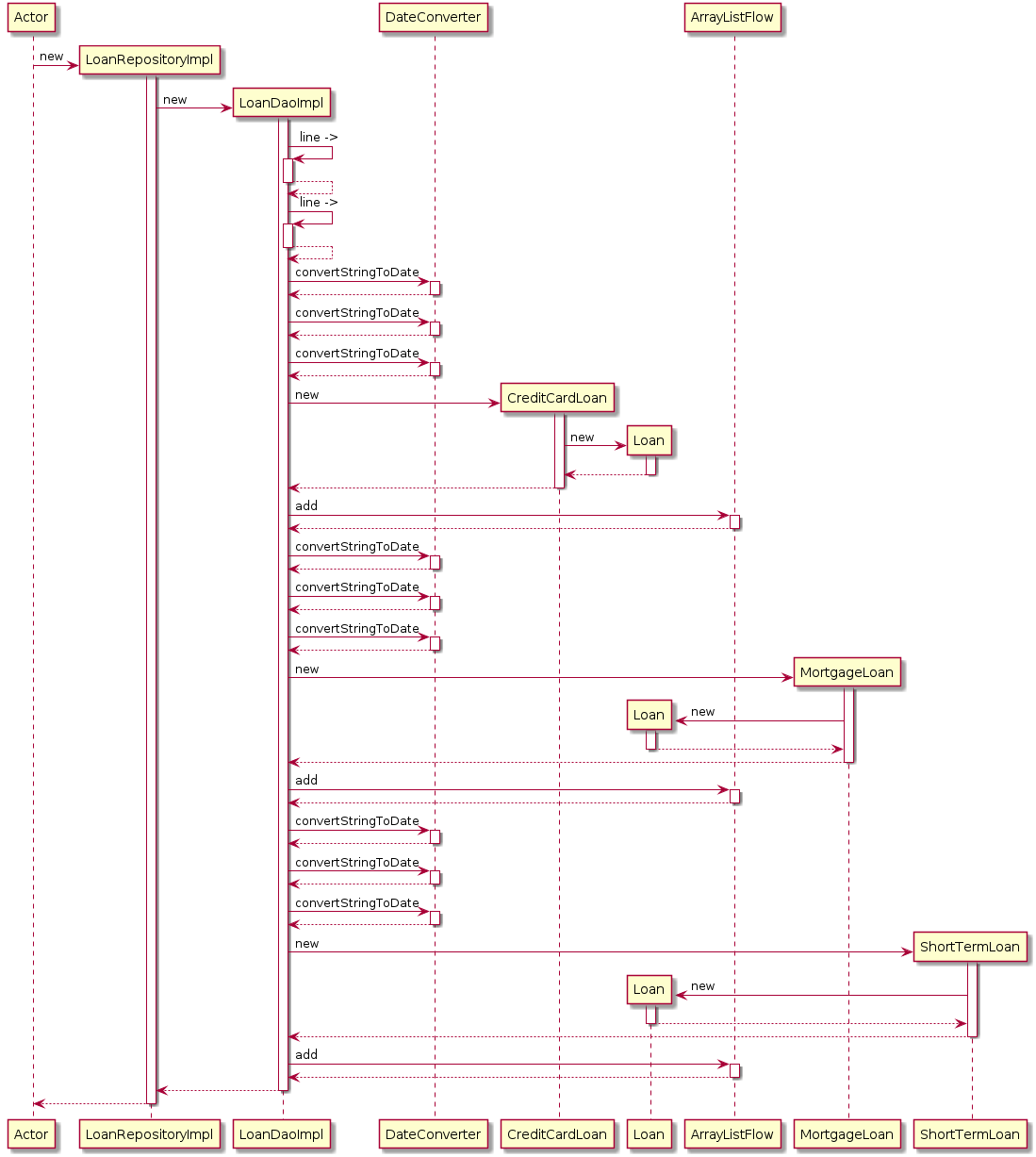
Customer Repository Implementation



Loan Dao Implementation



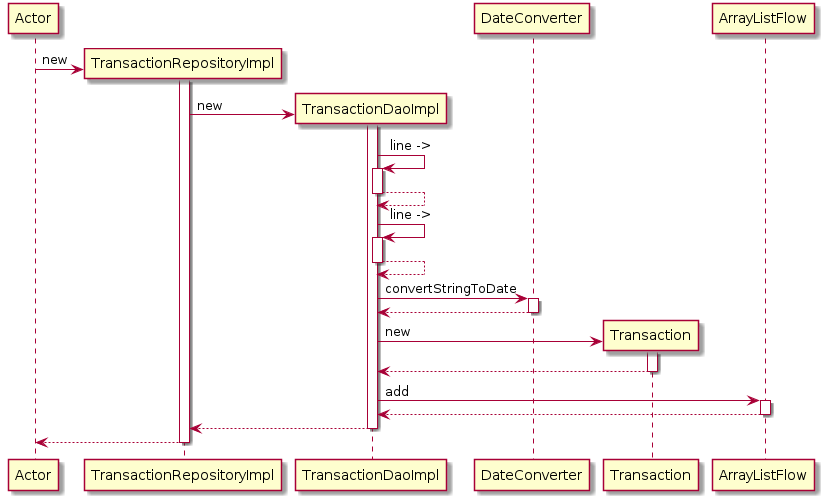
Loan Repository Implementation



Transaction Dao Implementation

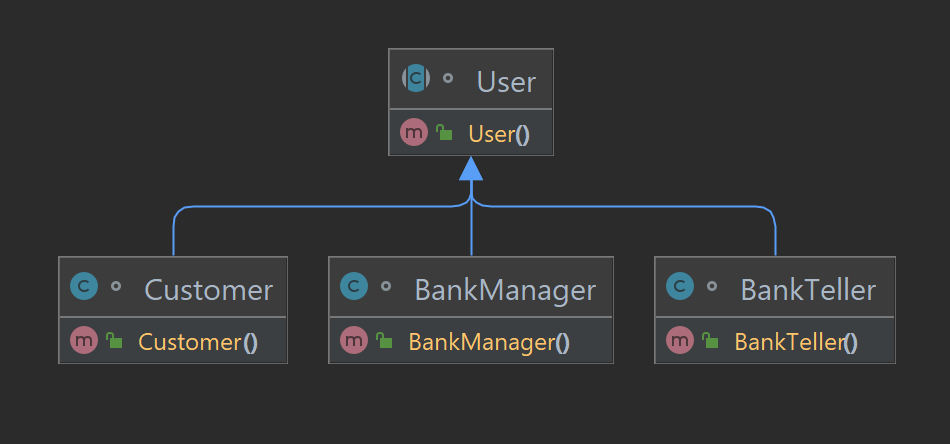


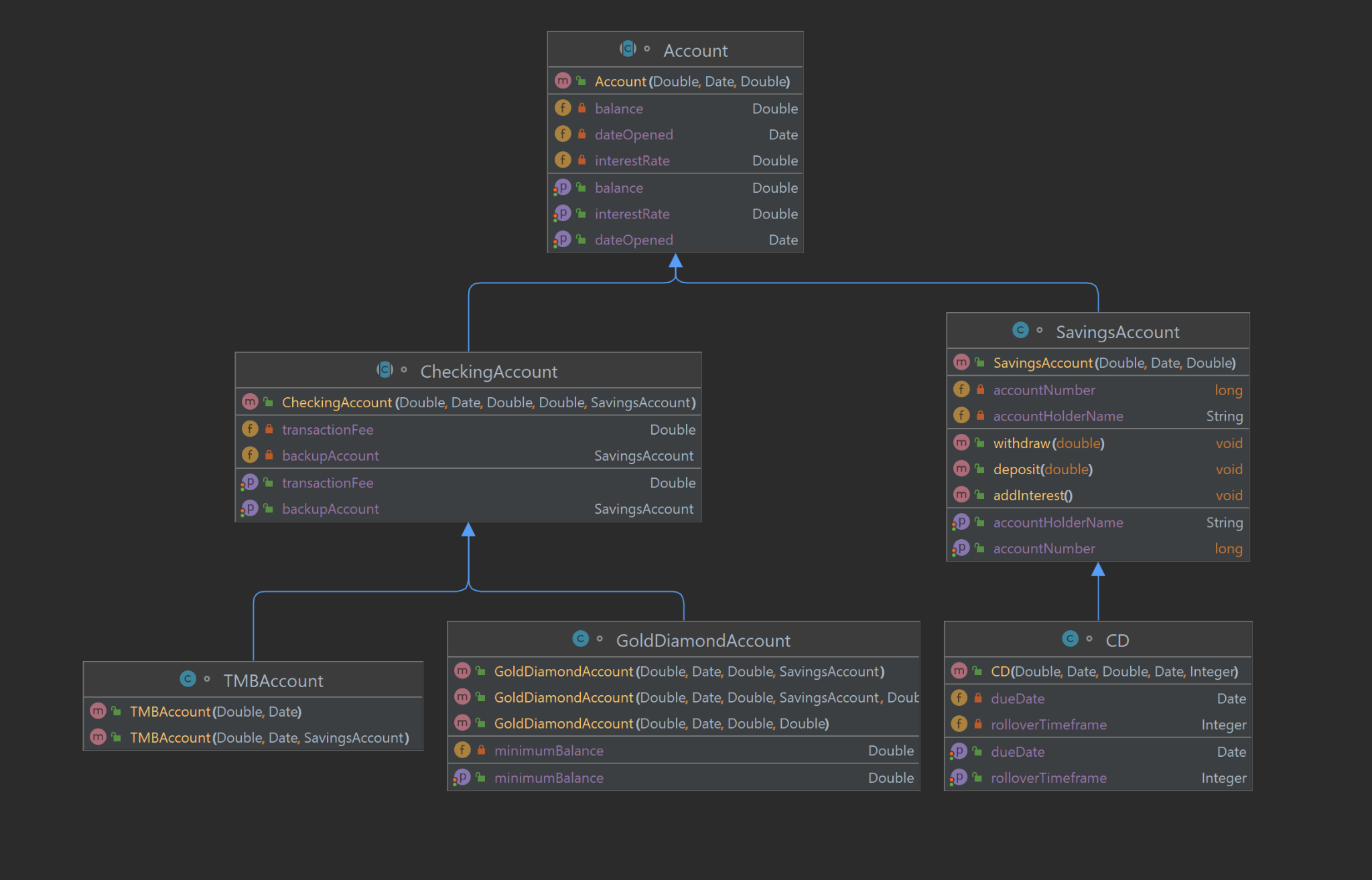
Transaction Repository Implementation

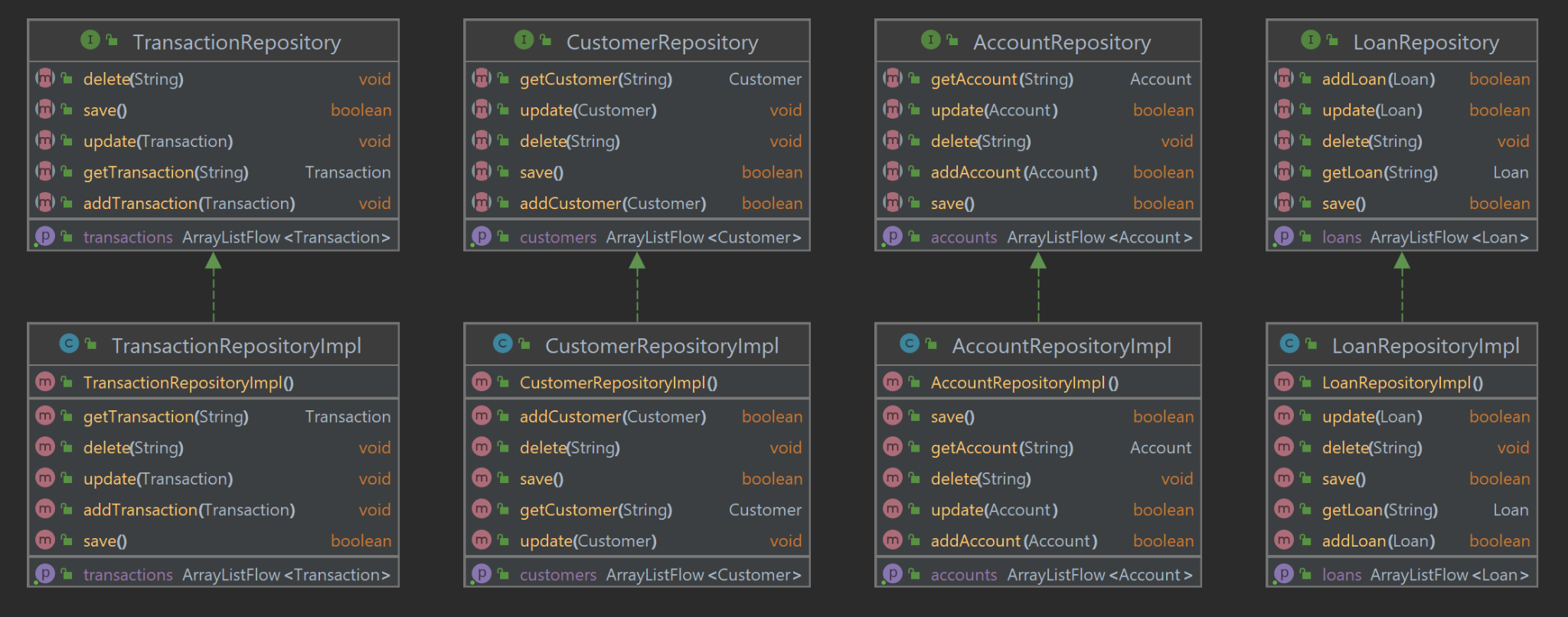
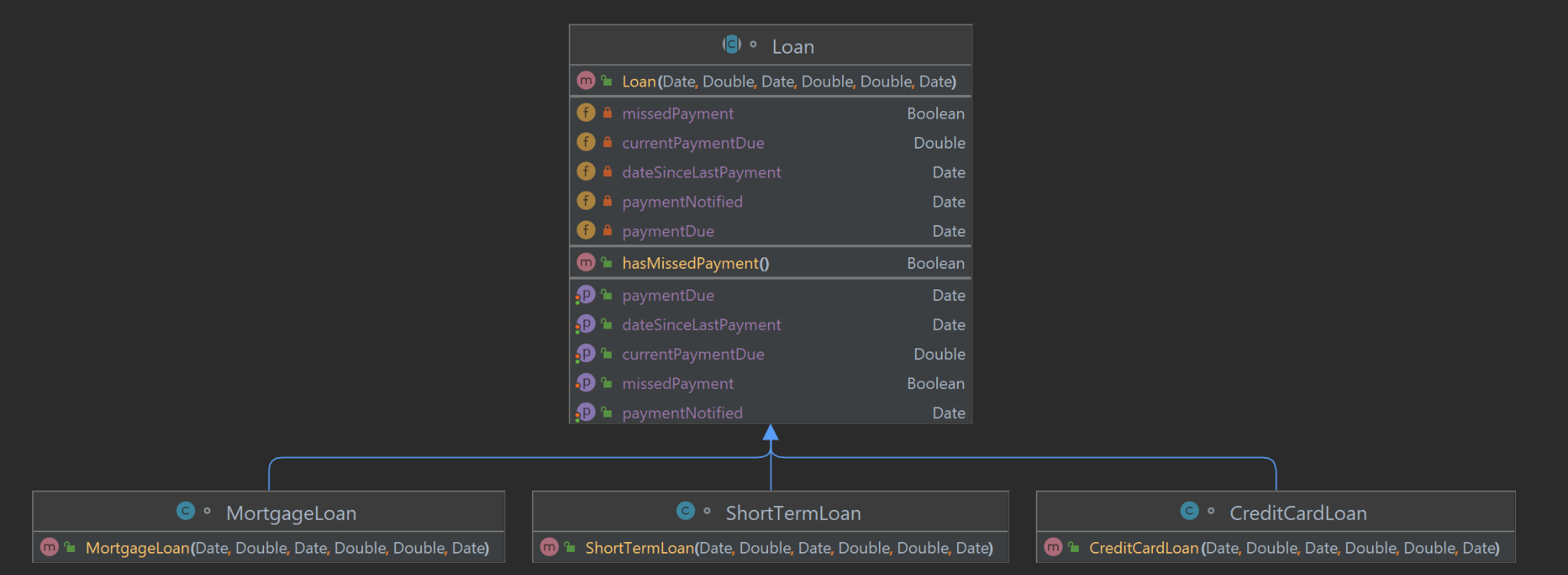


Class Diagrams

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.

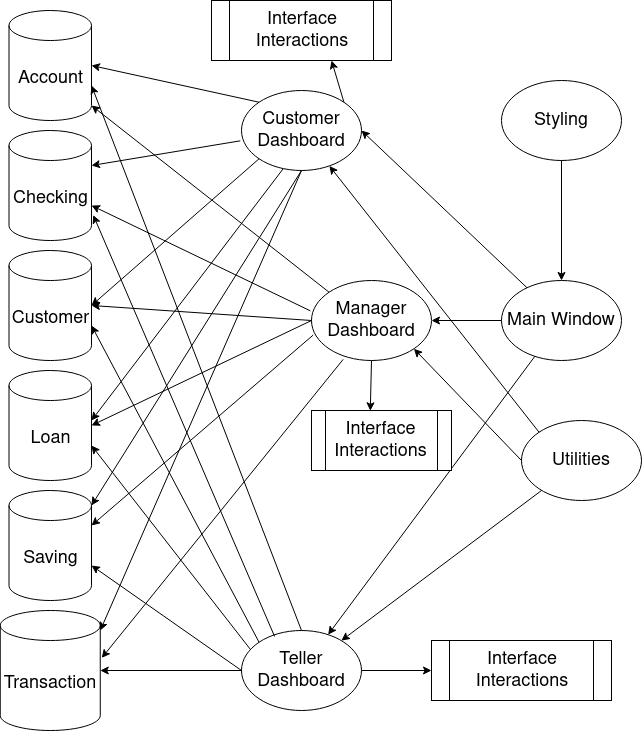






Architecture Diagram

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*

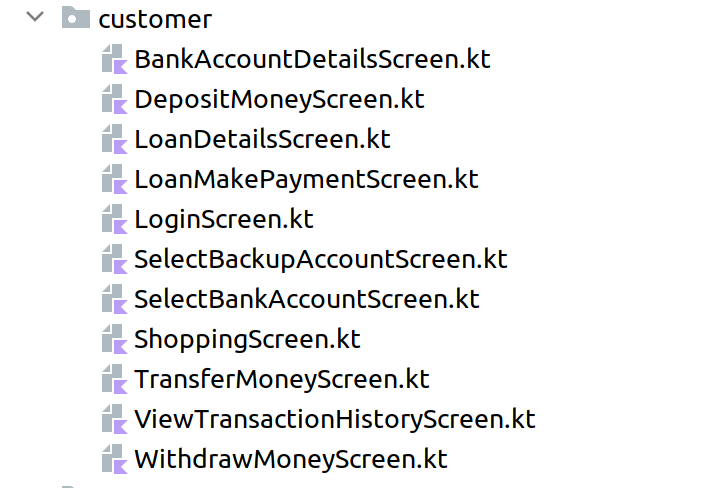
*/\*\*  
 \* This composable creates a window starting at our entry screen  
 \*/*@OptIn(ExperimentalDecomposeApi::class)@Composablefun BankWindow(state: BankWindowState) {

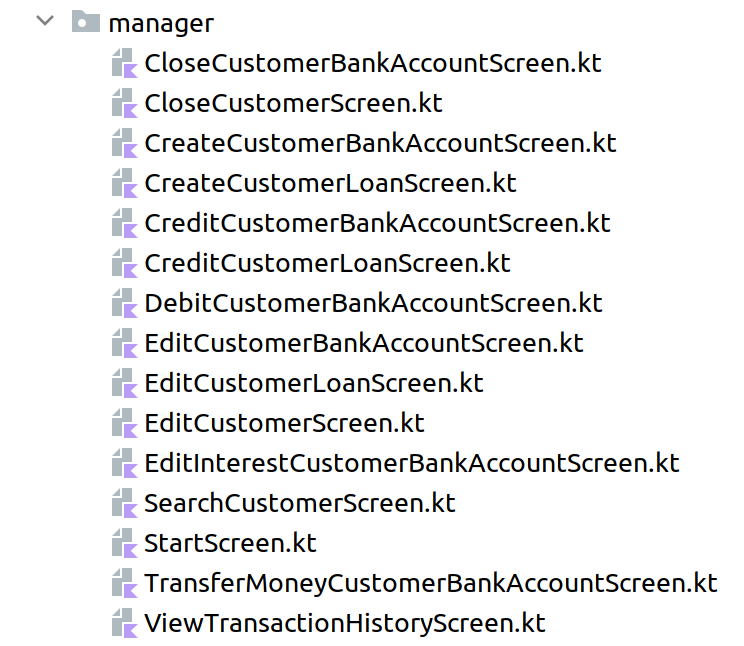
We also have a BankWindowState.kt to keep track of values in the BankWindow.

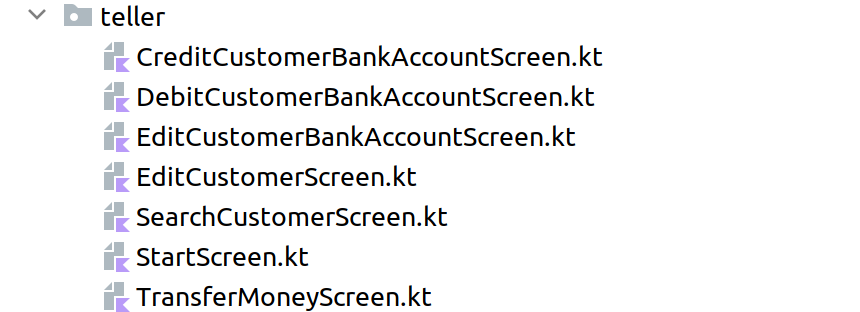
*BankWindowState.kt*

*/\*\*  
 \* 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.