Sanjana Urba
501173408
COE528 Section 5
March 17, 2024

# Bank Account Application Project Report

In this final project, a simple Bank Account GUI was created using Netbeans IDE and JavaFx. Further, use case diagrams and class diagrams were used, as well as data abstraction and the State design pattern to apply concepts learned throughout the course. This report will describe each aspect in relation to the code created.

The Use Case diagram models the functional behavior of the Bank Account System from the user's perspective. The actors are the users of the Bank Account which include the Customers and the Manager. The actors ‘LogIn’ to the system and must have the input credentials authenticated, hence the ‘includes’ relationship formed, because it is a mandatory action. An ‘extends’ relationship is formed in the case of ‘LoginFailure’, as it is an optional result that may not always occur. The Customer is able to ‘DepositMoney’ and ‘WithdrawMoney’ which must be validated, hence an ‘includes’ relationship is formed. The Customer can also make an ‘OnlinePurchase’ which forms an ‘includes’ relationship to ‘CalculateTotalCost’ and ‘ValidatePurchase’. Within these use cases, the price of the items along with the membership fee is calculated. If the transactions mentioned are successful depending on the account balance, the Customer undergoes an ‘UpdateFile’ indicated by an ‘extends’ relationship, and a generalization relationship is formed to specify an ‘UpdateLevel’ and ‘UpdateBalance’. The Customer is also able to ‘GetBalance’. The Manager is able to ‘AddCustomers’ and ‘DeleteCustomers’. To add customers, it must undergo ‘ValidateNewCustomer’ thus, an ‘includes’ relationship is formed. If successful, ‘CreateCutsomerFile’ occurs with their credentials, default balance and status, hence an ‘extends’ relationship is formed. If not, the ‘CutsomerAlreadyExists’ use case describes the error forming an ‘extends’ relationship. To delete a customer, it must ‘ValidateExistingCutsomer’ which is an ‘includes’ relationship. If successful, ‘DeleteCustomerFile’ occurs, removing the file along with their credentials and account information, unless the ‘CutsomerDoesNotExist’ occurs through an ‘extends’ relationship. Below, **Table 1** is used to specifically describe one of the use cases as followed by the template provided.

| | |
|---|---|
| **Name:** | DepositMoney |
| **Participating Actors:** | Customer |
| **Entry Conditions:** | ● The Customer is logged into their bank account. |

| | |
|---|---|
| **Flow of Events:** | 1. The Customer selects the option to deposit money.<br>    a. The System prompts user input of the deposit amount.<br>2. The Customer inputs the desired deposit amount.<br>    a. The System validates the requested deposit (ensures it is a positive number).<br>    b. The System updates the account balance to include the deposited amount, if the input is valid. Else, the account balance is not updated. |
| **Exit Conditions:** | ● The Customer has received confirmation that the account balance has been updated with the deposited money OR<br>● The Customer has received an error message stating the reason for unsuccessful deposit and is prompted to re-enter |
| **Quality Requirements:** | ● The update and error message are shown as soon as the amount is confirmed |

**Table 1: Use Case Description**

The Class Diagram models the static structural model of the Bank Account System. A total of 10 classes were used to organize the system; 4 classes to handle the main GUI components including user input fields, buttons and labels (LoginMenu, ManagerDashboard, CustomerDashBoard and OnlineShop), and 6 classes to handle specific functionalities of the components (Manager, Customer, CustomerLevel, SilverLevel, GoldLevel and PlatinumLevel). The LoginMenu is the initial class established when a user runs the program. This class contains variables and methods to initialize the stage containing all of the scenes and begin the user authentication processes. From here, depending on the username and password inputted by the user, ManagerDashboard or CustomerDashboard instances are created. Since the instantiation relies on the LoginMenu inputs, a **dependency relationship** is formed between each of the dashboards and the LoginMenu. The ManagerDashboard class initializes a new Manager instance and contains variables and methods to establish a new scene with specific user input fields and buttons. The Manager class handles the process of adding and deleting customers using text files. Thus, a **composition relationship** is formed since the MangerDashboard class encapsulates the functionality of the Manager class by controlling when and how the functionalities are used, hence ensuring that the Manager cannot operate independently without the presence of the ManagerDasboard. This interaction forms a strong aggregation as the MangerDashboard represents the whole containing the part which is the Manager. Similarly, the CustomerDashBoard class initializes a new Customer instance with variables and methods to establish a new scene with input fields and buttons controlling when functions are called. The Customer class handles the process of deposits, withdrawals, online purchases and getting the account balance, thus forming a **composition relationship**. The OnlineShop class is instantiated when the button within the CutsomerDasboard is clicked, and processes within the class occur depending on the Customer such as submitting purchases, resulting in updating balances/levels. Thus, a **dependency relationship** is formed. The abstract class of CustomerLevel uses a constructor to enforce a reference to the Customer object, forming a **composition relationship** with the Customer class. This indicates that the CustomerLevel cannot exist independently without the direct reference to the customer. Finally, the subclasses of SilverLevel, GoldLevel, PlatinumLevel formed an **inheritance relationship** with the abstract CutsomerLevel class as they implement methods in order for the purchase fees to be calculated, as well as transitioning of states following the State Design Pattern, which is discussed in further detail later.

The Customer class applies the aspects of data abstraction (to address criteria 2). The Overview clause highlights the purpose of the class. The Customer class represents a mutable bank account user with a username, password, account balance and a corresponding level. Customers can check their account information such as their balance and level. They are also able to deposit money, withdraw money, and purchase items from an online shop. The Abstraction Function provides a comprehensive understanding of what the class represents through which a customer has a username, password, role, level, and is implemented in the toString(). The rep invariant describes a set of conditions for the object's instance variables that must hold true. The Customer must have a username and password that is not null or empty, an account balance that is not negative, and a level describing the state representation either "silver", "gold" or platinum". The Customer class consists of multiple methods that allow the object to have numerous functionalities as mentioned in the Overview clause, thoroughly described using Requires, Modifies and Effects clauses. Along with updating account balance after every transaction using void methods, the Customer class also updates its file. This is done by locating the text file through the directory and username, and updating the parts of the file (particularly the balance and string of the level). The getters of getUsername(), getLevelString(), and getBalance() are used to display the information for customers to see and are found by reading the updated file as changes occur during runtime. There are also two setters that are used to set the post purchase balance and set a new level as it changes due to the customer's balance.

The State Design Pattern is applied in the CustomerLevel abstract class within the UML Class Diagram. The CustomerLevel consists of three concrete subclasses that represent the different states a Customer can be in. This includes SilverLevel, GoldLevel and Platinum Level. The Customer is passed as an argument to the CustomerLevel as well as the subclasses to determine its balance and set the state after changes are made to the balance. The transition of the Customer's state is handled by each subclasses implementation of the updateState() method, after any transaction has been made. A balance of less than $10000 is assigned to SilverLevel; a balance of $10000 or more, but less than $20000 is assigned to GoldLevel and a balance of $20000 or more is assigned to PlatinumLevel. According to which state the Customer is in, the matching subclass has a different implementation of the onlinePurchase() method that accounts for different purchase fees (SilverLevel of $20 fee, GoldLevel of $10 fee and PlatinumLevel of $0 fee). Thus, by encapsulating the Customer's behavior of state transitioning and online purchase calculations specific to each state, the CustomerLevel Class effectively demonstrates the State Design Patterns ability to manage internal state changes.