COE528 Project: Bank Application
Name: Fareez Mir
Student ID: 501159472
Section: 5

Section 1.0 - Use Case Diagram

The use case diagram for the bank application showcases the interactions between users and the system. The application consists of two actors: the Manager and the Customer. The Manager has the capabilities to “AddCustomer” and “DeleteCustomer,” which are operations to create a customer and delete a customer record, respectively, along with “Login” and “Logout” of the system. Likewise, Customers can “Login” and “Logout” of the system but have additional operations different from the Manager. Operations include “ViewBalance” to view their balance, “DepositMoney” to deposit their earnings, “WithdrawMoney” to retrieve their earnings, and “OnlinePurchase,” to purchase an item. The “DepositMoney” and “WithdrawMoney” have an *include* relationship with “ValidateDeposit” and “ValidateWithdrawal,” respectively, to check if the amount specified can be deposited/withdrawn. If successful, the *extend* relationship of “UpdateCustomerFile” updates their file with generalization arrows of updating level and balance to reflect changes to their account. A similar structure is used for “OnlinePurchase,” which *include* “CalculateTotalCost” to calculate the total cost of the item based on the fee (determined by the Customer’s level) and the price of the item and *include* “ValidatePurchase” to check if the Customer has the sufficient funds to purchase an item. “ValidatePurchase” can then optionally *extend* to “UpdateCustomerFile,” and as before, updates the Customer’s file by updating their balance and level accordingly.

Section 1.1 - Use Case Description: DeleteCustomer

Use case name:	DeleteCustomer
Participating actors	Initiated by Manager
Flow of events	1. Manager selects the option to remove a customer from the system. 2. Manager fills out the form by filling in the Customer username to delete. 3. The system validates the existence of the Customer file. 4. If the file exists, the system deletes the Customer's file. 5. System informs the Manager if the operation was successful or not
Entry Condition	• Manager must be logged in and have accessed the Manager Dashboard's "DeleteCustomer" option
Exit Condition	• Customer's data has been removed from the system, with appropriate notice to the Manager, OR • Manager has received an explanation indicating why the the operation could not be processed.

Section 2.0 - Class Diagram

The class diagram for the bank application outlines the structure and relationships of classes within the system. It contains the classes "Customer", "CustomerState", and "Item" alongside the scenes "CustomerScene", "ManagerScene", "LoginMenu", and "ShoppingScene", with "SceneHub" associating with all of them as the primary controller to display the primary scenes. Each scene implements the "Dimensions" interface to maintain consistency with the GUI dimensions. The "Customer" class aggregates state objects obtained from the "CustomerState" interface, which "CustomerSilver", "CustomerGold", and "CustomerPlatinum" all implement, representing the different levels of Customer accounts. These sub-states embody the State design pattern, which shows Customer behaviour depending on their level. The "ManagerScene" aggregates the "Manager" class to utilize functions to perform Manager specific tasks. Likewise, the "CustomerScene" aggregates the "Customer" class, utilizing its functions to read and write the Customer data from a text file and perform operations based on the file's data. Finally, "ShoppingScene" aggregates "Item" and "Customer" to manage the items displayed on the scene, along with handling any updates made to the "Customer" based on the items purchased. Finally, the main "BankApplication" uses an association relationship with "LoginMenu" to demonstrate that its scene acts as the application's start point; to display a login authentication menu to input credentials, which, depending on the credentials, displays the correct scene using "SceneHub".

Section 2.1 - Class Diagram Description: Customer

A key class within the Class Diagram is the “Customer” class, representing the functionalities of the Customer within the bank application. This class holds the calculations responsible for handling transactions when purchasing items online, handling withdrawals and deposits, and updating the customer’s data when necessary. Within the class, it uses 6 main variables: username, password, level, balance, customerData, and levelState. The two vital variables used in the class’ structure are customerData and levelState respectively. customerData reads the data from the corresponding text file of the Customer, and levelState dynamically changes the Customer’s level depending on events, maintaining the state design pattern in the system. Finally, levelState is used in the setLevelState() method, which updates the Customer’s level based on the balance from the substates, and is first used to set the initial level of the Customer.

Through the class’ design, it ensures that the Customer’s data is in line with the system, enforcing that their balance is not negative, having their credentials checked to be valid, and checking valid membership levels. These check’s are verified using the “repOK()” method, representing the representation invariant, which returns false if any of these checks are unusual with respect to the system (i.e. if the balance is negative, credentials are null or empty, and levelState is not null). The implemented “toString()” function, representing the abstraction function, highlights the information of the Customer, being their username, password, and level.

Section 3.0 - State Design Pattern in UML Class Diagram

The State Design Pattern within the UML Class Diagram that forms the State Design Pattern is the “CustomerState” interface, along with the sub-states “CustomerSilver”, “CustomerGold” and “CustomerPlatinum” which all inherit its methods. The classes each have the primary function, “updateLevel” to dynamically transition the level of the customer based on the customer’s balance. This is done by setting the level state to a new state object, transitioning up or down the level hierarchy depending on their balance. This exemplifies the state design pattern as the function directly alters the behaviour of the “Customer” object, since, depending on the level, changes the fee that is required by the Customer when purchasing an item. A Customer with a Silver level may require a fee of $20, but a Customer with a Platinum level requires no fee at all. Thus, the object, “Customer”, depends on its state, and changes its behaviour at run-time depending on the state.