COE528 (Winter 2024) - Project

Emilio Messina
501178927
Section 42

A Use Case Diagram visually represents the interactions between different user roles (actors) and a system. In the Figure 1 below, there are two primary actors: "Manager" and "Customer." The manager can login, logout, add customers, delete customers and these interactions are represented by the ovals known as the use cases. On the customer side, there are several use cases: "Login" (for accessing accounts), "Logout" (after using the system), " Balance" (shows account balance), "Achieve Level" (likely related to loyalty or rewards), "Make Purchase" (transaction), "Withdraw" (funds), and "Deposit" (add money to accounts). The lines connecting them to the user figures illustrate the functionalities available to each user type within the system. Overall, the Use Case Diagram provides a high-level view of how different actors interact with the system's features.

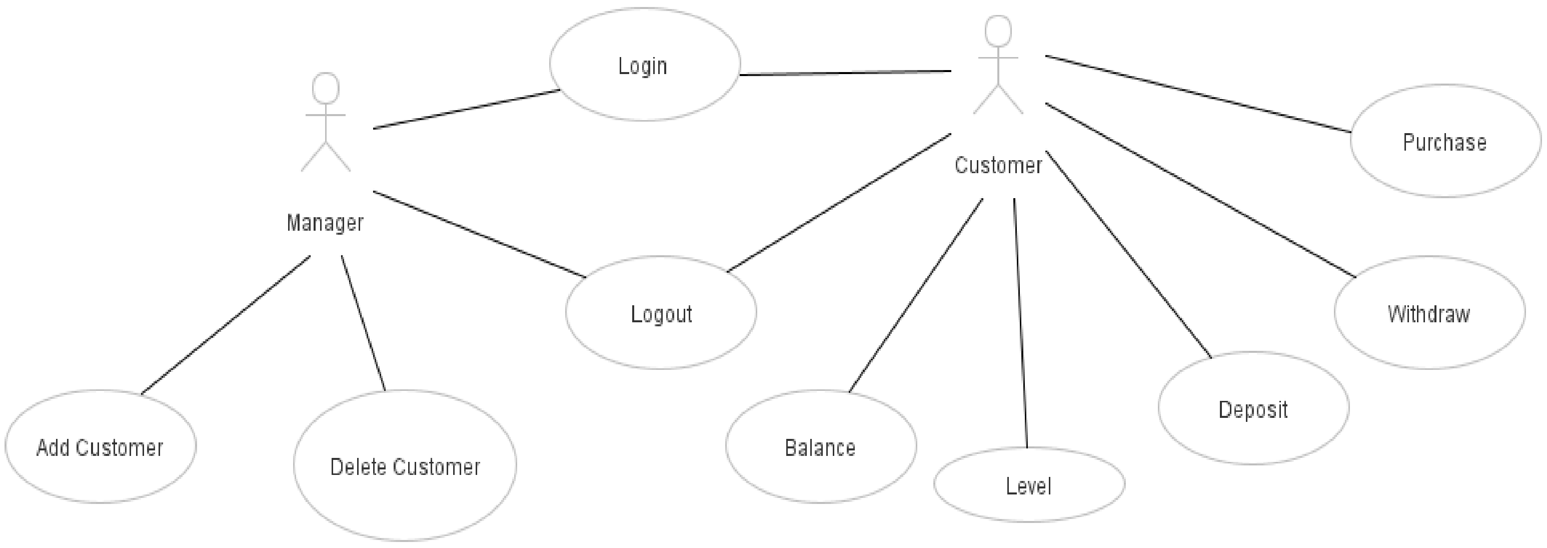

Figure 1: Use Case Diagram

Description of Add Customer Use Case:

1. Name: Add Customer
2. Participating actor: Manager
3. Entry condition: Manager is logged in
4. Exit condition:

- Manager must enter a name and password for the customer

- Customer created in customer.txt

5. Flow of events:

- Manager logs his/hers account
- The program displays a list of options “Add Customer”, “Delete Customer” and “Logout”.
- The program asks the manager for the customers username and password for their account.
- Customer is then created in customer.txt

Class Diagram diagrams illustrate several classes within the system listing their name attributes, methods and accessibility. First, we have the BankAccountApplication class, which manages the overall application and system. Next, the Manager class represents the system’s managers, responsible for managing customers and accounts. The Customer class represents individual customers, with attributes like “username” and “password”. Customers can retrieve account details, update information, and check balances. The BankAccount class represents individual bank accounts, including attributes such as “accountNumber”, “balance”, and “accountType”. Associated methods allow depositing, withdrawing, and transferring funds.

Finally, the States (SafeState, PlatinumState, GoldState, SilverState) classes indicate different states a customer’s account can be in, each with specific attributes and methods related to account features based on the customer’s status. The arrows connecting classes show associations or dependencies, relationships and responsibilities within the banking application system. Open arrow solid lines represent associations while dotted lines indicate that it is dependent on another class. Closed arrow solid line represents inheritance from another class.

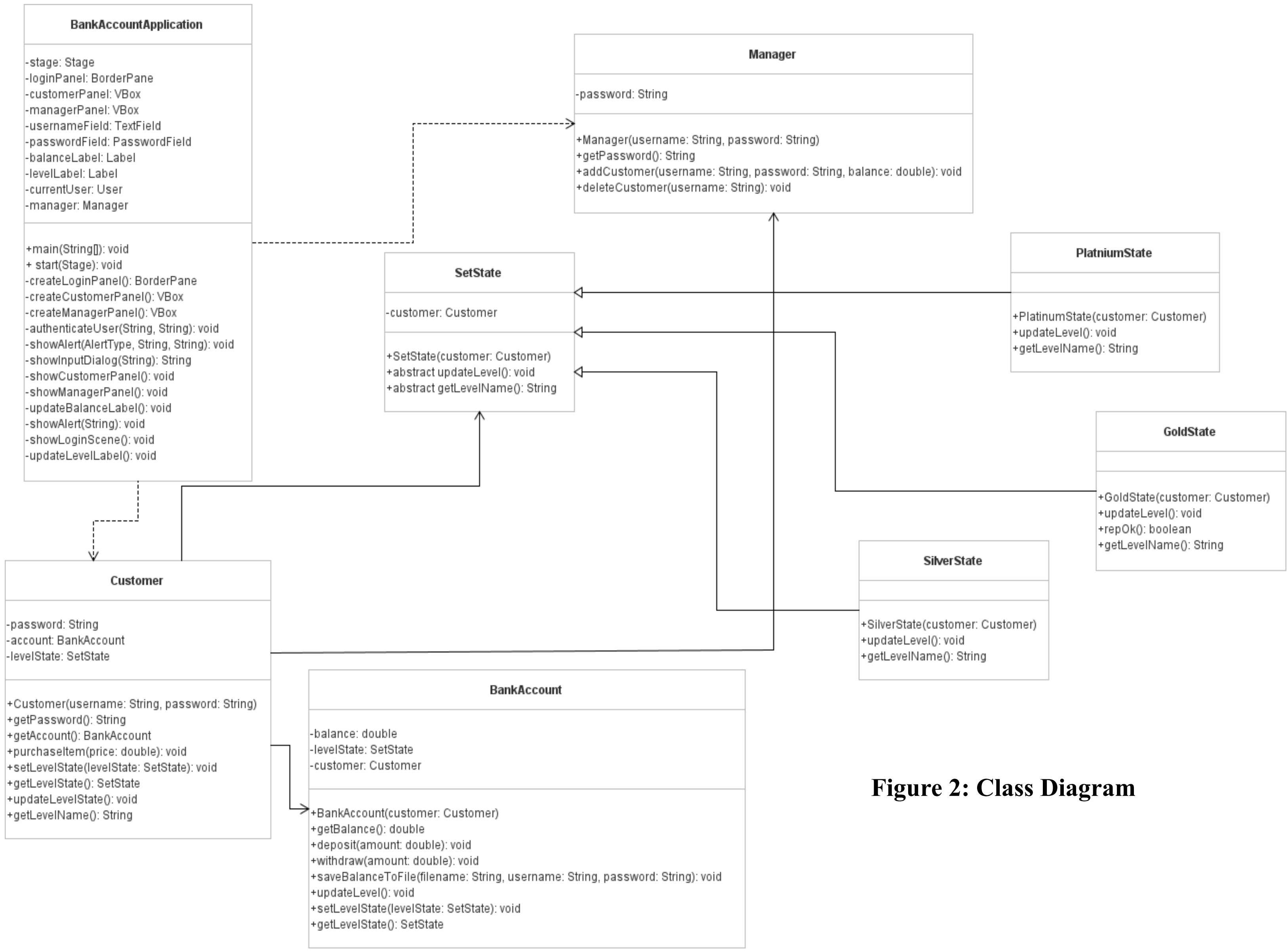
BankAccountApplication
-stage: Stage
-loginPanel: BorderPane
-customerPanel: VBox
-managerPanel: VBox
-usernameField: TextField
-passwordField: PasswordField
-balanceLabel: Label
-levelLabel: Label
-currentUser: User
-manager: Manager
+main(String[]): void
+ start(Stage): void
-createLoginPanel(): BorderPane
-createCustomerPanel(): VBox
-createManagerPanel(): VBox
-authenticateUser(String, String): void
-showAlert(AlertType, String, String): void
-showInputDialog(String): String
-showCustomerPanel(): void
-showManagerPanel(): void
-updateBalanceLabel(): void
-showAlert(String): void
-showLoginScene(): void
-updateLevelLabel(): void
Manager
-password: String
+Manager(username: String, password: String)
+getPassword(): String
+addCustomer(username: String, password: String, balance: double): void
+deleteCustomer(username: String): void
SetState
-customer: Customer
+SetState(customer: Customer)
+abstract updateLevel(): void
+abstract getLevelName(): String
PlatniumState
+PlatinumState(customer: Customer)
+updateLevel(): void
+getLevelName(): String
GoldState
+GoldState(customer: Customer)
+updateLevel(): void
+repOk(): boolean
+getLevelName(): String
SilverState
+SilverState(customer: Customer)
+updateLevel(): void
+getLevelName(): String
Customer
-password: String
-account: BankAccount
-levelState: SetState
+Customer(username: String, password: String)
+getPassword(): String
+getAccount(): BankAccount
+purchaseItem(price: double): void
+setLevelState(levelState: SetState): void
+getLevelState(): SetState
+updateLevelState(): void
+getLevelName(): String
BankAccount
-balance: double
-levelState: SetState
-customer: Customer
+BankAccount(customer: Customer)
+getBalance(): double
+deposit(amount: double): void
+withdraw(amount: double): void
+saveBalanceToFile(filename: String, username: String, password: String): void
+updateLevel(): void
+setLevelState(levelState: SetState): void
+getLevelState(): SetState

Figure 2: Class Diagram

The class I chose to create the overview, clauses and abstract/repOK functions for was the GoldState class. The GoldState represents the "GOLD" level state of a customer in the bank system. The class updates the level of the associated customer based on their account balance. If the account balance is less than $10,000, set the level to silver, if the account balance is $10,000 or more but less than $20,000, keep the level as gold if it's more than $20,000 it will set the level to platinum. Referring back to Figure 2 of my UML class diagram, the parts that form my state design pattern we learned in the lectures are "SetState" , "SilverState", "GoldState" and "PlatinumState". SetState is the parent class while the rest are the children classes of SetState.