COE528: Project Report

Name: Mohamed Khedr

ID: 501218833

section number: 02

Use Case Diagram Description:

The Use Case Diagram represents the interactions and actions that can be done within a banking system by two primary actors, a Customer and a Manager. The Customer can perform several actions: log in, log out, deposit money, withdraw money, get his balance, and do online purchase. On the other hand, the Manager can log in and log out in addition to a different set of responsibilities, including adding customers and deleting customers. The adding Customer use case includes the create account use case, which indicates that performing AddCustomer use case would include (require) the CreateAccount use case. This setup indicates that while Customers can manage their accounts, view their balance, and perform transactions, Managers have the authority to add or remove customers and oversee the account creation process. The diagram shows the distinct roles and interactions between customers and managers in this banking application.

AddCustomer Use Case Description:

Use case name	AddCustomer
Participating actors	Manager
Flow of events	1. The logged in manager initiates adding customer by clicking on the Add Customer button. 2. Two text fields will become visible to add the username and the password of the new user. 3. The manager collects the username and password from the client and enters them into the text-filled boxes. 4. The CreateAccount use case gets initiated creating a new Account instance with an initial balance of $100. 5.The data of the new Customer gets saved and a new file with the name of the customer's username gets created in the Data_Base directory, containing all of the new user's data.
Entry condition	Manager clicks on the Add Customer Button.
Exit condition	Manager confirms by clicking on the Add Customer Button.
Quality requirements	If a file already exists with the entered username, the new user will not be created. The use case CreateAccount is initiated during the 4th step in the flow of events.

Class Diagram Description:

The class diagram represents a simple banking system with various classes and interfaces that interact and operate together to implement the different responsibilities, scenarios, and states that make up this system. The diagram outlines the structure and relationships between these classes, highlighting inheritance, aggregations, dependencies, composition, and key operations.

First, there is the ‘BankAccount’ class with methods to get and set the balance and ensure account validity. It holds the account balance data and manages setting it whenever an operation is performed that may affect this balance, such as deposits and withdrawals.

Next, the state design pattern is managed and implemented by the abstract class ‘Level’ and its subclasses ‘Silver’, ‘Gold’, and ‘Platinum’. These three subclasses represent the different concrete states and implement different online fees for online purchases. The ‘Level’ class represents the state which is a part of the Customer - context - class (aggregation) and implements the different states (polymorphic objects) according to the current balance of the customer.

The ‘User’ interface declares and specifies the basic, common user operations (Login and Logout) that both the ‘Customer’ and the ‘Manager’ are expected to perform. Both the ‘Customer’ and the ‘Manager’ classes implement the ‘User’ interface and provide concrete implementations for the Login and Logout methods.

The ‘Manager’ class represents the manager actor and is responsible for creating, adding, and deleting customers. The ‘Manager’ class has a dependency relationship with the ‘Customer’ class, as shown in the diagram, since it uses the ‘Customer’ class in some of its methods.

Finally, the ‘Customer’ class represents the customer actor and is arguably the most important class that all the others serve in one way or another. The ‘Customer’ class has a ‘BankAccount’ and a ‘Level’, which are represented by the aggregation relationships shown in the class diagram. The methods in this class use some of the other classes to implement all the use cases expected to be performed by the customer. It includes methods for managing account operations such as deposits, withdrawals, balance inquiries, level updates, and online purchases.

Point 2 - Detailed Specification: ‘Customer’ Class

For the detailed specification tasks outlined in point number 2, I have chosen to focus on the **Customer** class. Nevertheless, I have diligently applied these specifications to all major classes within the project, including Manager, Customer, BankAccount, Level, Silver, Gold, and Platinum. Specifically, for each of these classes, I have:

- Written the Overview clause stating the responsibility of the class and its mutability as Javadoc comments.
- Defined the abstraction function and the representation invariant within the Javadoc comments.
- Provided the necessary clauses (such as effects, modifies, and requires) for each method as Javadoc comments.

- Implemented the abstraction function within the toString() method.
- Implemented the representation invariant within the repOk() method.

State Design Pattern:

In the UML class diagram for the bank account application, the State design pattern is represented through the relationship between the ‘Customer’ class (the context) and the ‘Level’ abstract class (the state) along with its concrete states’ implementations (Silver, Gold, Platinum). Below is a detailed description of how the components of the State design pattern are represented in the UML class diagram:

1. Context Class: Customer

   Responsibilities:

   - Maintains a reference to an instance of a Level object, which holds the current state of the customer.
   - Delegates state-specific behavior to the current Level object.
   - Manages the balance of the customer's account and updates the Level based on the balance.

   Key Methods:

   - updateLevel(): Determines and sets the appropriate Level object based on the customer's account balance.
   - doOnlinePurchase(double amount): Executes an online purchase by delegating to the current Level object's discount(double amount) method.

2. Abstract State Class: Level

   Responsibilities:

   - Defines the interface for state-specific behavior.
   - Provides default behavior that can be shared among concrete states.
   - Represents the state’s data type in the ‘Customer’ class that can be conveniently casted to the different concrete state implementations.

   Key Method:

   - abstract double discount(double amount): Calculates the total cost of an online purchase by adding the fee based on the customer's level.

3. Concrete States: Silver, Gold, Platinum

   Responsibilities:

   - Implement the exact fee value and the state-specific behavior defined by the Level abstract class.