Use Case Diagram

The use case diagram uses standard UML notation, like ovals, to represent use cases or methods and stick figures to represent the actors. The lines between the actors and the use cases represent the interactions. The diagram also includes automatic calls of validation methods, represented by a dashed line with an open arrowhead and "<<include>>" beside it. Lastly, the dashed line with an open arrowhead and "<<extend>>" beside it represents the extension of the use case attached to the <<extend>> end and the use case attached to the other end of the line under certain circumstances.

Use case name	*Purchase*
Participating actors	Initiated by *Customer*
Flow of events	1. *Customer* interacts with "Login as Customer" by inputting credentials. 2. Bank App responds by presenting the Customer GUI after checking credentials. *3.* *Customer* interacts with "Online Purchase" by inputting the amount. 4. Bank App responds with a display after checking funds in balance.
Entry condition	● *Customer* info exists in "Customer" folder ● Purchase amount of *Customer* over $50 and below or equal to balance.
Exit condition	● *Customer* either sees "PURCHASE COMPLETED!" or "ERROR, PLEASE TRY AGAIN!" based on the balance and purchase condition.
Special requirements	None

Class Diagram

The class diagram for the banking application represents the relationships and interactions between different classes in the application. It includes classes such as Customer, Manager and User. The User class is the base class with attributes like username, password, role, balance and methods for login and logout. The Manager and Customer classes inherit from the abstract class User. The Manager class has methods like addCustomer and deleteCustomer. The Customer class has an Account object and methods like depositBalance, withdrawBalance, getBalance, and onlinePurchase. The Account class is associated with the Customer class and has a balance and level attribute and methods like getBalance, setBalance, etc. Also, the Account class' overview and necessary clauses and the abstraction and rep invariant function, implemented as toString() and repOk(), are provided as Javadoc comments. The levels (Silver, Gold and Platinum) inherit attributes and methods from an abstract class Level following the State Design Pattern. The diagram illustrates the flow of data and control between these classes, providing a visual representation of the software architecture of the banking application.