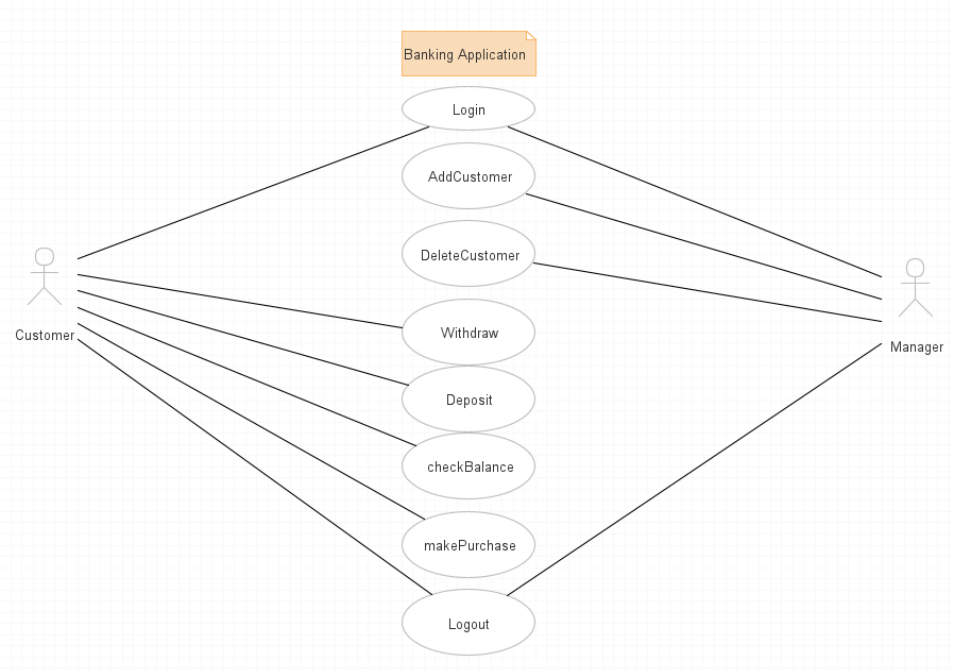
Use Case Diagram:

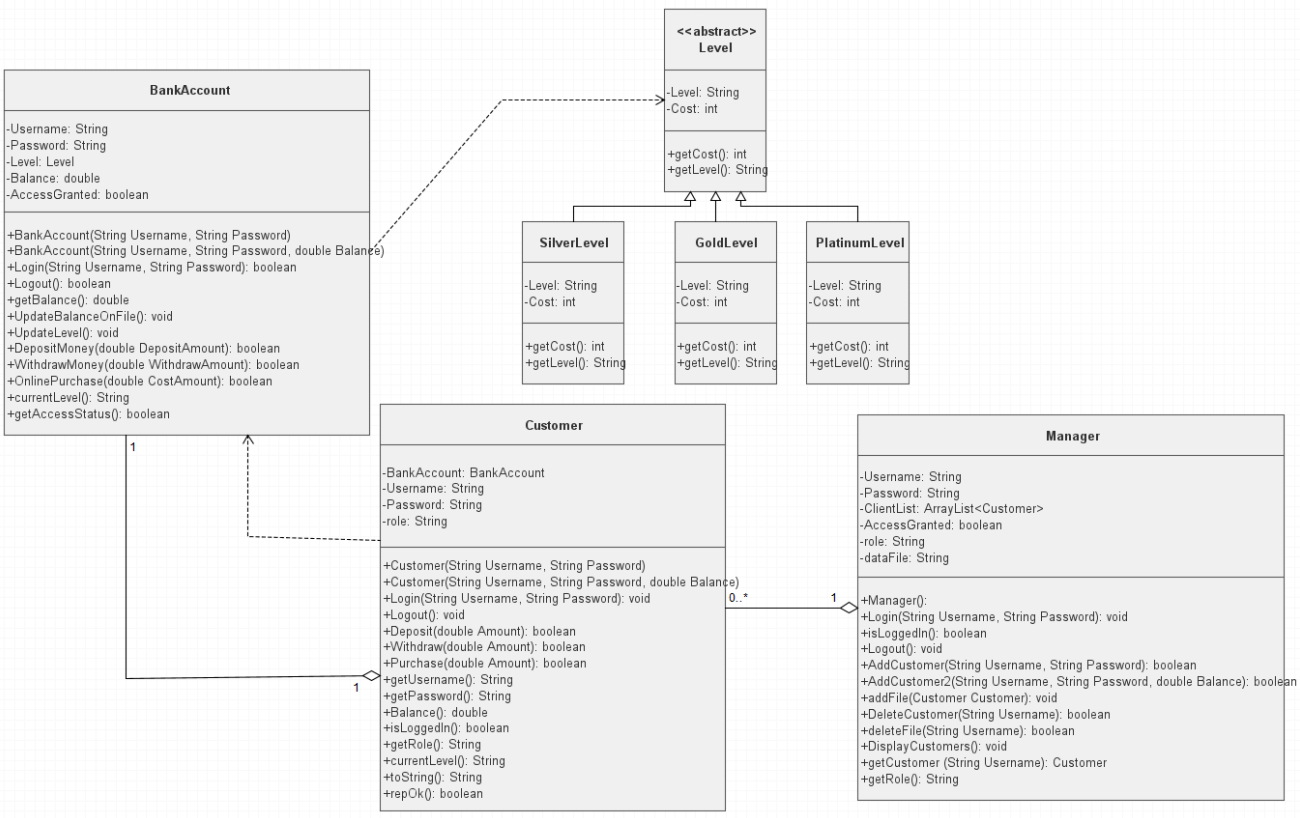
Use case name	BankingApplication
Participating Actors	Manager and Customer
Flow of Events	1. Manager enters username and password to login. 2. After authentication, the Manager can add/delete Customers, and then can logout. 3. Customer enters username and password to login. 4. After authentication Customer can check balance, Level, make a withdrawal, deposit or purchase, and then can logout.
Entry Condition	Manager or Customer wants to access the bank system.
Exit Condition	Manager or Customer wants to log out of bank system.

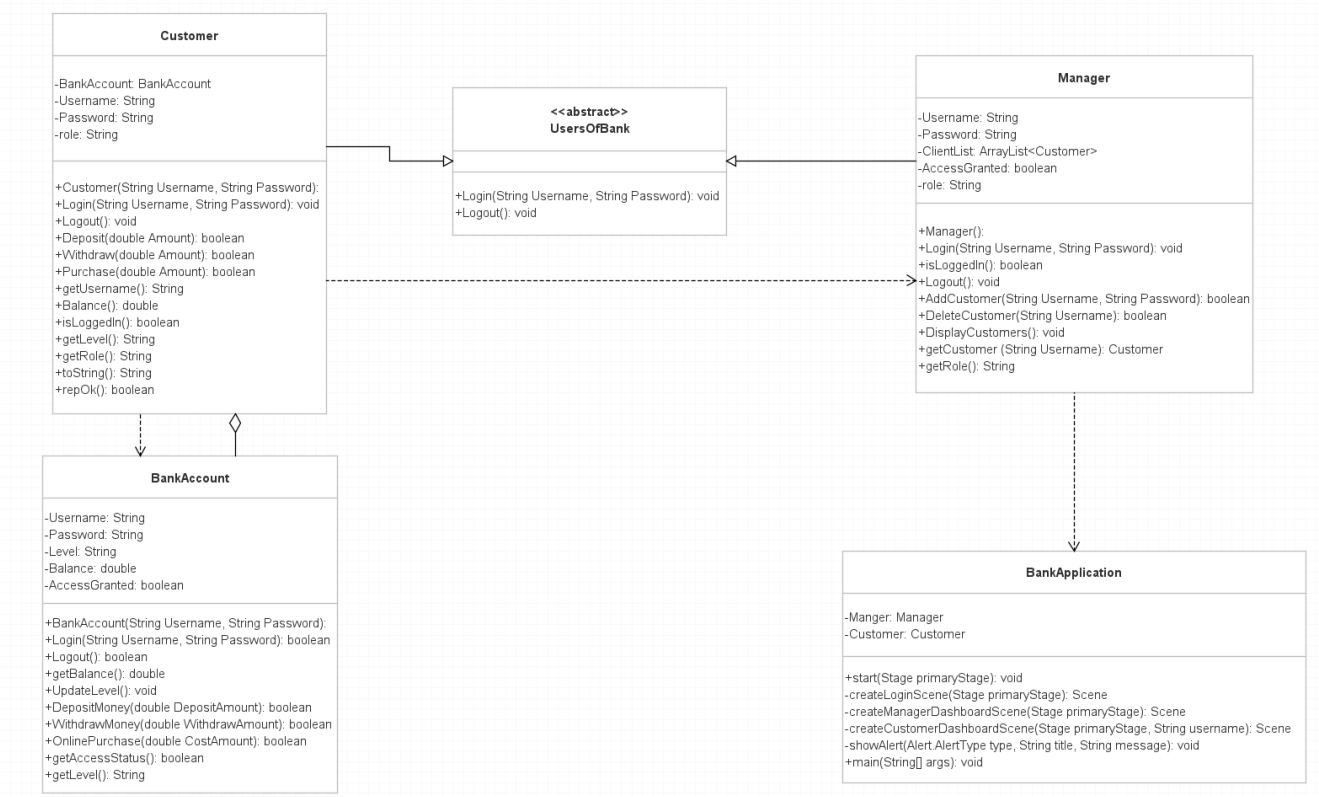
In my UML use case diagram there are 2 actors, a Manager and Customer. Both Manager and Customer can use the Login and Logout method in order to log in or log out of the bank system however there are some methods only the Customer can access and some only the Manager is able to access. The Customer has access to the Login, Withdraw, Deposit, Purchase and Balance methods which have the entry condition of a Customer wanting to log in, make a Withdrawal, Deposit, Purchase or check their Bank Account Balance. And the exit condition of wanting to log out would make a Customer use the Logout method. The Manager has access to the Login, AddCustomer, DeleteCustomer method which has the entry condition of the Manager wanting to log in, add or delete customers from the banking system. The exit condition of wanting to logout is when the Manager uses the Logout method.



Class Diagram:

In my UML class diagram there are 7 classes: BankAccount, Customer, Manager, Level, SilverLevel, GoldLevel and PlatinumLevel. BankAccount. The Level class is abstract and since the SilverLevel, GoldLevel and PlatinumLevel classes extend it there is an inheritance relationship between all these classes. Each BankAccount class has a Level which is initialized depending on the balance of the BankAccount, everytime the Balance is updated the Level of the BankAccount is also updated which is why the BankAccount class is dependent on the Level class because the extra cost during a purchase is dependent on the Level. Each Customer class has exactly 1 BankAccount class which is why there is an aggregation relationship between both classes, the Customer class is also dependent on the BankAccount class because if any changes were made inside of the BankAccount class the methods in the Customer class would change as well. For each Manager class there are exactly zero or more Customer's so there is an aggregation relationship between these classes. In the Customer class of my Banking Application project I wrote down the Overview clause, I also wrote the abstraction function and Representation invariant for this class. In order to check the abstraction function and representation invariant I also implemented the methods toString() and repOk() in the Customer class then for each method in the Customer class I also provided the necessary EFFECTS, MODIFIES and REQUIRED clauses which describe each method.





State design Pattern:

In my UML class diagram the Classes which implemented the State design Pattern are the Level class, which is extended by the SilverLevel, GoldLevel and PlatinumLevel class. Inside of the BankAccount class there is a Level object, depending on the balance of the BankAccount class the state of the Level will change to either SilverLevel, GoldLevel or PlatinumLevel. Each Level will have a different behavior to the BankAccount which is the extra cost during a purchase. So the part that forms the State Design pattern in my class diagram is the BankAccount, Level, SilverLevel, GoldLevel and PlatinumLevel class.