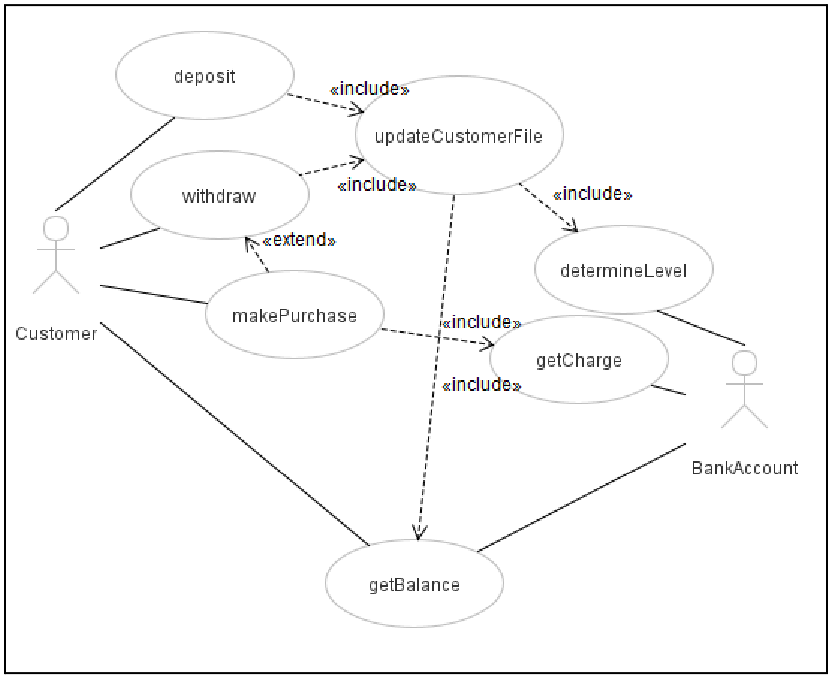
Report for BankApp program made by: Dip Tandel (#501171728) - Section 13

General Use Case Diagram (from the perspective of the Customer):



Participating actors:

- Initiated by Customer
- Communicates with BankAccount

Flow of Events:

1. The Customer chooses to **deposit** money into their BankAccount,
   at which point the updateCustomerFile function is activated to change the information in their file (which itself refers to determineLevel and getBalance from the BankAccount)
2. The Customer chooses to **withdraw** money out of their BankAccount,
   at which point the updateCustomerFile function is activated to change the information in their file (which itself refers to determineLevel and getBalance from the BankAccount)
3. The Customer chooses to make an **online purchase**, which is simply a withdrawal with an additional fee; provided by BankAccount's getCharge function.
4. The Customer chooses to simply **see their balance,** which is provided by BankAccount's getBalance function.

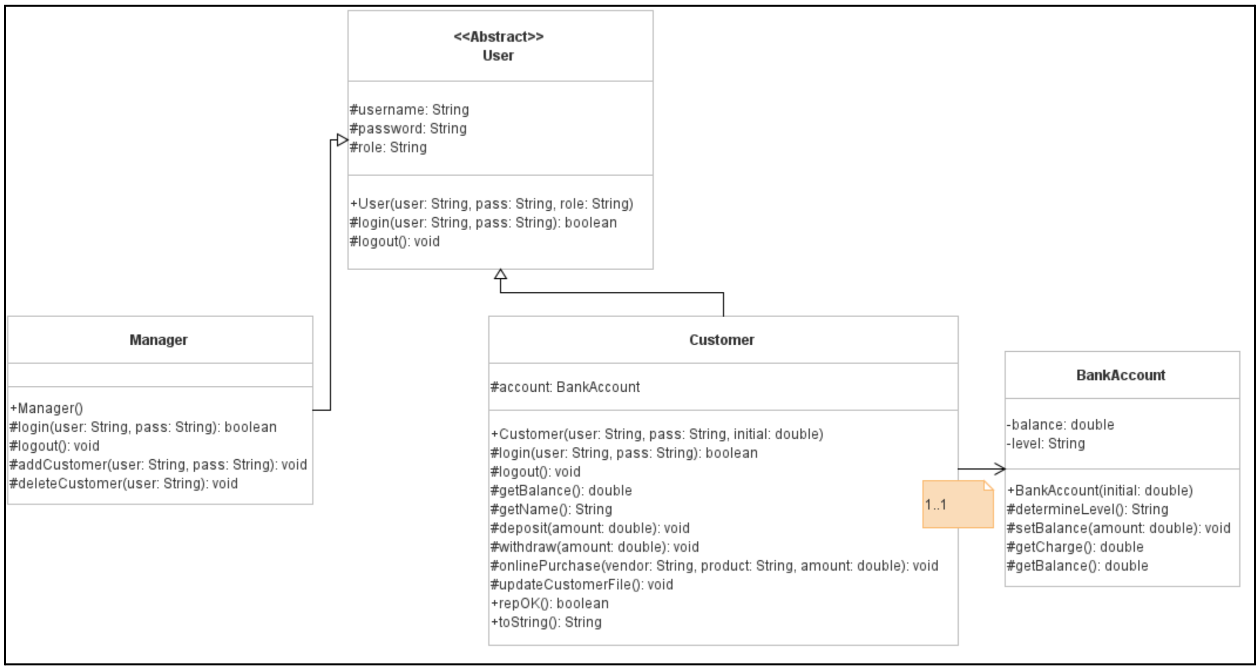
Entry Condition:

- The Customer is logged into their account and is presented with 4 options.

Exit Condition (any of the following):

- The Customer successfully makes a deposit or receives an error message.
- The Customer successfully makes a withdrawal or receives an error message.
- The Customer successfully makes an online purchase or receives an error message.
- The Customer views their current balance.

Class Diagram (main file not included - simply contains GUI scenes):


<<Abstract>>
User
#username: String
#password: String
#role: String
+User(user: String, pass: String, role: String)
#login(user: String, pass: String): boolean
#logout(): void
Manager
+Manager()
#login(user: String, pass: String): boolean
#logout(): void
#addCustomer(user: String, pass: String): void
#deleteCustomer(user: String): void
Customer
#account: BankAccount
+Customer(user: String, pass: String, initial: double)
#login(user: String, pass: String): boolean
#logout(): void
#getBalance(): double
#getName(): String
#deposit(amount: double): void
#withdraw(amount: double): void
#onlinePurchase(vendor: String, product: String, amount: double): void
#updateCustomerFile(): void
+repOK(): boolean
+toString(): String
1..1
BankAccount
-balance: double
-level: String
+BankAccount(initial: double)
#determineLevel(): String
#setBalance(amount: double): void
#getCharge(): double
#getBalance(): double

To see a higher quality version of this diagram, please view the violet.html files provided within the project folder of the submission.

Brief Description:

The Abstract class "User" describes the common attributes any user would have: a username, password, and a role. Each user would also need a method to log in and out of their BankApp. The two classes, "Manager" and "Customer" extend the abstract class since each is a type of "User", **inheriting** User's common attributes and methods, as well as implementing any unique attributes or methods of their own. For instance, the Customer requires an additional attribute called an account (type: BankAccount), thus, the class "Customer" is **associated** with the class "BankAccount". The note indicating (1..1) implies that there is exactly one instance of the object "BankAccount" associated with a "Customer" object. In order to maintain the security of the application, most attributes in BankAccount are private, most changes are instead performed by the Customer class instead (ie, deposit(), withdraw(), and onlinePurchase()). Moreover, the Manager class does not need many methods as a manager does not have an account, the Manager class simply needs to be able to add or delete customers. This is done by creating files which contain Customer information (Customer login() reads the username and password from these files, whereas the manager username and password are fixed as "admin"). The Customer class is also responsible for updating its .txt file; updateCustomerFile(), this method is called after each deposit, withdrawal, or online purchase.

Since the Customer class is the most critical for the functionality of this application, I selected the **Customer class to include an Overview clause**, abstraction function, rep invariant, as well as the necessary "effects", "modifies" and "requires" clauses for all of its methods.

Finally, the level of the Customer's account has to change dynamically during the application's runtime **(state design pattern)**, this is handled by the updateCustomerFile() method (inside Customer class), which itself refers to the determineLevel() method (inside BankAccount class). As mentioned earlier, the function is called every time there is a deposit, withdrawal, or online purchase, and it updates/overwrites the .txt file to keep the level and balance up to date. It is also referred to in the main function which contains the GUI, since one of the labels (when the Customer chooses to view balance) needs to be updated before the Customer views the label.