Vincent Yeung

500760793

Section 08

**Report: COE528 Final Project**

**Use Case Diagram:**

The Use Case diagram generated demonstrates the functionality of the Bank Account application. There are two actors associated with the application, being the manager and customers. The customers can log in, log out, check their balance, deposit money, withdraw money, and make online purchases. The manager can log in, log out, add and remove customers files. In order to perform removal of customer files, the manager must first login to their account to complete the task. Once the manager is logged in, they can add as many customers as they can and delete any customer files that exists. The customer can also log in, log out, assuming their files have been created by the manager. If login is successful, they have access to depositing and withdrawing money, making online purchases, checking the balance, and log out.

**Use case description: Deleting Customer**

Participating actors: Communicates with Manager

Flow of events:

1. The manager must log in to their account, by entering their username and password, and hitting login in the log in screen.
2. The specific customer’s file must exist in the directory where customer files are stored.
3. Once it’s verified that it exists, the manager will be able to delete the customer’s files.
4. Lastly, the admin can choose to add more customers, delete customers, or log out.

Entry Event: The manager logs in and enters data of an existing customer file.

Exit Event: The manager clicks the Delete Customer button in their account screen.

**Class Diagram:**

The class diagram is used to explore the relationships among the classes. The relationships include inheritance, aggregation, and depends. In the Class diagram generated, the inheritance relationship can be seen with the abstract class *Level* class, and the child classes **Silver**, **Gold** and **Platinum**. The inheritance relationship can also be seen with the abstract class *Person*, with child classes **Manager** and **Customer**. The aggregation relationship can be seen with the **Account** class and the abstract class *Level*. This relationship exists because every account has a level, depending on the balance associated with the account. Aggregation can also be seen **Account** class and the abstract class *Person*, as each account has either a customer or a manager. Aggregation can also be seen with the **BankApp** class and the **Account** class. This relationship exists because BankApp has accounts. This differs from aggregation relationship because although the BankApp has many accounts, if the BankApp disappears, the information will remain stored in a database elsewhere, and so the information does not disappear. The last relationship, depends relationship, was not used in this project. Abstract classes in the class diagram are shown as <<ClassName>>, public visibility is denoted by “+”, and private by “-”.

**Addressing Point two:**

The class I have chosen to address point two, was the Person class. The person class has overview clause stated at the top, describing the class, as well as the abstract function and representation invariant in form of Javadoc comments. The abstract function is implemented as a toString method, and the representation invariant is implemented as a method with return type Boolean, called repOK().

**State Pattern Design:**

The state pattern is seen with the abstract class called *Level*, its child classes **Silver**, **Gold**, and **Platinum**, and the Context class which is **Account**. The change of level of the context is delegated to the concrete classes, which are the child classes of *Level*. If there are internal changes of the object, the Account class relies on the concrete classes to perform the change of level for each account. Each concrete class overrides an abstract method in *Level,* and has a toString method which returns the appropriate level of the account.