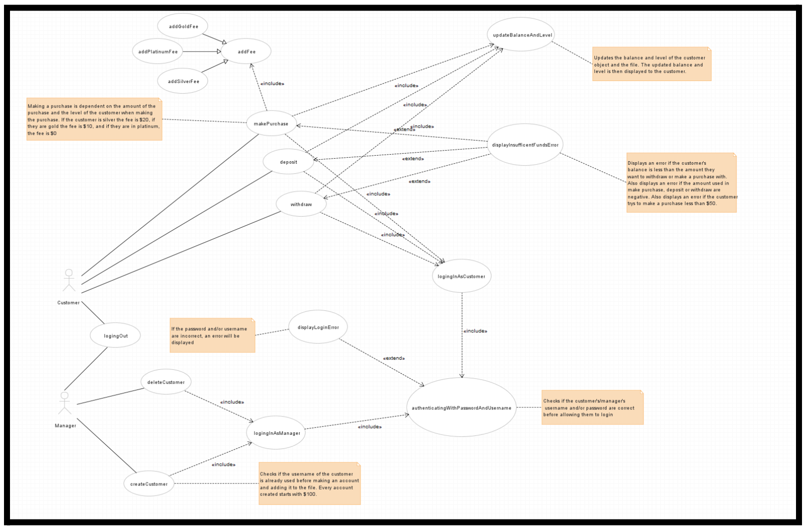
COE528 Final Project Report

By Davidy Kwok

Introduction:

This project aims to create a banking application for two user types: Customers and Managers. The application was developed using JavaFX, NetBeans 8.2, and JavaFX Scene Builder. This report will discuss the classes used to build the application, illustrated by a class diagram and a use case diagram.

Use Case Diagram:



Making a purchase is dependent on the amount of the purchase and the level of the customer when making the purchase. If the customer is silver the fee is $20, if they are gold the fee is $10, and if they are in platinum, the fee is $0

If the password and/or username are incorrect, an error will be displayed

Checks if the username of the customer is already used before making an account and adding it to the file. Every account created starts with $100.

Updates the balance and level of the customer object and the file. The updated balance and level is then displayed to the customer.

Displays an error if the customer's balance is less than the amount they want to withdraw or make a purchase with. Also displays an error if the amount used in make purchase, deposit or withdraw are negative. Also displays an error if the customer trys to make a purchase less than $50.

Checks if the customer's/manager's username and/or password are correct before allowing them to login

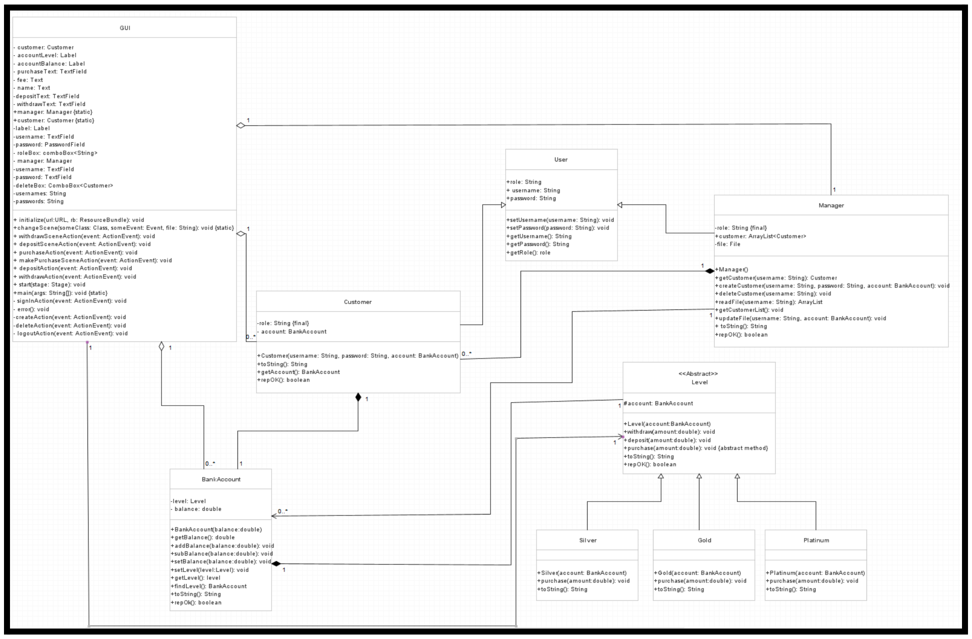
A use case diagram provides an abstract view of the behavior of a system. This diagram models the different functionalities of the banking system from the customers' and manager's perspective. The customer can make a purchase, deposit and withdraw money ,view their level and balance as well as logout. For the customer to access all these features the customer must be logged in. Once the customer enters a username and password, these fields are authenticated and if either the username and/or password are incorrect, an error message will be displayed. If the customer chooses to make a purchase, if the amount is insufficient or a invalid amount is entered, an error message will appear. If the amount is feasible, depending on the level of the customer, a fee is applied and the balance and level are updated. If the customer chooses to deposit or withdraw similar to make a purchase, if the amount is insufficient or invalid, an error will be displayed. If the values entered are valid, the system will update the level and balance of the customer. The manager's actions consist of logging out, logging in , creating a customer and removing a customer. The manager must log in before creating and deleting accounts. The username and password are authenticated before the manager can log in. If the username and/or password are incorrect, an error message will display. If the username already exists and the manager tries to create a new account with that username, an error message is displayed.

COE528 Final Project Report

Use Case Description:

Use Case Name	makePurchase
Participating Actors	Customer
Flow of Events	1. The Customer enters their username and password to login. 2. The System verifies the information and either displays an error if incorrect or allows the Customer access to their account. 3. The Customer enters the amount they want to use to make a purchase. If the amount is greater than their balance or a invalid amount, then an error is displayed. 4. A fee is added to the amount depending on the level of the customer.(Silver $20, Gold $10, and Platinum 0$). 5. The system updates the balance and level of the Customer accordingly.
Entry Condition	Customer logs in
Exit Condition	- Customer logs out. - Customer switches to deposit or withdraw.

Class Diagram:


GUI
- customer: Customer
- accountLevel: Label
- accountBalance: Label
- purchaseText: TextField
- fee: Text
- name: Text
-depositText: TextField
- withdrawText: TextField
+manager: Manager {static}
+customer: Customer {static}
-label: Label
-username: TextField
-password: PasswordField
- roleBox: comboBox<String>
- manager: Manager
-username: TextField
-password: TextField
-deleteBox: ComboBox<Customer>
-usernames: String
-passwords: String
+ initialize(url:URL, rb: ResourceBundle): void
+changeScene(someClass: Class, someEvent: Event, file: String): void {static}
+ withdrawSceneAction(event: ActionEvent): void
+ depositSceneAction(event: ActionEvent): void
+ purchaseAction(event: ActionEvent): void
+ makePurchaseSceneAction(event: ActionEvent): void
+ depositAction(event: ActionEvent): void
+ withdrawAction(event: ActionEvent): void
+ start(stage: Stage): void
+main(args: String[]): void {static}
- signInAction(event: ActionEvent): void
- error(): void
-createAction(event: ActionEvent): void
-deleteAction(event: ActionEvent): void
- logoutAction(event: ActionEvent): void
User
+role: String
+ username: String
+password: String
+setUsername(username: String): void
+setPassword(password: String): void
+getUsername(): String
+getPassword(): String
+getRole(): role
Manager
-role: String {final}
+customer: ArrayList<Customer>
-file: File
+Manager()
+getCustomer(username: String): Customer
+createCustomer(username: String, password: String, account: BankAccount): void
+deleteCustomer(username: String): void
+readFile(username: String): ArrayList
+getCustomerList(): void
+updateFile(username: String, account: BankAccount): void
+ toString(): String
+repOK(): boolean
Customer
-role: String {final}
- account: BankAccount
+Customer(username: String, password: String, account: BankAccount)
+toString(): String
+getAccount(): BankAccount
+repOK(): boolean
<<Abstract>>
Level
#account: BankAccount
+Level(account:BankAccount)
+withdraw(amount:double): void
+deposit(amount:double): void
+purchase(amount:double): void {abstract method}
+toString(): String
+repOK(): boolean
BankAccount
-level: Level
- balance: double
+BankAccount(balance:double)
+getBalance(): double
+addBalance(balance:double): void
+subBalance(balance:double): void
+setBalance(balance:double): void
+setLevel(level:Level): void
+getLevel(): level
+findLevel(): BankAccount
+toString(): String
+repOk(): boolean
Silver
+Silver(account: BankAccount)
+purchase(amount:double): void
+toString(): String
Gold
+Gold(account: BankAccount)
+purchase(amount:double): void
+toString(): String
Platinum
+Platinum(account: BankAccount)
+purchase(amount:double): void
+toString(): String
1
0..*

COE528 Final Project Report

This class diagram consists of 9 classes, with the GUI class comprising 7 controller.java classes. In the diagram, the Level class has an "inherits" relationship with the Silver, Gold, and Platinum classes, as they all extend the abstract class. The Level class has a composition relationship with the BankAccount class, as without a BankAccount, there isn't a Level. The Level class also has an association relationship with the GUI as the GUI uses the methods in the Level class. The BankAccount class has 3 more relationships with other classes. The first relationship is a composite relationship with the Customer class, as without the Customer class, the BankAccount class would not exist. The second relationship is an associated relationship with the Manager class, as the Manager class uses the BankAccount object. The last relationship is an aggregation relationship with the GUI, since a GUI has many BankAccounts. The Customer class has 3 more relationships. The first is an "inherits" relationship with the User class, as Customer extends User. The second is a composition relationship with the Manager class, as the Customer can't exist without the Manager first creating a Customer. The last relationship is an aggregation relationship with the GUI, as the GUI has a Customer object. The final relationship is an aggregation relationship between the Manager and the GUI, as the GUI has a Manager object.

Javadoc Comments:

The selected classes for writing Javadoc comments are the Manager class, Customer class, BankAccount class, and Level class. These classes were chosen because each plays a crucial role in the program's operation. The Manager class enables users to create and delete customers; without this function, no customers can enter the system. The Customer class sets the password and username for each customer; without this class, there would be no customers able to use the system. The BankAccount class manages the user's balance; without it, the balance would not update. The Level class is used to set the customer's level, which determines the fee for the purchase function. Additionally, the Level class includes the withdraw, deposit, and purchase functions, so without it, customers would not be able to perform any actions in the system.

State Design:

The State Design Pattern allows an object to change its behavior when its internal state changes. This pattern is implemented with the Level, Silver, Platinum, and Gold classes. The Level class is abstract and contains withdraw and purchase methods, as well as an abstract purchase method. Whenever a customer withdraws or deposits money, the Level class will change the customer's level state if the balance changes enough. For the purchase method, each level has a different implementation. Silver incurs a $20 fee, Gold incurs a $10 fee, and Platinum has no fee. Since the purchase method's behavior changes based on the internal state, it follows the State Design Pattern.