Rose Pagano - 501165663 Section 9

COE 528 Final Project Report

Part I: Case Diagram

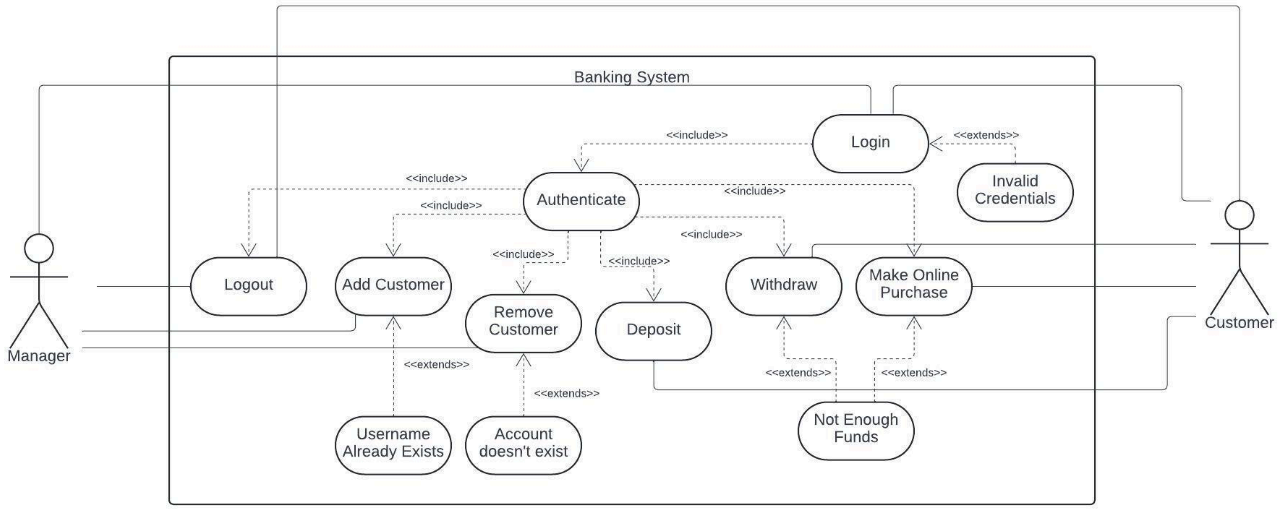


Figure 1: Case Diagram
Prompt: Describe your Use Case Diagram in a paragraph.

The Case Diagram (Figure 1), was constructed using 12 cases and 2 actors - a singular Manager and several Customers. The Manager has access to the cases Login, Logout, Add Customer, and Remove Customer. While the Customers have access to the deposit, withdraw and make online purchase cases. This is shown with the solid lines. All scenarios start from the login function where from there the information is either authenticated or prompted back to the login screen when one of actors information is invalid. After the automatic authentication this is where depending on the where the actor is a Customer or Manager (based on their credentials), they have access to the different functionalities of the application. Where the action can be completed successfully or rerouted back if information is not valid. These cases use the <<extends>> clause since they are seldom invoked use cases. The <<includes>> clause is used to represent the next action that could occur if the previous action runs successfully. In sum, this case diagram is a visual representation of all the cases that the application would encounter and how they are managed.

Prompt: Describe one of the use cases following the template from Lecture.

Use case name	`AddCustomer`
Participating actors	Initiated by `Manager`
Flow of Events	1. `Manager` enters the username and password of the customer that they would like to add and clicks “Add Customer”

	2. The `Banking System` responds by checking if the entered information is valid. Valid information is: - the username doesn't already exist - the username/password doesn't contains any spaces - username/password field isn't empty 3. If the above conditions are the met, the `Customer` is added to the database with a starting balance of $100.00, and a success message is displayed.
Entry condition	`Manager` has logged in
Exit condition	The `Customer` has been added to the database with a success message. OR The `Banking System` prompts the `Manager` to enter valid information.
Special Requirements	None.

Part II: Class Diagram & State Design Pattern

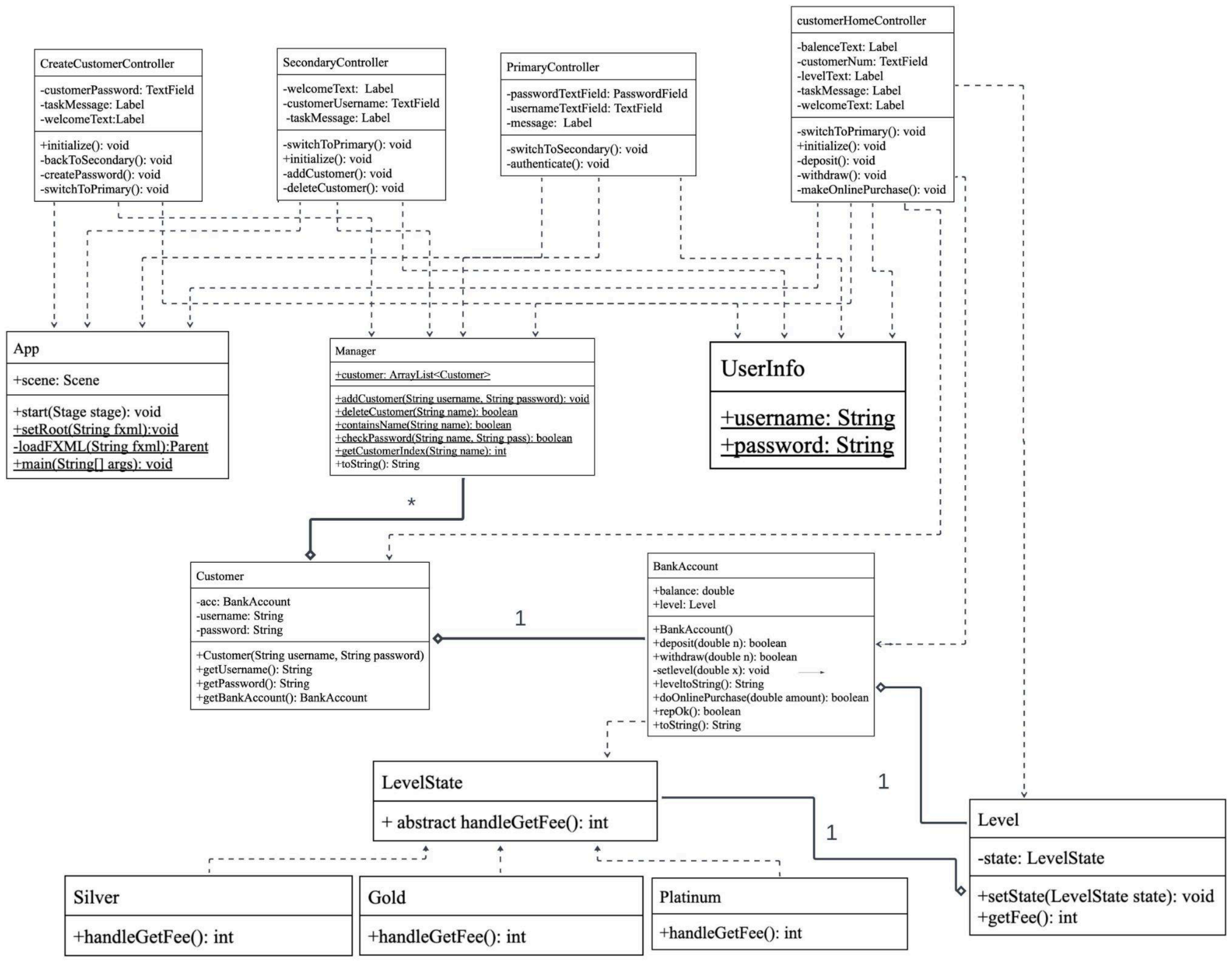
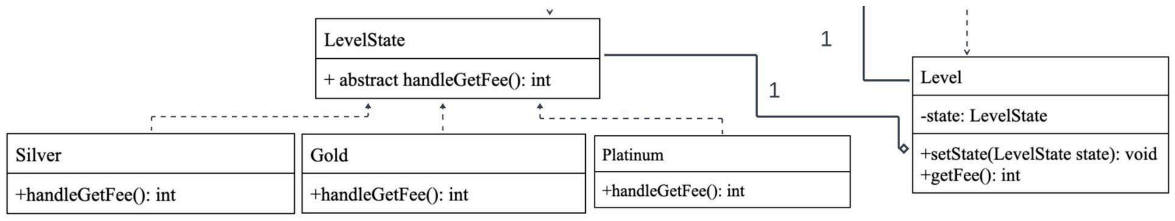


Figure 2: Class Diagram

Prompt: Describe your Class Diagram in a paragraph.

The Class diagram (Figure 2) illustrates behavior of the Banking Application. In this diagram, the dotted line with an open arrow indicates the dependency relationship between classes. The closed line with a diamond represents the aggregation relationship between classes with the number representing the multiplicity of the object's instances. The dotted line with a closed arrow represents the implementation of interfaces. Here we have the Controller classes, in a dependency relationship with the App, Manager, UserInfo, Customer, and BankAccount class. The manager class has an ArrayList of type Customer, therefore they have an aggregation relationship. As well as all Customers have an instance variable BankAccount, and LevelState in Level. The Level, LevelState, Silver, Gold and Platinum classes will be discussed in further below as it regards the State Design Approach. To conclude, the class diagram displays all classes, methods and instance variables and how they interact with each other.

Prompt: Refer to your UML class diagram and indicate the part(s) that form the State design pattern.



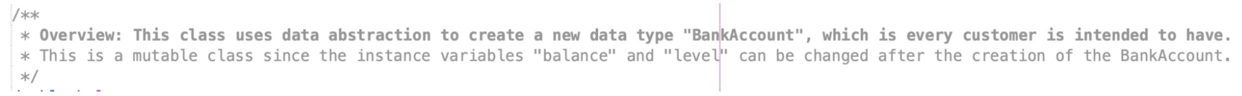
This section of the class diagram displays the State Design Pattern, which is used to help with the functionality of customers making an online purchase. Here, the BankAccount Object has the ability to have different fees depending on the state it is in. Here we have the "LevelState" class being the context that delegates a state-specific request to handleGetFee() to Level's getFee(). Silver, Gold and Platinum are the different states which return a different respective fee. To the getFee() method. This allows for a streamlined approach when adjusting the BankAccount balance after making an online purchase.

Part III: Class of Choice - BankAccount

The BankAccount Class is the chosen class for the Overview, effects, modifies, requires clause, abstraction function, and rep invariant. Below is the outline of each section of this class.

<u>Part A</u>: The Overview Clause

Prompt: Write the Overview clause stating the responsibility of the class and whether the class is mutable or not. Provide this as javadoc comments.



```
/**
 * Overview: This class uses data abstraction to create a new data type "BankAccount", which is every customer is intended to have.
 * This is a mutable class since the instance variables "balance" and "level" can be changed after the creation of the BankAccount.
 */
```

Figure 3: Overview clause for BankAccount

<u>Part B</u>: The Abstraction Function & Rep Invariant

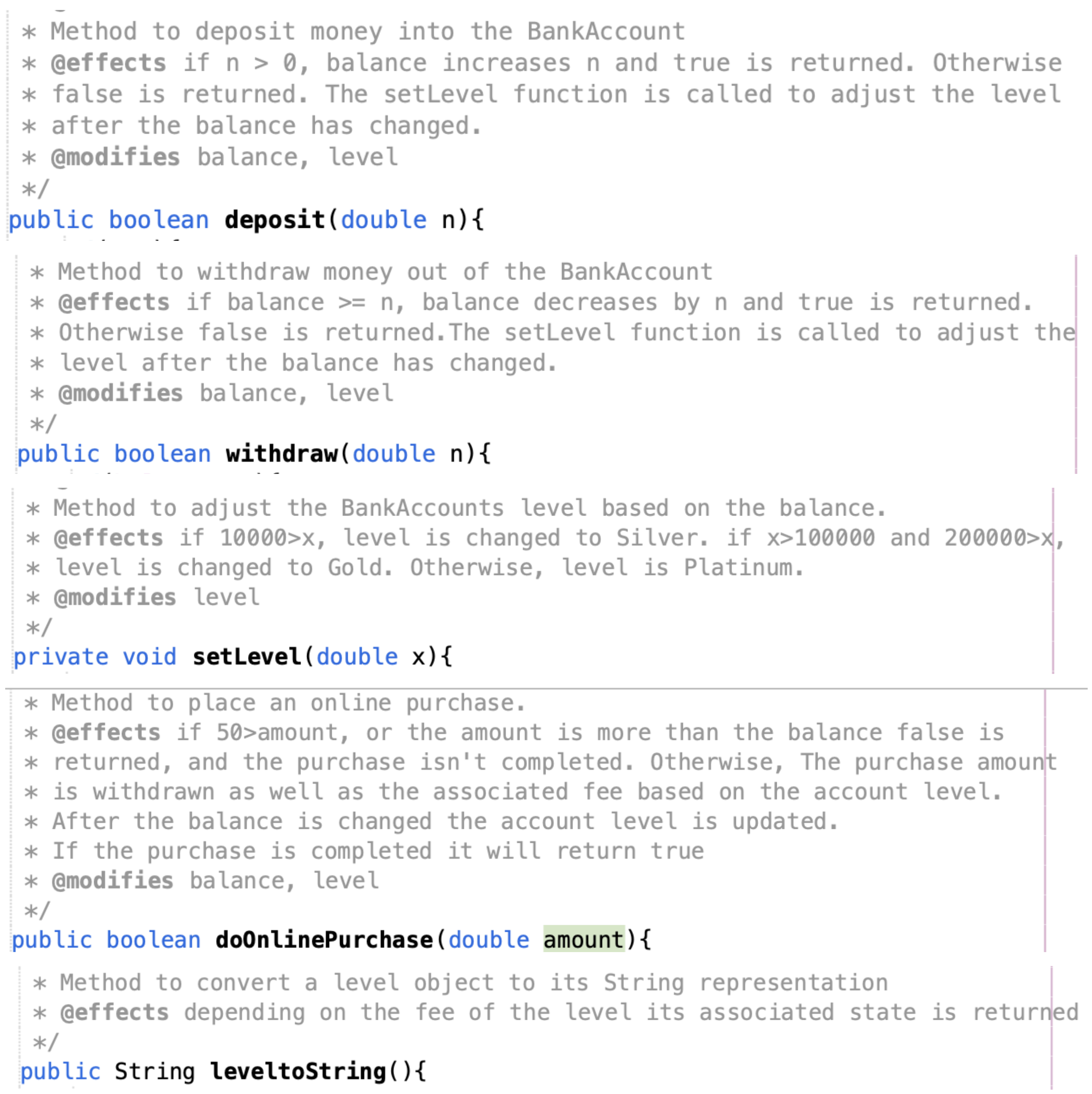
Prompt: Write the abstraction function and the rep invariant as javadoc comments.

```
* Abstraction function: A typical Bank Account has a balance and a level
* AF(balance, level) = A Bank Account with a balance and level.
* Rep Invariant:
* balance > 0
```

Figure 4: Abstraction Function & Rep Invariant for BankAccount

<u>Part C</u>: Necessary Clauses

Prompt: Provide the necessary clauses (e.g. effects, modifies and requires) for each method as javadoc comments for the method



```
 * Method to deposit money into the BankAccount
 * @effects if n > 0, balance increases n and true is returned. Otherwise
 * false is returned. The setLevel function is called to adjust the level
 * after the balance has changed.
 * @modifies balance, level
 */
public boolean deposit(double n){
```

```
 * Method to withdraw money out of the BankAccount
 * @effects if balance >= n, balance decreases by n and true is returned.
 * Otherwise false is returned.The setLevel function is called to adjust the
 * level after the balance has changed.
 * @modifies balance, level
 */
public boolean withdraw(double n){
```

```
 * Method to adjust the BankAccounts level based on the balance.
 * @effects if 10000>x, level is changed to Silver. if x>100000 and 200000>x,
 * level is changed to Gold. Otherwise, level is Platinum.
 * @modifies level
 */
private void setLevel(double x){
```

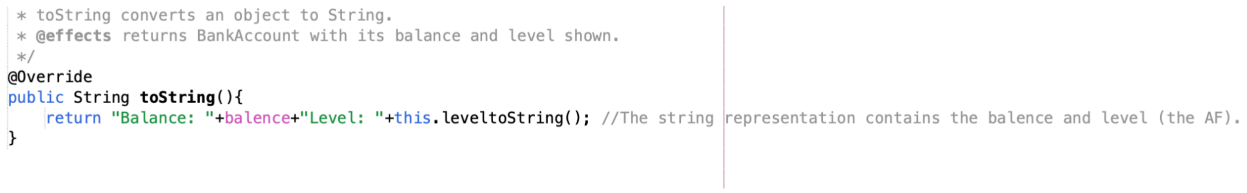
```
 * Method to place an online purchase.
 * @effects if 50>amount, or the amount is more than the balance false is
 * returned, and the purchase isn't completed. Otherwise, The purchase amount
 * is withdrawn as well as the associated fee based on the account level.
 * After the balance is changed the account level is updated.
 * If the purchase is completed it will return true
 * @modifies balance, level
 */
public boolean doOnlinePurchase(double amount){
```

```
 * Method to convert a level object to its String representation
 * @effects depending on the fee of the level its associated state is returned
 */
public String leveltoString(){
```

Figure 5: Requires, Modifies, Effects clauses for each method in BankAccount

<u>Part D</u>: toString Method with Abstraction function

Prompt: Implement the abstraction function in the toString() method

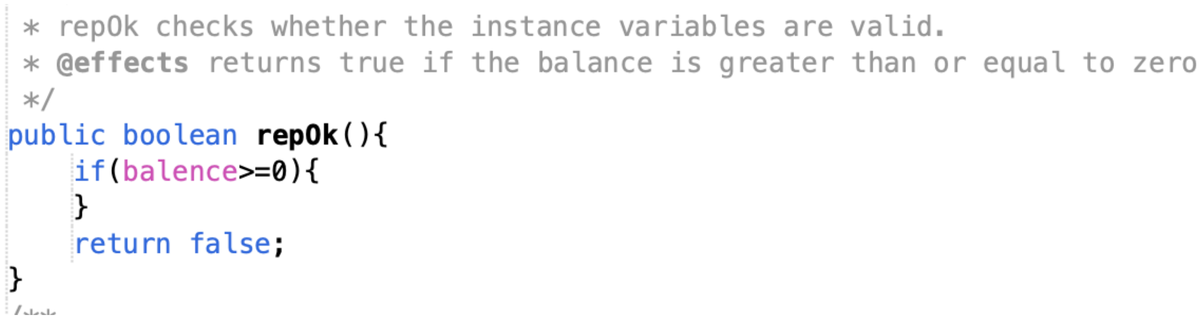


```
 * toString converts an object to String.
 * @effects returns BankAccount with its balance and level shown.
 */
@Override
public String toString(){
    return "Balance: "+balence+"Level: "+this.leveltoString(); //The string representation contains the balence and level (the AF).
}
```

Figure 6: toString Method with Abstraction function in BankAccount

<u>Part E</u>: repOk Method with Rep invariant

Prompt: Implement the rep invariant in the repOk() method



```
 * repOk checks whether the instance variables are valid.
 * @effects returns true if the balance is greater than or equal to zero
 */
public boolean repOk(){
    if(balence>=0){
    }
    return false;
}
```

Figure 7: repOk Method with Rep invariant in BankAccount

Refer to BankAccount.java for full source code.

References

Amarasinghe, S., Chlipala, A., Devadas, S., Ernst, M., Goldman, M., Guttag, J., Jackson, D., Miller, R., Rinard, M., & Solar-Lezama, A. (2021). *You are not logged in. Reading 11: Abstraction Functions & Rep Invariants*. MIT. Retrieved March 25, 2024, from https://web.mit.edu/6.031/www/sp21/classes/11-abstraction-functions-rep-invariants/

Booch, G., & Jain, S. (2021, September 6). *Use Case Diagram for Online Banking System*. GeeksforGeeks. Retrieved March 25, 2024, from https://www.geeksforgeeks.org/use-case-diagram-for-online-banking-system/

GeeksforGeeks. (2024, February 13). *Important Topics for the State Design Pattern*. GeeksforGeeks. Retrieved March 25, 2024, from https://www.geeksforgeeks.org/state-design-pattern/

JavaFX Tutorial. (n.d.). Tutorialspoint. Retrieved March 25, 2024, from https://www.tutorialspoint.com/javafx/index.htm

Scene (JavaFX 8). (n.d.). Oracle Help Center. Retrieved March 25, 2024, from https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Scene.html

UML Class Diagrams. (n.d.). cs.wisc.edu. Retrieved March 25, 2024, from https://pages.cs.wisc.edu/~hasti/cs302/examples/UMLdiagram.html