XI'AN JIAOTONG LIVERPOOL UNIVERSITY

YEAR 4 SEMESTER 2
ASSIGNMENT SUBMISSION

Name	Sahand	Sabour
ID Number	*1614650*	
Major	*Computer Science and Technology*	
Module Title	*Software Engineering II*	
Module Code	*CSE306*	
Assignment Title	*Assignment 1*	
Submission Deadline	*5th April, 2020 23:59PM*	
Lecturer Responsible	*Tin Soon Phei*	

Signature: Sahand Sabour Date: 2020/04/05

For Academic Office use:	Date Received	Days Late	Penalty

Part A: Identifying Patterns

1. This Java code snippet displays a composition of objects in a tree structure that represents part-whole hierarchies. To enunciate, in the given example, the Manager class acts as the composite class and it stores subordinates, which are Employee objects. Each instance of the Manager object would then have a list of Employee objects, which overall would construct a tree structure of connections between the objects. Hence, it is believed that this section implements the **Composite Pattern**, which is a structural pattern.

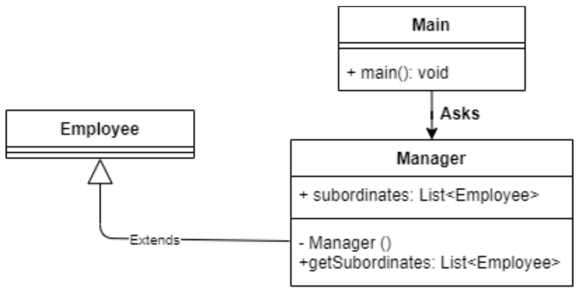
Main
+ main(): void
Asks
Employee
Manager
+ subordinates: List<Employee>
Extends
- Manager ()
+getSubordinates: List<Employee>

Figure A-1: part1-Java Code UML Diagram

2. In the given code snippet, AnotherObject class represents the functionality of SomeObjectImpl class and provides its functionalities to the users. Hence, it can be implied that the AnotherObject class is used a substitute/placeholder object for the SomeObjectImpl class and the client. With this implication, AnotherObject acts as a proxy class and therefore, **Proxy Pattern**, which is a structural pattern, is believed to be implemented in this section.

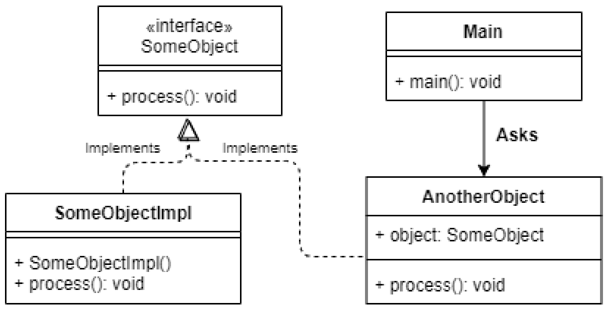

Figure A-2: part2-Java Code UML Diagram

3. In the given implementation, the ApplicationRelay class handles all the communications between different User objects and consequently, reduces communication complexity

between these objects by acting as a mediator. Therefore, it is believed that the **Mediator Pattern** is used, which is a behavioral pattern.

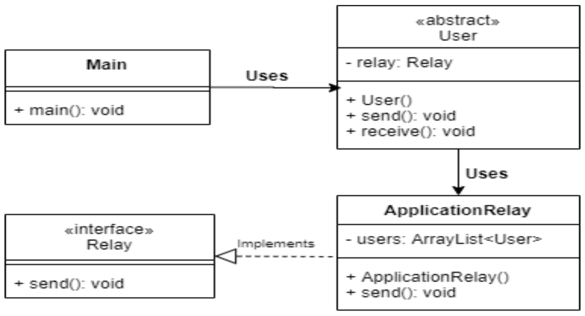

Figure A-3: part3-Java Code UML Diagram

Part B: Correcting Problems

1) The Level enum includes three constants (states): bronze, silver, and gold. For each instance of the FitnessCustomer class, it is beneficial to specify a modifiable state in order to represent change in the customer's level since the object's behavior would change with its change of state. Hence, it is suggested that the **State Pattern**, which is a behavioral pattern, should be implemented.
2) The modified Java code would be as follows:

```java
public class FitnessCustomer {
    private Level level;

    public void setLevel(Level level) {
        this.level = level;
    }
    public double getFees() {
        return level.getFees();
    }
    public boolean canAccessPool() {
        return level.canAccessPool();
    }
    public boolean hasOwnLocker() {
        return level.hasOwnLocker();
    }
    public double getEquipmentDiscount() {
        return level.getEquipmentDiscount();
   }
}
```

FitnessCustomer.Java

```java
public interface Level {
    public double getFees();

    public boolean canAccessPool();

    public boolean hasOwnLocker();

    public double getEquipmentDiscount();

    public FitnessCustomer getCustomer();
}
```

Level.Java

```java
public class GoldLevel implements Level {

    private final double FEE = 2000;
    private final double DISCOUNT = 200;
    private FitnessCustomer customer;

    public GoldLevel(Level level) {
        this(level.getCustomer());
    }
    public GoldLevel(FitnessCustomer customer) {
        this.customer = customer;
    }
    @Override
    public double getFees() {
        return FEE;
    }
    @Override
    public boolean canAccessPool() {
        return true;
    }
    @Override
    public boolean hasOwnLocker() {
        return true;
    }
    @Override
    public double getEquipmentDiscount() {
        return DISCOUNT;
    }
    @Override
    public FitnessCustomer getCustomer() {
        return customer;
    }
}
```

GoldLevel.Java

```java
public class SilverLevel implements Level {

    private final double FEE = 1500;
    private final double DISCOUNT = 100;
    private FitnessCustomer customer;

    public SilverLevel(Level level) {
        this(level.getCustomer());
    }
    public SilverLevel(FitnessCustomer customer) {
        this.customer = customer;
    }
    @Override
    public double getFees() {
        return FEE;
    }
    @Override
    public boolean canAccessPool() {
        return true;
    }
    @Override
    public boolean hasOwnLocker() {
        return false;
    }
    @Override
    public double getEquipmentDiscount() {
        return DISCOUNT;
    }
    @Override
    public FitnessCustomer getCustomer() {
        return customer;
    }
}
```

SilverLevel.Java

```java
public class BronzeLevel implements Level {

    private final double FEE = 1000;
    private final double DISCOUNT = 50;
    private FitnessCustomer customer;

    public BronzeLevel(Level level) {
        this(level.getCustomer());
    }
    public BronzeLevel(FitnessCustomer customer) {
        this.customer = customer;
    }
    @Override
    public double getFees() {
        return FEE;
    }
    @Override
    public boolean canAccessPool() {
        return false;
    }
    @Override
    public boolean hasOwnLocker() {
        return false;
    }
    @Override
    public double getEquipmentDiscount() {
        return DISCOUNT;
    }
    @Override
    public FitnessCustomer getCustomer() {
        return customer;
    }
}
```

BronzeLevel.Java

3) The following Java Code was used to test the State Pattern Implementation.

```
public class Assignment1_Part2 {
    public static void main(String[] args) {
        FitnessCustomer Sahand = new FitnessCustomer();
        Sahand.setLevel(new GoldLevel(Sahand));

        System.out.println("Fee:  "+ Sahand.getFees()+"\tDiscount:  "+
        Sahand.getEquipmentDiscount());
        System.out.println((Sahand.canAccessPool()?"Has":"No")
       +" Pool Access");
        System.out.println((Sahand.hasOwnLocker()?"Has":"No")
       +" Own Locker");
        System.out.println();

        Sahand.setLevel(new SilverLevel(Sahand));

        System.out.println("Fee:  "+ Sahand.getFees()+"\tDiscount:  "+
       Sahand.getEquipmentDiscount());
        System.out.println((Sahand.canAccessPool()?"Has":"No")
       +" Pool Access");
        System.out.println((Sahand.hasOwnLocker()?"Has":"No")
       +" Own Locker");
        System.out.println();

        Sahand.setLevel(new BronzeLevel(Sahand));

        System.out.println("Fee:  "+ Sahand.getFees()+"\tDiscount:  "+
       Sahand.getEquipmentDiscount());
        System.out.println((Sahand.canAccessPool()?"Has":"No")+
       " Pool Access");
        System.out.println((Sahand.hasOwnLocker()?"Has":"No")+
       " Own Locker");
        System.out.println();
    }
}
```

Assignment1_PartB.Java

The obtained test results were recorded as follows:

```
run:
Fee: 2000.0             Discount: 200.0
Has Pool Access
Has Own Locker

Fee: 1500.0             Discount: 100.0
Has Pool Access
No Own Locker

Fee: 1000.0             Discount: 50.0
No Pool Access
No Own Locker

BUILD SUCCESSFUL (total time: 0 seconds)
```

Assignment1_PartB test results

Part C: Combining Design Patterns

1) The possible elegant overlay for Deposite pattern is display in the Figure C-1.

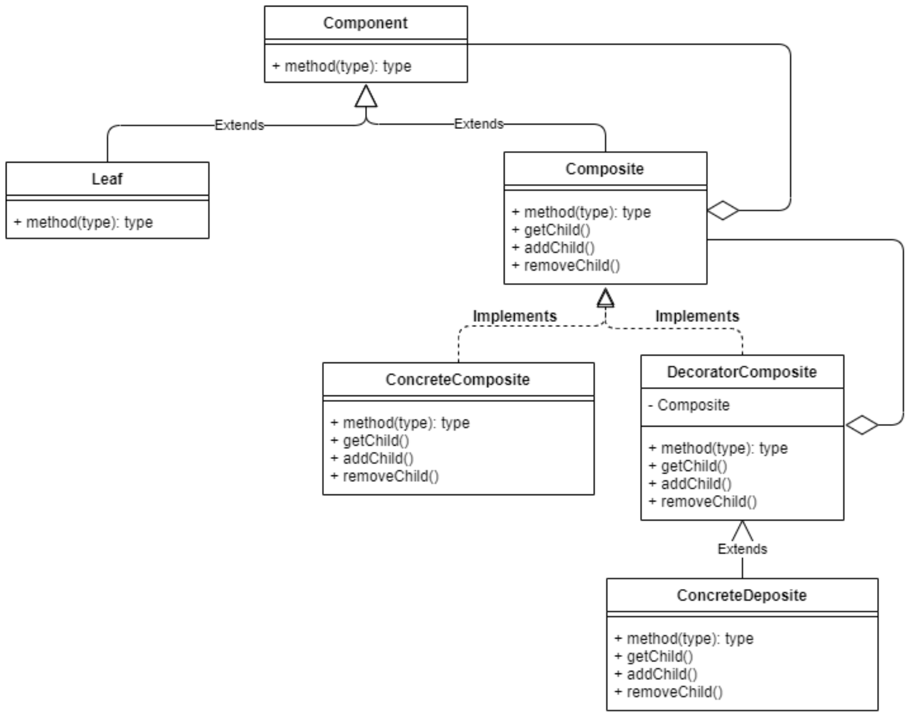

Figure C-1: Deposite Pattern UML

Based on the above figure, for a component in the program, a composite class would be created. This is beneficial when a group of objects are required to be treated in the similar way as the component class. Consequently, the composite class would be used as the base object for the decorator pattern. In this way, additional functionalities can be attached to the composite instances without modifying the composite object, which would result in a concrete deposite (decorator and composite) object.

2) The below figure (Figure C-2) illustrated the Prodapter pattern. As displayed in the figure, when a client attempts to use an interface, the proxy pattern would provide the functionality of this interface without giving the client direct access. This is highly beneficial when reducing complexity. Accordingly, the created proxy class would be connected to an adapter. This would enable the proxy object to be compatible with the client's requirements.

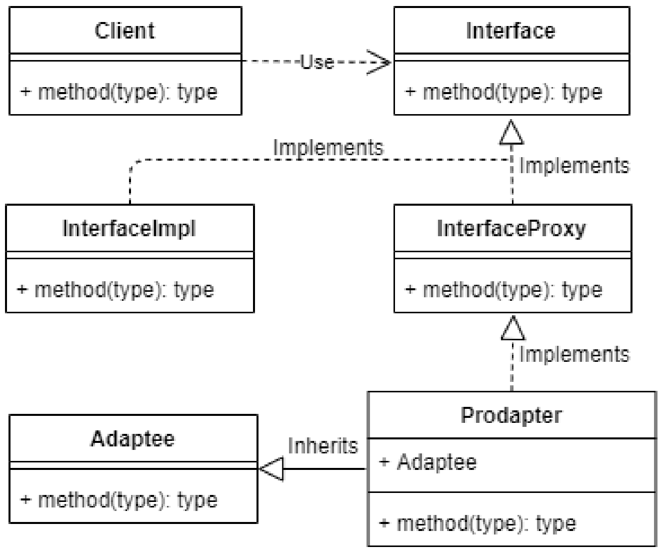

Figure C-2: Prodapter Pattern UML

3) Figure C-3 displays the UML for the Stategy pattern. Implementation of the State interface would cause the Context to change its behavior based on its State. Hence, based on the project requirements, a number of concrete states can be implemented via this interface. Subsequently, for each concrete state, a strategy is created. This enables concrete state classes to change its behavior or algorithm at runtime based on the client's request.

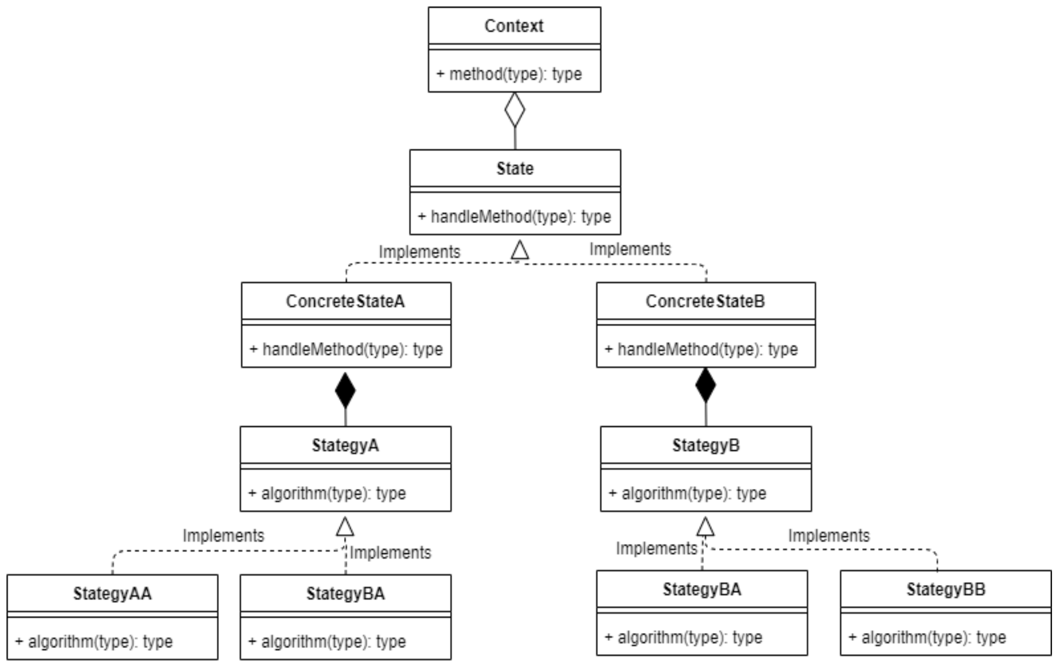

Figure C-3: Stategy Pattern UML