BIM492

DESIGN PATTERNS

HOMEWORK 1

Şevval Türkyılmaz	**49927389416**	**sevvaltrkylmzz**
Cansu Gürel	**16157401836**	**CansuG**

1. STATEMENT OF WORK: Write your problem definition in detail using 100-200 words.

- THE PROBLEM DEFINITION:

The problem we are trying to solve in this project is about an ice cream shop that has so many orders that they can't quite catch up. The ice cream shop is very well known around the city and has many flavors for the customers to choose from. Because of the busy schedule they have, they need a solution to keep track of all their orders and to save them some time in their business venture. Some customers want to pay with a credit card and due to the fact that the popularity of PayPal payment method these days, the shop owners want to give the customers variety of payment methods so all of them would be satisfied and the process of ordering would be lessened in the aspect of time.

2. DESIGN PATTERNS: Select at least two design patterns that are useful in solving the problem you defined. Explain why and how you employ such design patterns in your Project.

- DESIGN PATTERNS:

We chose Strategy and Decorator design patterns.

The reason we chose the strategy design pattern is that in order to provide different payment methods to customers, we need different strategies (behaviors). The strategy design pattern is one of the behavioral design patterns and is used when we have multiple algorithms for a specific task and the client decides the actual implementation to be used at runtime. In our problem, we have different payment behaviors to implement, so the strategy pattern is the most suitable for this issue.

The Decorator design pattern is one of the structural design patterns, which provides a dynamic way of extending an object's functionality. It's different than the traditional way of adding new functionality into an object using Inheritance, instead, it uses Composition which makes it flexible and allows the addition of new functionalities at the run time, as opposite to Inheritance, which adds new functionality at compile time. Ice cream is a classic example of a decorator design pattern. You create basic ice cream and then add toppings to it as you prefer. The added toppings change the taste of the basic ice cream. You can add as many toppings as you want.

- HOW WE IMPLEMENTED OUR DESIGN PATTERNS:

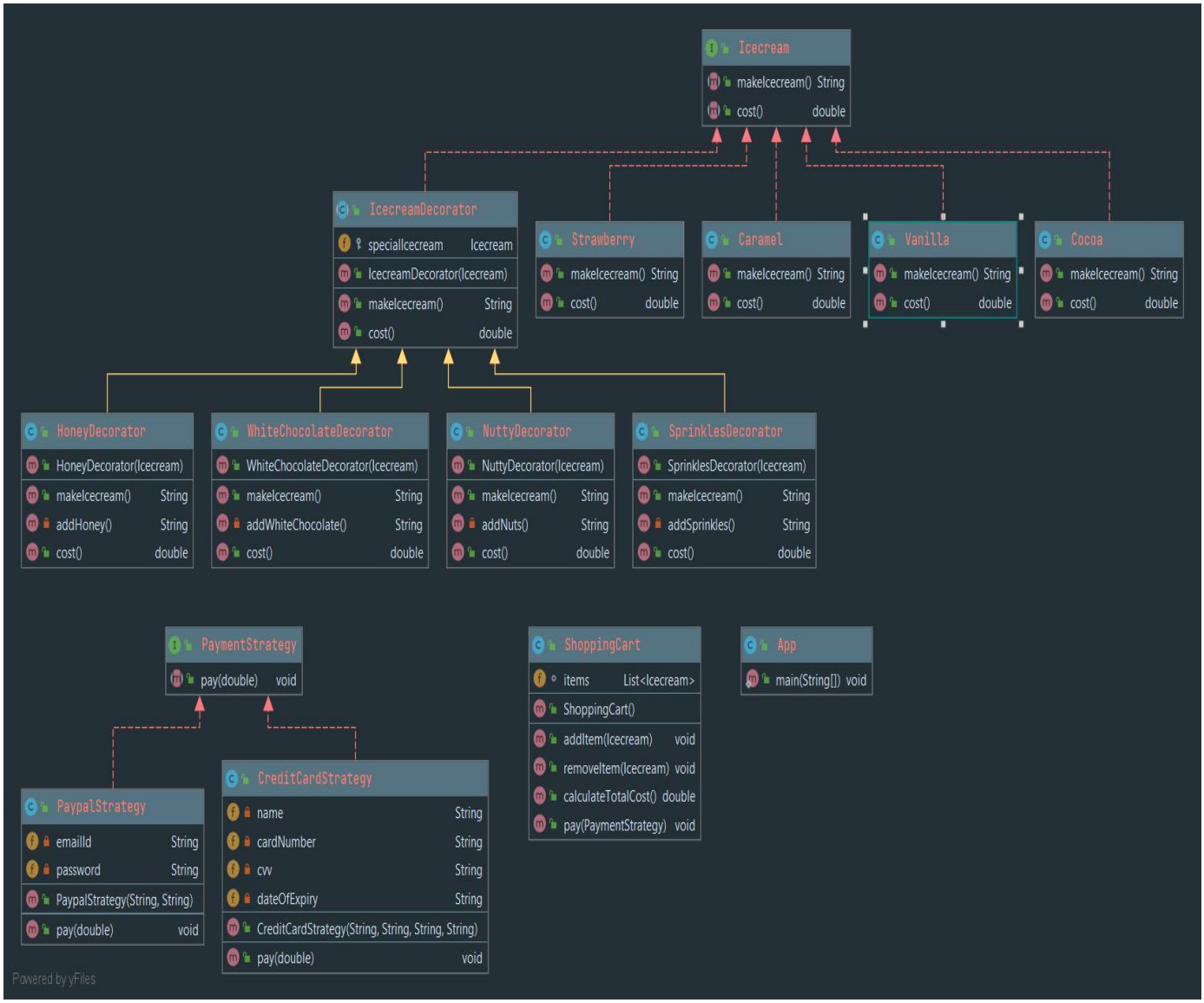
For our problem, we will try to implement a simple Shopping Cart for adding the ice cream orders.

First, we start with an interface named Icecream. It has concrete classes which are Vanilla, Strawberry, Cocoa, and Caramel. Then, we create an abstract class that contains (aggregation relationship) an attribute type of the interface. The constructor of this class assigns the interface type instance to that attribute. This class is the decorator base class. Then, we can extend this class and create as many concrete decorator classes. The concrete decorator class will add its own methods. After / before executing its own method, the concrete decorator will call the base instance's method. Key to this decorator design pattern is the binding of the method and the base instance happens at runtime based on the object passed as a parameter to the constructor.

And as for the payment part, where we have two payment strategies – using Credit Card or using PayPal.

First, we will create the interface for our project, in our case to pay the amount passed as an argument. Then after creating the interface of our base payment function, we move on to implementing two different strategies, in our case which are credit cards and PayPal. The PayPal and credit card classes have unique pay methods, which override the main interface'(Payment) method pay(). The PayPal strategy takes users' emails and passwords as parameters. The credit card strategy takes users' credit card information as parameters to provide payment. And in the end, the payment of the selected ice cream on the card proceeds with the desired payment method.

3. UML: Draw a detailed class diagram of your proposed solution using some UML Diagram Drawing Tool.



**********This UML Diagram is also attached as .jpg file.**********

4. RESEARCH:

We found a very well-known open source project which was using Observer design pattern. This kind of managing the subscribers are often used by projects which demands flow of information to its users, so they would always inform the users and in case of not wanting to proceed with the subscription anymore allowing them to easily unregister from the system.

- SOURCE CODE:

```
package observer;

import java.util.ArrayList;
import java.util.List;

interface Subject {
  void register(Observer obj);
  void unregister(Observer obj);
  void notifyObservers();
}

class DeliveryData implements Subject {

  private List<Observer> observers;
  private String location;

  public DeliveryData() {
    this.observers = new ArrayList<>();
  }

  @Override
  public void register(Observer obj) {
    observers.add(obj);
  }

  @Override
  public void unregister(Observer obj) {
      observers.remove(obj);
  }
```

```
  @Override
  public void notifyObservers() {
    for(Observer obj : observers) {
      obj.update(location);
    }
  }

  public void locationChanged() {
    this.location = getLocation();
    notifyObservers();
  }

  public String getLocation() {
    return "YPlace";
  }
}


interface Observer {
  public void update(String location);
}


class Seller implements Observer {
  private String location;

  @Override
  public void update(String location) {
    this.location = location;
    showLocation();
  }

  public void showLocation() {
    System.out.println("Notification at Seller - Current
Location: " + location);
  }
}

class User implements Observer {
  private String location;
```

```
  @Override
  public void update(String location) {
    this.location = location;
    showLocation();
  }

  public void showLocation() {
    System.out.println("Notification at User - Current Location:
" + location);
  }
}

class DeliveryWarehouseCenter implements Observer {
  private String location;

  @Override
  public void update(String location) {
    this.location = location;
    showLocation();
  }

  public void showLocation() {
    System.out.println("Notification at Warehouse - Current
Location: " + location);
  }
}

public class ObserverPatternTest {

  public static void main(String[] args) {
    DeliveryData topic = new DeliveryData();

    Observer obj1 = new Seller();
    Observer obj2 = new User();
    Observer obj3 = new DeliveryWarehouseCenter();

    topic.register(obj1);
    topic.register(obj2);
    topic.register(obj3);

    topic.locationChanged();
```

```
    topic.unregister(obj3);

    System.out.println();
    topic.locationChanged();

  }
}
```

- EXPLANATION:

Observer pattern is used when there is one-to-many relationship between objects, such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

In the Subject interface, the main 3 methods have been created to be used by the DeliveryData concrete class.

Other interface is Observer interface which has update method, and it has three observers as Seller, User and DeliveryWarehouseCenter.

DeliveryData class can change location; register, unregister, and notify the observers. These observers register so that they can be updated when the product has changed its location via phone or email. There is a system for finding location via GPS. Any time the product changed location, locationChanged() method in DeliveryData is called. This method gets the location and notify the observers. If any observer doesn't want to be updated anymore, it can be unregistered from the system.