Assignment 9

Title:

Identify and Implement GOF pattern

Problem Statement:

- Identification and Implementation of GOF pattern
- Apply any two GOF patterns to refine Design Model for a given problem description Using effective UML 2 diagrams and implement them with a suitable object oriented language.

Objective:

- To Study GOF Patterns.
- To identify applicability of GOF in the system
- Implement System with pattern.

Theory:

Strategy Design Pattern

In Software Engineering, the strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm's behavior to be selected at runtime. The GoF Design Patterns are broken into three categories: Creational Patterns for the creation of objects; Structural Patterns to provide a relationship between objects; and finally, Behavioral Patterns to help define how objects interact.

- **Creational Design Patterns**
  - **Abstract Factory** : Allows the creation of objects without specifying their concrete type.
  - **Builder** : Uses to create complex objects.
  - **Factory Method** : Creates objects without specifying the exact class to
  - create.
  - **Prototype** : Creates a new object from an existing object.
  - **Singleton** : Ensures only one instance of an object is created.

- **Structural Design Patterns**
  - **Adapter**. Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.

- **Bridge**. Decouples an abstraction so two classes can vary independently.
- **Composite** . Takes a group of objects into a single object.
- **Decorator** . Allows for an object's behaviour to be extended dynamically at run time.
- **Facade** . Provides a simple interface to a more complex underlying object.
- **Flyweight** . Reduces the cost of complex object models.
- **Proxy** . Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

- **Behavior Design Patterns**
  - **Chain of Responsibility**. Delegates command to a chain of processing objects.
  - **Command** . Creates objects which encapsulate actions and parameters.
  - **Interpreter** . Implements a specialized language.
  - **Iterator** . Accesses the elements of an object sequentially without exposing its underlying representation.
  - **Mediator** . Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
  - **Memento** . Provides the ability to restore an object to its previous state.
  - **Observer** . Is a publish/subscribe pattern which allows a number of observer objects to see an event.
  - **State** . Allows an object to alter its behaviour when its internal state changes.
  - **Strategy** . Allows one of a family of algorithms to be selected on-the-fly at run-time.
  - **Template Method**. Defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behaviour.
  - **Visitor** . Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Implementation

1. **Factory Design Pattern**

A factory pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. A factory method in the interface defers the object creation to one or more concrete subclasses at run time. The subclasses implement the factory method to select the class whose objects need to be created.

Example :

- In our project , we are using a factory design pattern for User class which is an abstract class.

- Then there are three subclasses Participant, Volunteer & Manager extending User class.
- Then created a factory to generate object of concrete class i.e. UserSelectFactory.
- Use a factory to get objects of concrete class by passing information as shown in Userfactory.

To apply the factory method pattern , let us first create the abstract User class and its subclasses.

```
public abstract class User
{
  private String name;
  private String gender;
  private String contactno;
  private String email;
  private String username;
  private char[] password;

  public void register() {
    // Code
  }

  public void login() {
    // Code
  }
}
```

Participant class

```
public class Participant extends User {

    @Override
    public void bookSlot() {
        selectEvent();
        selectTeam();
        selectPayment();
    }
}
```

Volunteer class

```java
public class Volunteer extends User {

    @Override
    public void bookSlot() {
        checkParticipantDetails();
        checkTeamDetails();
        updatePayment();
        updateWinner();
    }
}
```

Manager class

```java
public class Manager extends User {

    @Override
    public void bookSlot() {
        addVolunteer();
        checkParticipantDetails();
        checkTeamDetails();
        checkPayment();
        checkWinner();
    }
}
```

Next, we will create the UserSelectFactory class, which is the Creator in the application.

```
public class UserSelectFactory() {

    public User createUser(String type) {

        if(type == null) {
            return null;
        }
        else if(type =="Participant") {
            return new Participant();
        }
        else if(type == "Volunteer") {
            return new Volunteer();
        }
        else if(type == "Manager") {
            return new Manager();
        }
        else return null;
    }
}
```

Create a factory UserFactory to get objects of concrete class by passing information (type of user) as shown in Userfactory.

```
public class UserFactory() {

    public static void main(String[] args) {

        UserSelectFactory userselect = new UserSelectFactory();

        String type;

        System.out.println("Enter the type of user : ");

        Scanner sc=new Scanner(System.in);
        type=sc.next();

        User user = userselect.createUser(type);
    }
}
```

2. **Singleton Design Pattern**

The singleton pattern is one of the Gang of Four creational design patterns. In software engineering, this is a term that implies a class which can only be instantiated once, and a global point of access to that instance is provided. In the Java programming language, there are a few standard implementations of the singleton pattern.

Example :

- In our Project, the Administrator is a singleton class.
- Administrator class is instantiated only once.

```
public class Administrator
{
  private static Administrator instance = null;

  private String name;
  private String gender;
  private String contactno;
  private String email;
  private String username;
  private char[] password;

  private Administrator() {}
    public static Administrator getInstance() {
        if (instance == null) {
            instance = new FactoryManager();
        }
        return instance;
    }

    public void addManager() {
        // code
    }

    public void addVolunteer() {
        // code
    }

    public void addEvents() {
        // code
    }
```

```java
public void checkParticipantDetails() {
    // code
}

public void checkPayments() {
    // code
}

//Getter and Setter methods
public String getname() { return name; }
public String getgender() { return gender; }
public String getcontact() { return contactno; }
public String getemail() { return email; }

public void setname(String name) {
    this.name=name;
}

public void setcontact(String contactno) {
    this.contactno=contactno;
}

public void setemail(String email) {
    this.email=email;
}

public void setpassword(char[] pwd) {
    password=Arrays.copyOf(pwd,pwd.length);
}
}
```

3. **Facade Design Pattern**

   The Facade design pattern is one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

   Example :

   - In our project, to book a slot, the process is complex.
   - We require different services before slot booking : Availability Service, PaymentService, Booking Service.

- So, we have created SlotBookingFacade which is very simple for users to book a slot.

To apply the facade pattern to our order fulfillment example, let's start with the domain class – Slot .

```
public class Slot {
    public Integer slotNo;
    public String eventName;
    public Slot(){}
    public Slot(int slotNo){
        this.slotNo=slotNo;
    }
}
```

We will next write the subsystem service classes.

```
public class AvailabilityService {
    public static boolean isAvailable(Slot slot){
        /*Check database for slot availability*/
        return true;
    }
}

public class BookingService {
    public static void bookEventSlot(Slot slot){
        /*Update slot counter for event */
    }
}


public class PaymentService {
    public static boolean makePayment(){
        /*Connect with payment gateway for payment*/
        return true;
    }
}
```

Let's Create SlotBookingServiceFacade interface which is implemented by SlotBookingServiceFacadeImpl contains actual implementation.

```
public interface SlotBookingServiceFacade {
    boolean slotBook(int slotNo);
}


public class SlotBookingServiceFacadeImpl implements SlotBookingServiceFacade{
    public boolean slotBook(int sno){
        boolean slotBooked=false;
        Slot slot=new Slot();
        slot.slotNo=sno;
        if(AvailabilityService.isAvailable(slot))
        {
            System.out.println("Slot "+ slot.slotNo+" is available for "+ slot.eventName "event");
            boolean paymentConfirmed= PaymentService.makePayment();
            if(paymentConfirmed){
                System.out.println("Payment confirmed...");
                BookingService.bookEventSlot(slot);
                System.out.println("Slot Booked...");
                slotBooked=true;
            }
        }
        return slotBooked;
    }
}
```

Next we will write the controller class – the client of the facade.

```
public class SlotBookingController {
    SlotBookingServiceFacade facade;
    boolean slotBooked=false;
    public void slot_book(int slotNo) {
        slotBooked=facade.slotBook(slotNo);
        System.out.println("SlotBookingController: Slot Booking for event completed. ");
    }
}
```

4. **Template Method Pattern**

The interface method in the abstract class that clients call is the template method. In simple terms, a method that defines an algorithm as a series of steps.

Example :

- In our project, There is one Event which is an abstract class.
- Then there are two subclasses extending Event - TechnicalEvent, NonTechnicalEvent
- But the base Event class will act as a template for both types of vehicle.

Abstract Event class and its subclasses

```
public abstract class Event {
  Integer id;
  String eventName;
  Integer participationAmount;
  Date startDate;
  Date endDate;
}

public class technicalEvent extends Event {
  String eventType="technical";
}

public class nonTechnicalEvent extends Event {
  String eventType="nonTechnical";
}
```

Let’s create Template class

```
public class Template {

    public Event registerEvent(char eventType) {
        Event event=null;
        switch (eventType) {
            case 'A':
                event = new technicalEvent();
                break;

            case 'B':
                event = new nonTechnical();
                break;

            default:
                break;
        }

        return event;
    }
}
```

Thus the following class can use Template class to create instances of any Class.

```
public class Controller {
    public static void main(String[] args) {
        Template template=new Template();
        Event event1=new Event('A');
        Event event2=new Event('B');
    }
}
```

Conclusion:

Thus in this assignment we have successfully identified and implemented GOF patterns.