Assignment 9
Identification and Implementation of GOF pattern

Aim:
Apply any four GOF patterns to refine the Design Model for a given problem description Using effective UML 2 diagrams and implement them with a suitable object-oriented language.

Problem Statement:
● Identification and Implementation of GOF pattern
● Apply any two GOF pattern to refine the 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 the applicability of GOF in the system.
● Implement a system with a pattern,

Theory:

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.

I. 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.

II. 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 behavior 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.

III. 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 that allows several observer objects to see an event.
- State. Allows an object to alter its behavior 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 behavior.
- Visitor. Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Implementation :

I. FACTORY

The Factory Method design pattern is one of the Gang of Four 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.
The Factory Method design pattern is used instead of the regular class constructor for keeping within the solid principles of programming, decoupling the construction of objects from the objects themselves.

In this project:

- In our project, we are using a factory design pattern for a report class. The report is declared as Interface.
- Then created concrete classes implementing the same interface: Department, Subject, Class, Teacher, Student. Then created a factory to generate an object of concrete class i.e. Report_select.
- Use a factory to get objects of the concrete class by passing information as shown in Report_factory.

```
public class Report_select{
     public Report generateReport(String reportName){
         if(reportName == null)
         {
            return null;
         }
         if(reportName.equalsIgnoreCase("DEPARTMENT_REPORT")){
            return new DepartmentReport();
         }
         else if(reportName.equalsIgnoreCase("SUBJECT_REPORT")){
            return new SubjectReport();
         }
         else if(reportName.equalsIgnoreCase("CLASS_REPORT")){
            return new ClassReport();
         }
         else if(reportName.equalsIgnoreCase("TEACHER_REPORT")){
            return new TeacherReport();
         }
         else if(reportName.equalsIgnoreCase("STUDENT_REPORT")){
            return new StudentReport();
         }
         return null;
     }
};
```

```
public class Report_Factory{
     public static void main(String[] args){
        Report_select repSelect = new Report_select();
        Report dep = repSelect.generateReport("DEPARTMENT");
        dep.getReport();
        Report sub = repSelect.generateReport("SUBJECT");
        sub.getReport();
        Report cla = repSelect.generateReport("CLASS");
        cla.getReport();
        Report teacher = repSelect.generateReport("TEACHER");
        teacher.getReport();
        Report stud = repSelect.generateReport("STUDENT");
        stud.getReport();
     }
};
```

II. 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. In this project:

- The User is a public interface with the getDetails() function.
- Then we have a Teacher class that implements a User interface and in this class, we have to override the getDetails() function and the same goes for the Subject Coordinator class and HoD class.
- Details is also a class in which we have created instances of Teacher, Subject Coordinator, and HoD.
- Then again getDetails() is overridden using these instances.
- In the facade class, we have created an instance of Details class and overridden the getDetails using these instances.

```
package facade ;
public interface User {
 void getDetails();
}
public class Teacher_Details implements User {
     public void getDetails(){
        System.out.println("Detail for Teachers");
     }
}
public class HOD_Details implements User {
     public void getDetails(){
        System.out.println("Detail for HOD");
     }
}
public class SubjectCoordinator_Details implements User {
     public void getDetails(){
        System.out.println("Detail for Subject Coordinator");
     }
}
```

```
public class Details {
    private User teacher ;
    private User subject-coordinator ;
    private User hod ;
    public Details(){
        teacher = new Teacher_Details();
        subject-coordinator = new SubjectCoordinator_Details();
        hod = new HOD_Details();
    }
    public void generateTeacherDetails(){
       teacher.getDetails();
    }
    public void generateSubjectCoordinatorDetails(){
       subject-coordinator.getDetails();
    }
    public void generateHODDetails(){
       hod.getDetails();
    }
   }
   public class Facade {
        public static void main(String[] args) {
        Details dt = new Details();
        dt.generateTeacherDetails();
        dt.generateSubjectCoordinatorDetails();
        dt.generateHODDetails();
    }
}
```

III. STRATEGY DESIGN PATTERN

- In Strategy pattern, a class behavior or its algorithm can be changed at run time.
- We create objects which represent various strategies and a context object whose behavior varies as per its strategy object.
- We are going to create a *Context* class which uses *User*.
- In our system, the behaviour will vary dependent upon the designation entered at run time by the User object.

```
public class StrategyPatternDemo {
    public static void main(String[] args) {
    System.out.println("Enter the type of user");
    Scanner myObj = new Scanner(System.in);
    String type = myObj.nextLine();
    if(type=="subj-coordinator")
      Context context = new Context(new Subject-Coordinator() );
    else if(type=="Teacher")
      context = new Context(new Teacher());
    else
      context = new Context(new HOD());
    }
}
```

```
class HOD extends User
{
    void getdetails()
    {
        Scanner myObj = new Scanner(System.in);
        name = myObj.nextLine();
        designation= "hod";
        return;
    }
}

public class Context {
   private User user;

   public Context(User user){
      this.user = user;
   }

   public int executeUser(){
      return user.getdetails();
   }
}
```

```
public interface User
{
    String name,designation,subject,class_;
    void getdetails();
}
class Teacher extends User
{
    void getdetails()
    {
        Scanner myObj = new Scanner(System.in);
        name = myObj.nextLine();
        designation= "teacher";
        subject= myObj.nextLine();
        class_= myObj.nextLine();
        return;
    }
}
class Subject-Coordinator extends User
{
    void getdetails()
    {
        Scanner myObj = new Scanner(System.in);
        name = myObj.nextLine();
        designation= "subject-coordinator";
        subject= myObj.nextLine();
        return;
    }
}
```

IV. SINGLETON DESIGN PATTERN

Singleton pattern involves a single class which is responsible to create an object while making sure that only a single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.
In this project, there is a class HOD which can instantiate only a single object.

```
public interface User
{
    String name,designation,subject,class_;
    void getdetails();
}
class HOD extends User
{
    private static HOD instance = new HOD();
    private HOD(){}
    public static HOD getdetails()
    {
        Scanner myObj = new Scanner(System.in);
        name = myObj.nextLine();
        designation= "hod";
        return;
    }
}
```

Conclusion :
Thus in this assignment, we have successfully identified patterns and their applicability in the system and implemented four GOF patterns viz. Factory, facade, strategy and singleton.