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.

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 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.

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 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 :**

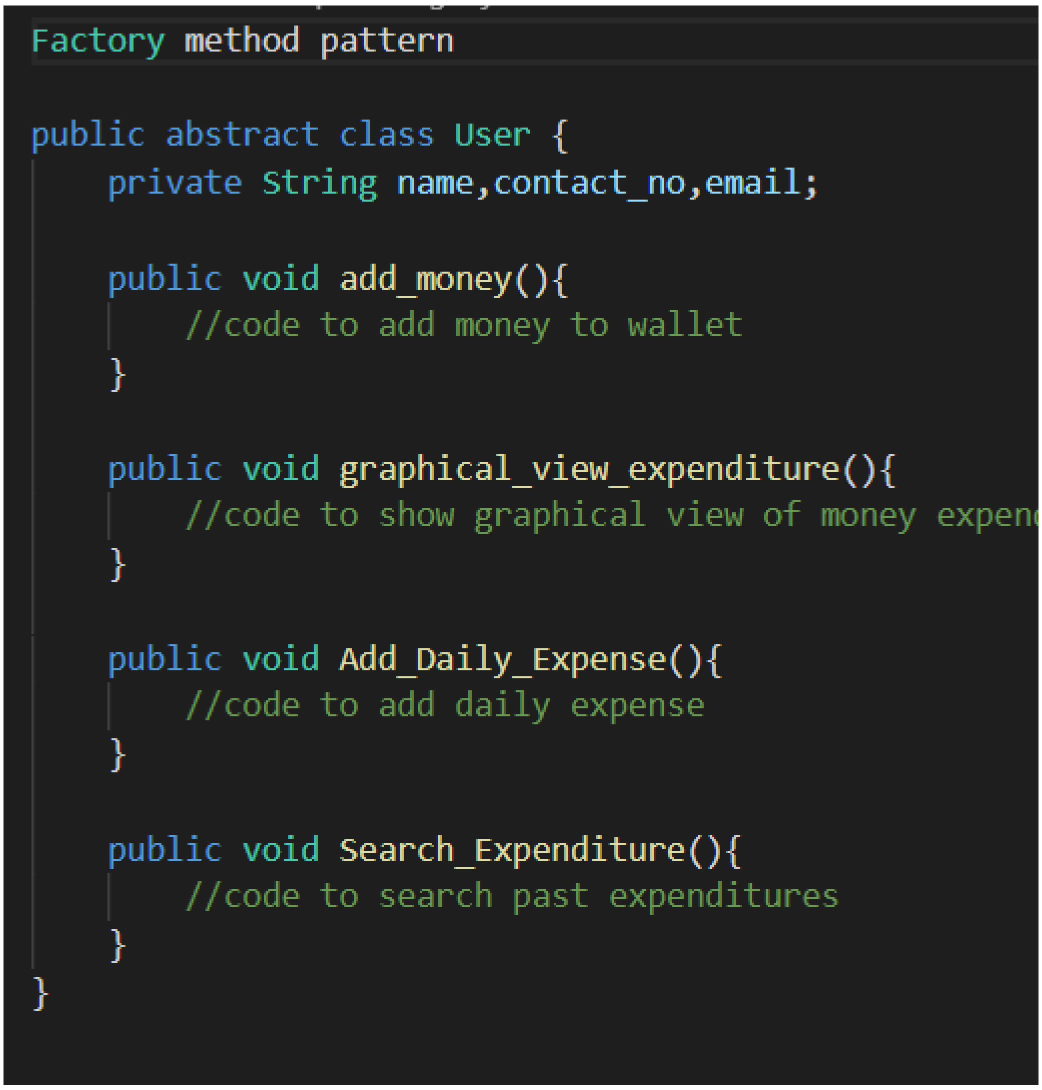
A factory pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.

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.

To apply the factory method pattern , let us first create the abstract `User` class



```
Factory method pattern

public abstract class User {
    private String name,contact_no,email;

    public void add_money(){
        //code to add money to wallet
    }

    public void graphical_view_expenditure(){
        //code to show graphical view of money expend
    }

    public void Add_Daily_Expense(){
        //code to add daily expense
    }

    public void Search_Expenditure(){
        //code to search past expenditures
    }
}
```

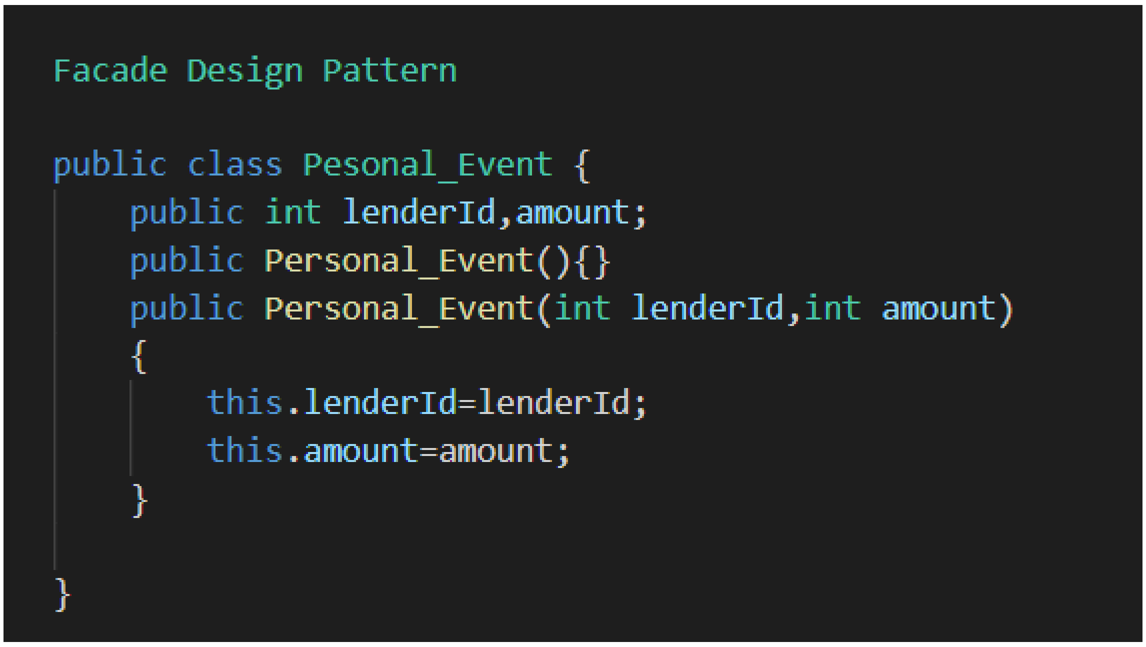
Facade Design Pattern

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 create, the process is complex.
- We require different services before s create Personal Event : BorrowMoneyService, LendMoneyService,validateWalletService, PersoanlEventService
- So, we have created PersonalEventServiceFacade which is very simple for users to create personal event.

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

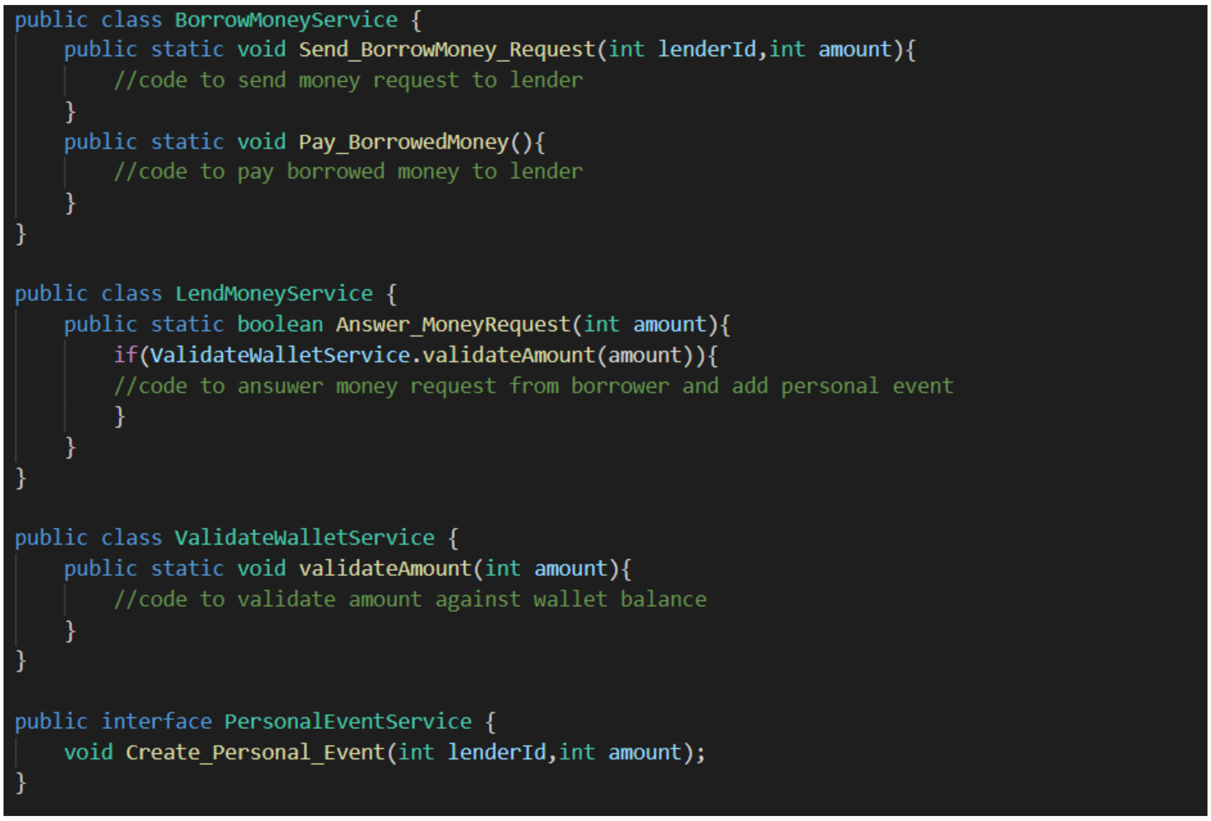


```
Facade Design Pattern

public class Pesonal_Event {
    public int lenderId,amount;
    public Personal_Event(){}
    public Personal_Event(int lenderId,int amount)
    {
        this.lenderId=lenderId;
        this.amount=amount;
    }

}
```

We will next write the subsystem service classes.



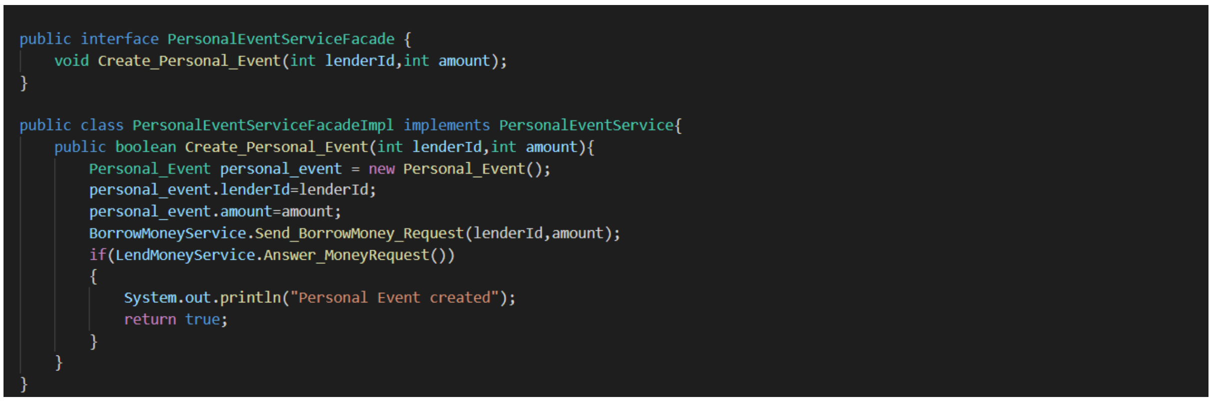
```java
public class BorrowMoneyService {
    public static void Send_BorrowMoney_Request(int lenderId,int amount){
        //code to send money request to lender
    }
    public static void Pay_BorrowedMoney(){
        //code to pay borrowed money to lender
    }
}

public class LendMoneyService {
    public static boolean Answer_MoneyRequest(int amount){
        if(ValidateWalletService.validateAmount(amount)){
        //code to ansuwer money request from borrower and add personal event
        }
    }
}

public class ValidateWalletService {
    public static void validateAmount(int amount){
        //code to validate amount against wallet balance
    }
}

public interface PersonalEventService {
    void Create_Personal_Event(int lenderId,int amount);
}
```

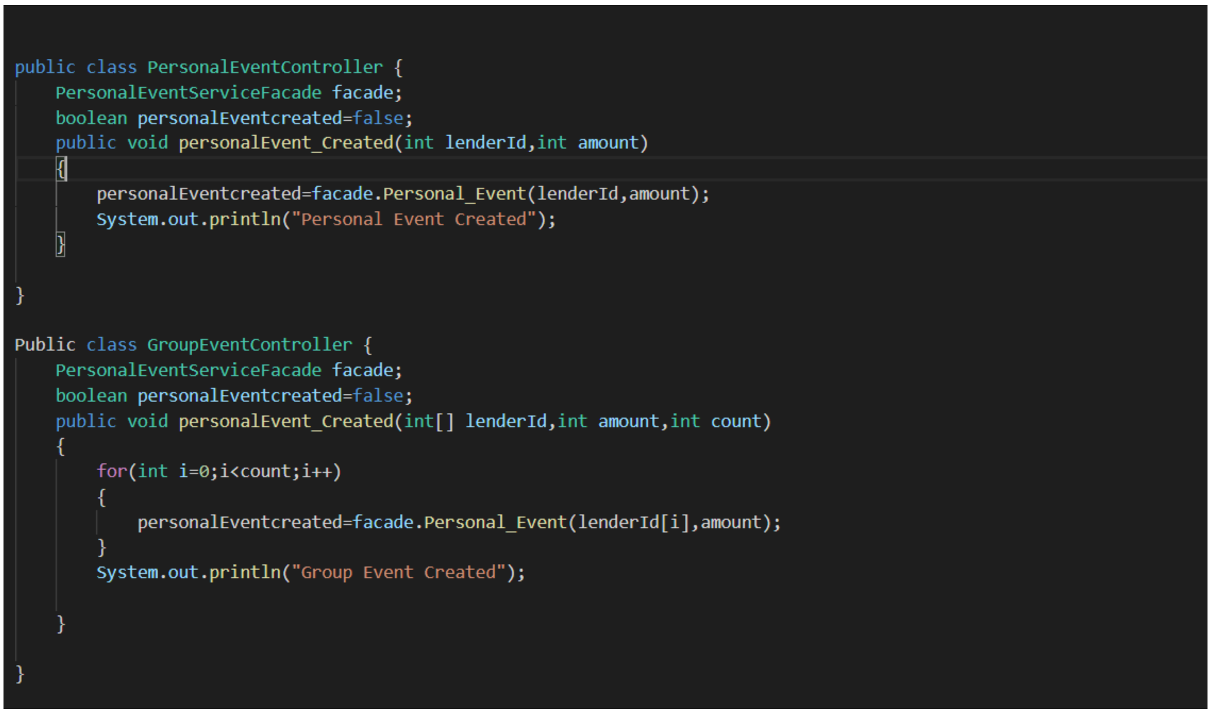
Let’s Create PersonalEventServiceFacade interface which is implemented by PersonalEventServiceFacadeImpl contains actual implementation.



```java
public interface PersonalEventServiceFacade {
    void Create_Personal_Event(int lenderId,int amount);
}

public class PersonalEventServiceFacadeImpl implements PersonalEventService{
    public boolean Create_Personal_Event(int lenderId,int amount){
        Personal_Event personal_event = new Personal_Event();
        personal_event.lenderId=lenderId;
        personal_event.amount=amount;
        BorrowMoneyService.Send_BorrowMoney_Request(lenderId,amount);
        if(LendMoneyService.Answer_MoneyRequest())
        {
            System.out.println("Personal Event created");
            return true;
        }
    }
}
```

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



```
public class PersonalEventController {
    PersonalEventServiceFacade facade;
    boolean personalEventcreated=false;
    public void personalEvent_Created(int lenderId,int amount)
    {
        personalEventcreated=facade.Personal_Event(lenderId,amount);
        System.out.println("Personal Event Created");
    }

}

Public class GroupEventController {
    PersonalEventServiceFacade facade;
    boolean personalEventcreated=false;
    public void personalEvent_Created(int[] lenderId,int amount,int count)
    {
        for(int i=0;i<count;i++)
        {
            personalEventcreated=facade.Personal_Event(lenderId[i],amount);
        }
        System.out.println("Group Event Created");

    }

}
```

Conclusion :

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