Object Oriented Programming Assignment

Group 73

Project-3
Audi Ticket Booking

Group Members:

AMAN PHOGAT

SAMARTH GANDOTRA

SATYAM SAXENA

VEDANG BHUPESH SHENVI NADKARNI

PLAGIARISM STATEMENT

We certify that this assignment is our own work, based on my/our personal study and/or research and that We have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment has not previously been submitted for assessment in any other unit, except where specific permission has been granted from all unit coordinators involved, or at any other time in this unit, and that I have not copied in part or whole or otherwise plagiarized the work of other students and/or persons.

Name : AMAN PHOGAT,SAMARTH GANDOTRA,SATYAM SAXENA,VEDANG BHUPESH SHENVI NADKARNI

Date : 04-12-2022

Drive link:
https://drive.google.com/drive/folders/1kL0bCGQXLvzC41ZNO0EM_2WDqjAcMSwT?usp=sharing

Contribution Table

	ID. No.	Name	Contributions
1)	2021A7PS2437	Samarth Gandotra	implemented classes, Documentation (UML dig., design analysis)
2)	2020A3PS1781P	Satyam Saxena	implemented classes, file handling, documentation (SOLID analysis, UML dig.)
3)	2020B5A70897	Vedang Nadkarni	made boiler plate, implemented classes, debugging code, Multithreading

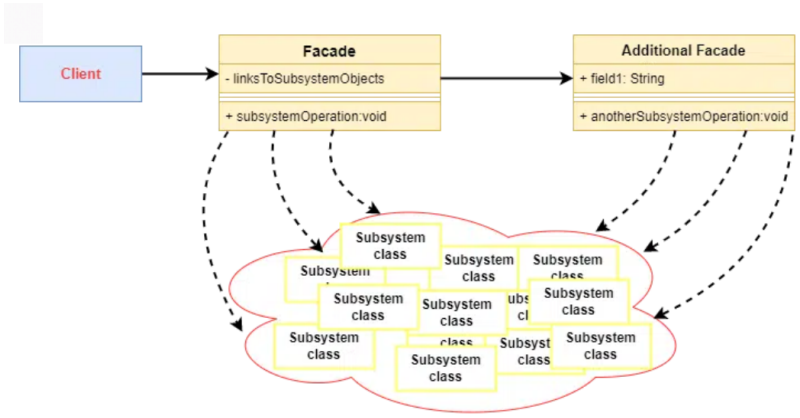
Design Patterns

Usage of Facade Design Pattern

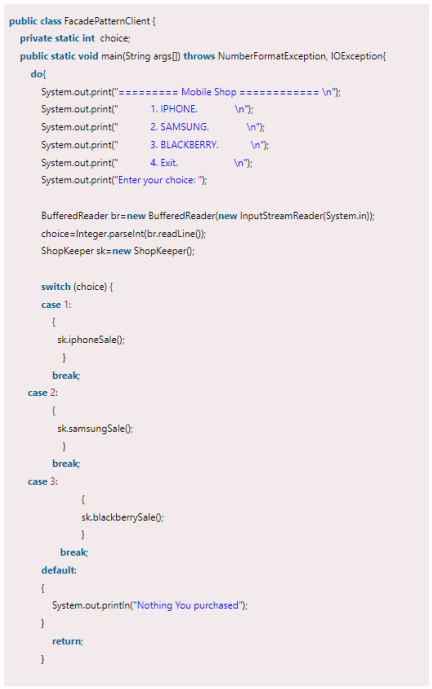
Facade Design Pattern is a part of the Structural Design Patterns. Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing systems to hide its complexities.

A façade is a class that gives a straightforward interface to a complicated subsystem with many moving pieces. When compared to interacting directly with the subsystem, a facade may have limited capabilities. It does, however, only include the elements that clients care about. This design pattern uses a single class to provide client-side simplified methods while delegating calls to existing system classes' functions.

So Facade Design Pattern is quite useful as it gives a common interface for the given task and we can implement that interface to create many subsystems which contains details about many specific instance variables. A suitable diagram is attached below which explains the Facade Design Pattern followed by an example:-



Facade design pattern



```
public class FacadePatternClient {
   private static int  choice;
   public static void main(String args[]) throws NumberFormatException, IOException{
      do{
         System.out.print("========= Mobile Shop ============ \n");
         System.out.print("          1. IPHONE.            \n");
         System.out.print("          2. SAMSUNG.           \n");
         System.out.print("          3. BLACKBERRY.        \n");
         System.out.print("          4. Exit.              \n");
         System.out.print("Enter your choice: ");

         BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
         choice=Integer.parseInt(br.readLine());
         ShopKeeper sk=new ShopKeeper();

         switch (choice) {
         case 1:
            {
             sk.iphoneSale();
              }
            break;
      case 2:
            {
             sk.samsungSale();
              }
            break;
      case 3:
                  {
                  sk.blackberrySale();
                  }
             break;
         default:
         {
            System.out.println("Nothing You purchased");
         }
            return;
         }
```

```
public interface MobileShop {
   public void modelNo();
   public void price();
}
```

Step 2

Create a **Iphone** implementation class that will implement **Mobileshop** interface.

File: Iphone.java

```
public class Iphone implements MobileShop {
   @Override
   public void modelNo() {
      System.out.println(" Iphone 6 ");
   }
   @Override
   public void price() {
   System.out.println(" Rs 65000.00 ");
   }
}
```

Step 3

Create a **Samsung** implementation class that will implement **Mobileshop** interface.

File: Samsung.java

```
public class Samsung implements MobileShop {
   @Override
   public void modelNo() {
   System.out.println(" Samsung galaxy tab 3 ");
   }
   @Override
   public void price() {
      System.out.println(" Rs 45000.00 ");
   }
}
```

Step 4

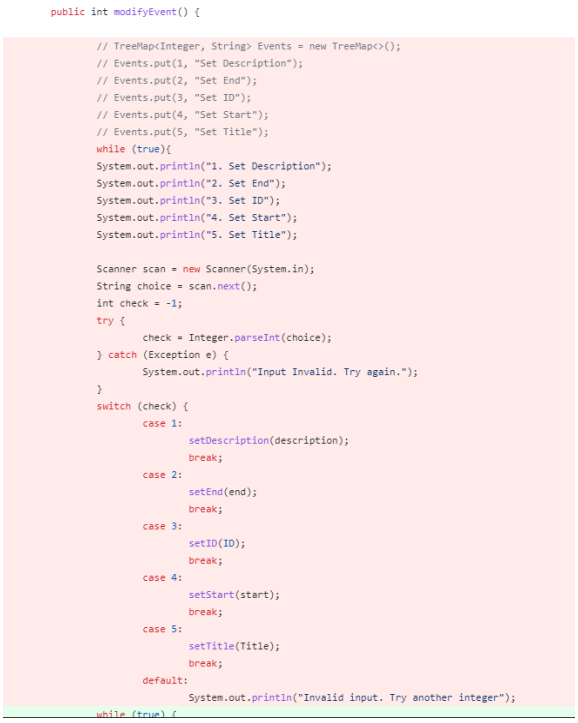
Create a **Blackberry** implementation class that will implement **Mobileshop** interface .

File: Blackberry.java

```
public class Blackberry implements MobileShop {
   @Override
   public void modelNo() {
   System.out.println(" Blackberry Z10 ");
   }
   @Override
   public void price() {
      System.out.println(" Rs 55000.00 ");
   }
}
```

So the figures shown above here are examples of Facade Design Patterns and it clearly shows how a public interface MobileShop is created following which all the subsystems are created. It clearly shows how systematic the above code is, hiding all the complexities from the Client. This also reduces the chances of having bugs in the code reducing the work of debugging.

So this is somewhat we have tried to implement in our project but not exactly Facade Design. Have a look at the code attached below:-



```
public int modifyEvent() {

        // TreeMap<Integer, String> Events = new TreeMap<>();
        // Events.put(1, "Set Description");
        // Events.put(2, "Set End");
        // Events.put(3, "Set ID");
        // Events.put(4, "Set Start");
        // Events.put(5, "Set Title");
        while (true){
        System.out.println("1. Set Description");
        System.out.println("2. Set End");
        System.out.println("3. Set ID");
        System.out.println("4. Set Start");
        System.out.println("5. Set Title");

        Scanner scan = new Scanner(System.in);
        String choice = scan.next();
        int check = -1;
        try {
                check = Integer.parseInt(choice);
        } catch (Exception e) {
                System.out.println("Input Invalid. Try again.");
        }
        switch (check) {
                case 1:
                        setDescription(description);
                        break;
                case 2:
                        setEnd(end);
                        break;
                case 3:
                        setID(ID);
                        break;
                case 4:
                        setStart(start);
                        break;
                case 5:
                        setTitle(Title);
                        break;
                default:
                        System.out.println("Invalid input. Try another integer");
        while (true) {
```

So I have attached a small snippet of our code which somewhat looks similar to the other example, but is not same, here we have made a separate class altogether to define all the methods such as setDescription, setEnd, setID, setStart, setTitle rather than making a common interface and then extending outwards. Then we have called different methods using the switch case in the modiffy event class.We have applied this in many parts of the code. After knowing all the design patterns possible, I would say that I would want to implement the Facade Design Pattern to my Project rather than not using a Design Pattern altogether.

S.O.L.I.D ANALYSIS

1.S.Single Responsibility

We have created multiple classes related to events.All of them have different responsibilities. This is done in order to reduce dependencies to a single class.This made debugging and testing code easier.
For example, the class Event has details of all the events which are present by default.The class EventViewer on the other hand uses **file handling** to add more events from admin if needed and then lists them out .The class Bookable events checks if a particular event is bookable then it books it according to the request sent.The class Booker thread uses **multithreading** to take booking requests .The class Modifyable events allows us to modify event details like end time,description,etc. On the other hand class Seat contains details of the seat like its booking status.

2.O.Open for Extension, Closed for Modification

Inheritance has been used at multiple places in the code because while creating classes we had some basic functions that the program should do, but while implementing the code we realized that we will need to implement additional functionality.So rather than modifying the current class we decided to extend/implement the current class/interface.For example, while coding in the boiler plate we made a class called bookable event whose job was to book the event according to the request sent.But then we realized sometimes if request to book an event is given after the event is over or the request from admin requires to change the specific details like starting time ,ending time,etc then we will need to add additional functionality.To implement this we had two options either to modify bookable event class or create a new class which extends the modifiable events.So we took the

design decision to extend the class so that we do not mess up an already running bookable event class.

3.**L.Liskov Substitution**

We have created certain methods in the parent class which are useless in the context of the child class which concludes that our code is not in agreement with the Liskov substitution principle.For example in the context of three classes GeneralUser,Admin,User, in the class GeneralUser we have implemented the functionality of login and signup for a user . But the functionality of signup is not valid in the context of an admin.Which is completely in disagreement of the Liskov Substitution principle because we cannot use the child class in a function which uses the parent class and expects a valid output.

4.I.Interface Segregation

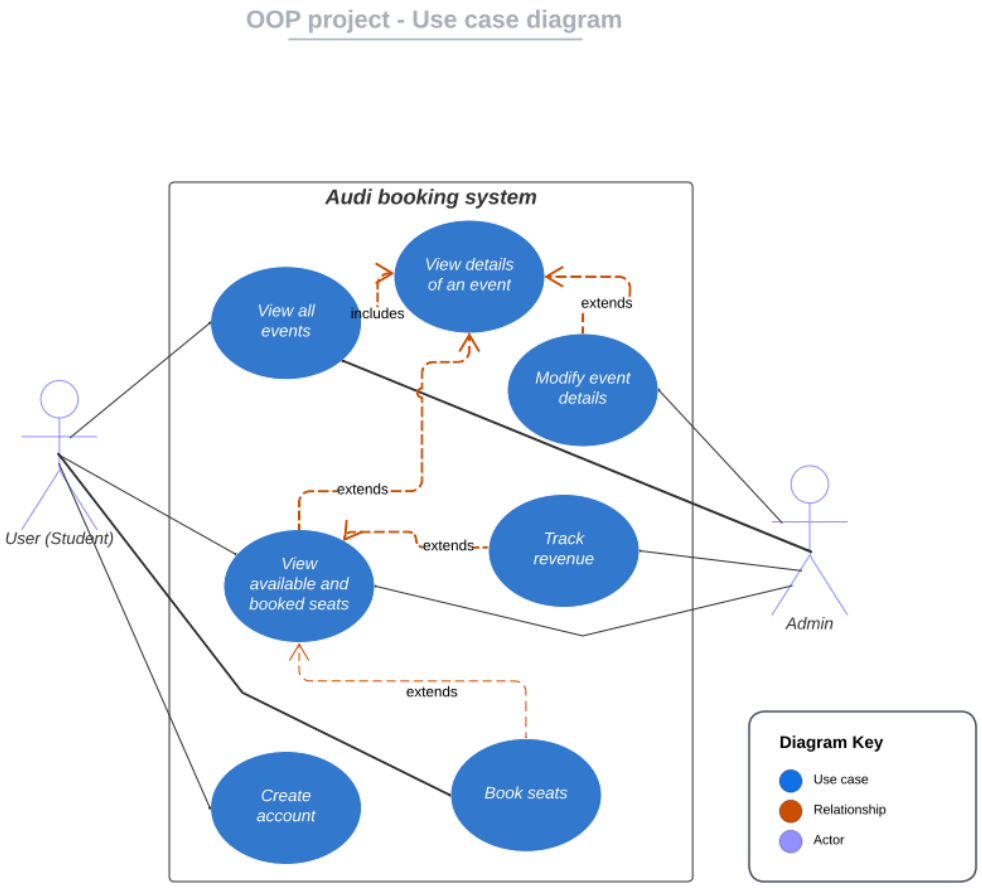
In the context of GeneralUser,Admin,User we have implemented various functions in the interface GeneralUser.While in the child class(which implements this GeneralUser class)i.e. Admin,User, we have implemented the General class's functions inside a single class rather than segregating them into further classes.Hence our code is not in agreement with Interface Segregation.

5.D.Dependency Inversion

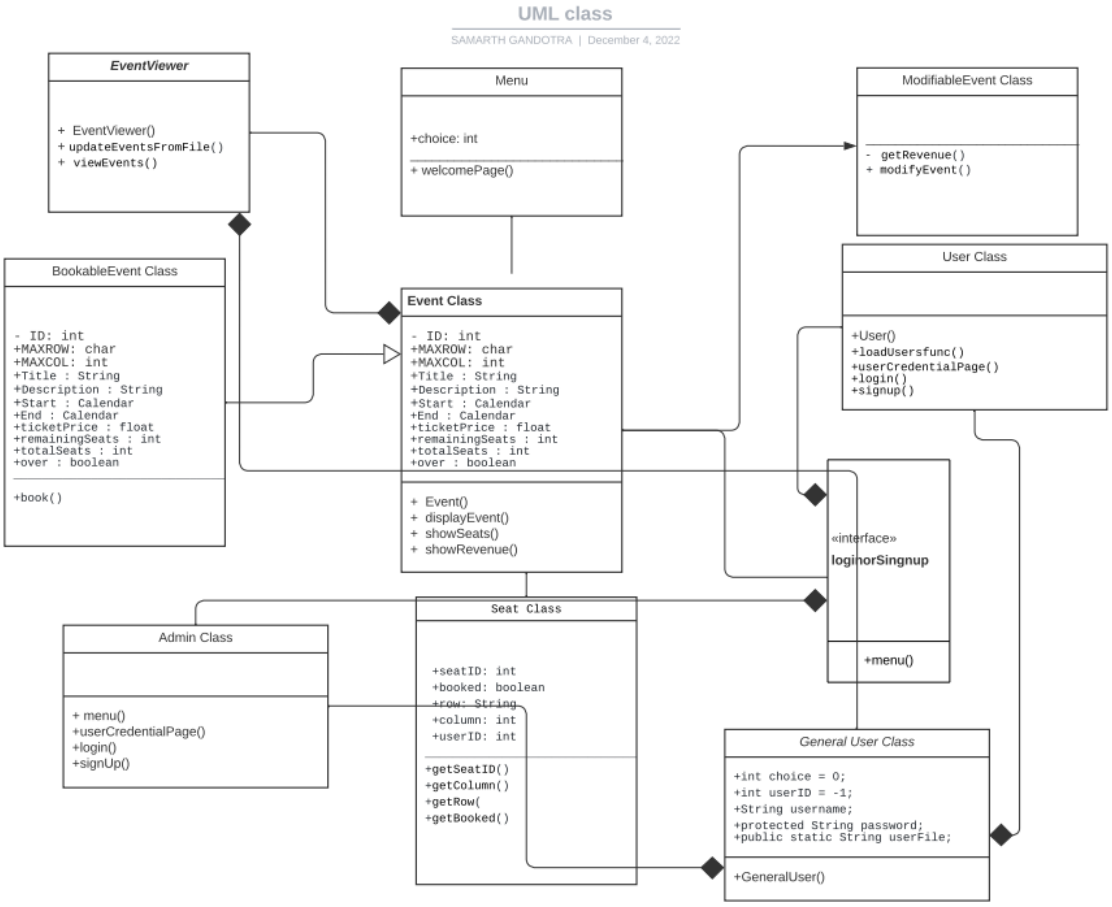
In our code there are many instances in which the higher level modules are dependent on lower level modules which makes the higher level modules not fit for re-use and harder to modify.For example in case of these of the abstract class LoginOrSignup there's another class event viewer .Both of them are inherited by the class GeneralUser.The GeneralUser class is then inherited by both User class and Admin class.In case of DIP implementation of the abstract class loginOrSignup should have been directly done by User and Admin classes instead of extending the GeneralUser class(which implements LoginOrSignup).

UML DIAGRAMS

Use case Diagram:-



Class Diagram



Sequence Diagram