Gandaki College of Engineering and Science.

Name :- Sroachitta Gurung
Roll No:- 47
OOSD Lab :- 1, 2, 3, 4, 5, 6, 7
Submitted to :- Er Rajendra Bdr Thapa
Sem : 6th Sem

Lab Report

TITLE: To develop user requirement and system requirement of MINOR - Project II

THEORY:

UseCase is a description of the ways in which a user can interacts with a system or product. It may establish the success scenarios, the failure scenarios, and any critical variations or exceptions.

System Requirements are the configuration that a system must have in order for a hardware or slw application to run smoothly and efficiently.

functional requirement are product features or function that developers must implement to enable users to accomplish their task.

figure:-

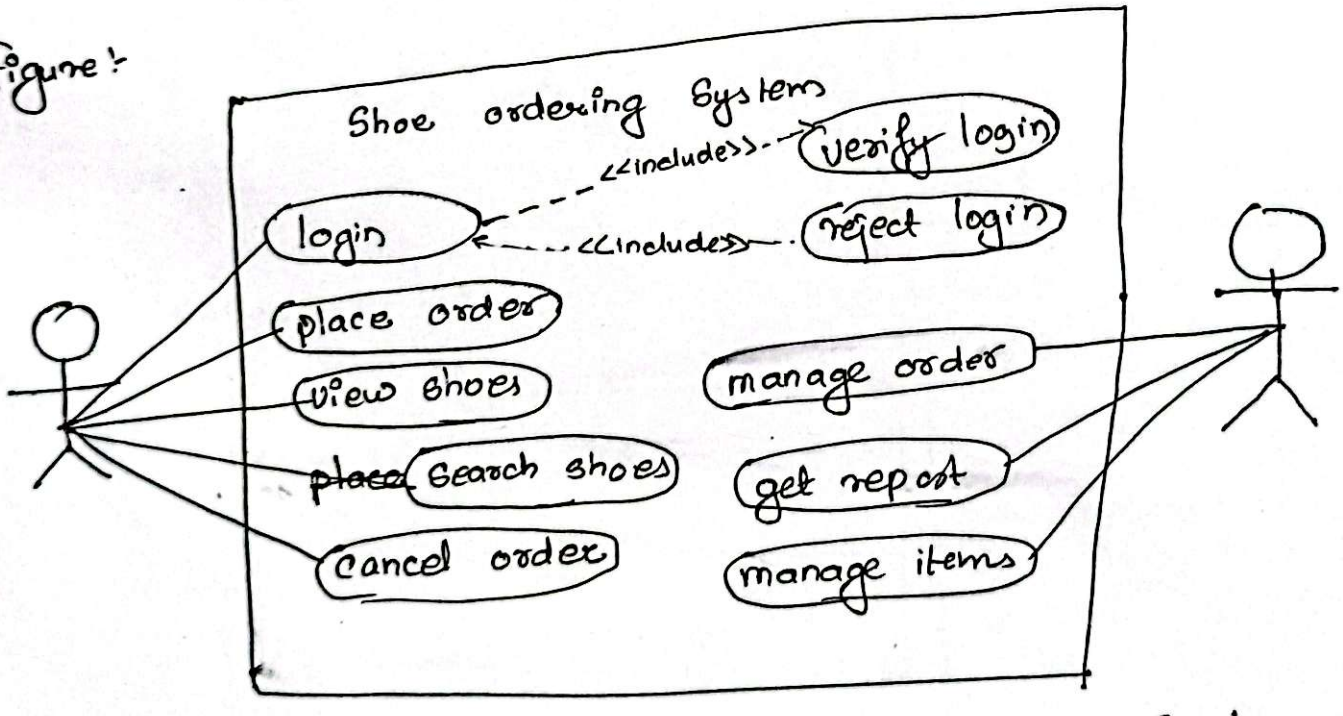


fig:- UseCase Diagram of the System

System Sequence Diagram

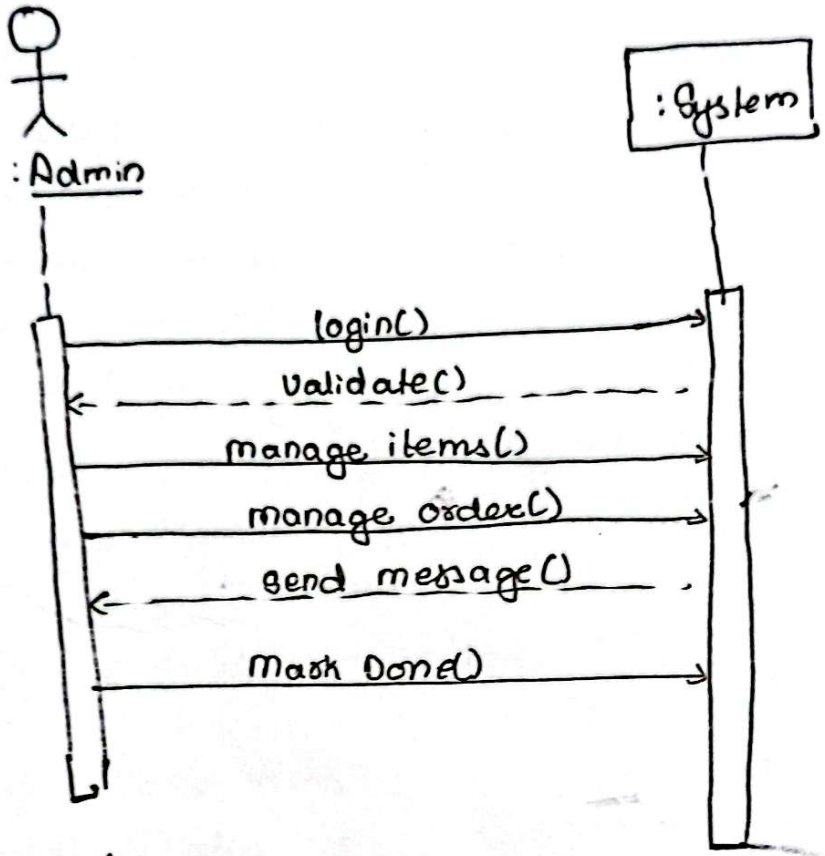


fig: SSD for admin interaction.

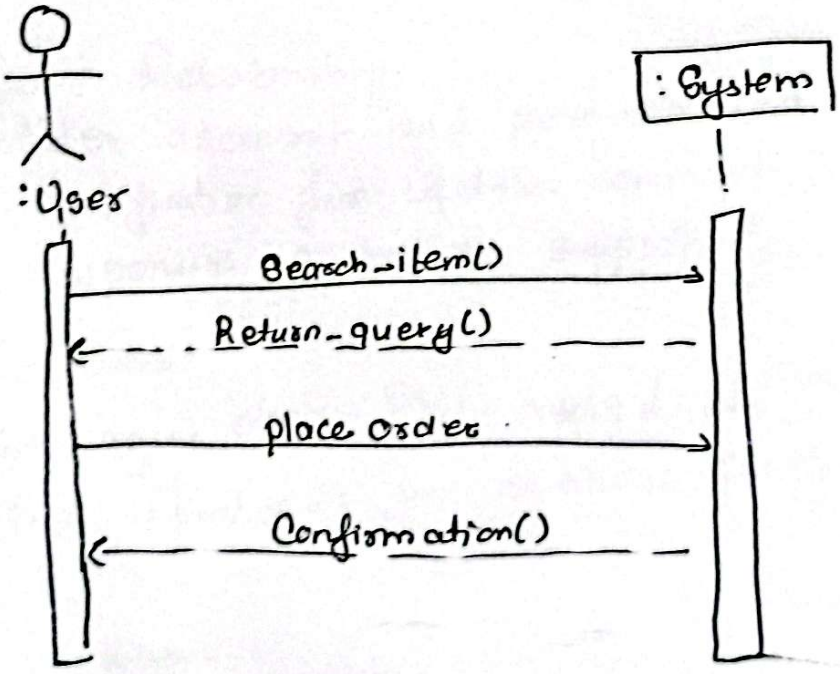


fig: SSD for User Interaction

Lab 8:

Roll No:

Business Canvas Model for "Sole-Mate"

Introduction:

Creating business canvas model can be helpful exercise to outline. Key element of your project using PHP. The Business model canvas is a ~~flat~~ strategic. magement tool that provides a visual framework for developing. describing and describing and analysing business-Model.

Below is a simplified versionfor our project:

(1) Customer segments:
- Buyer and Sellers.
- Professional occupation.

(2) Value Proposition
- Social Media
  Connect and explore new ideas.
  Share updates, thoughts & experiences.

(3) Customer Relationship
  User account and personalized profile.
  Notification for update. comment & interaction
  Responsive customer support for inquires & issues.

(4) Key Resources:
  Technology stack: PHP. mysql. etc.
  Skilled developer for continous platform maintenance and update.

(5) Key Activities.
- Platform dependent & Maintenance
- Content mgmt and moderation
- Continous improvement based on user feedback

Conclusion:

It is a dynamic tool that should evolve as social media and shop-selling platform grow and adopt market and user needs.

3) Creational Design Pattern : Lab 3

Theory :-

They are concerned with the way of creating objects The design patterns are used when a decision must be made at time of instantiation of class.

In Java, we create objects

StudentRecord s1 = new StudentRecord();

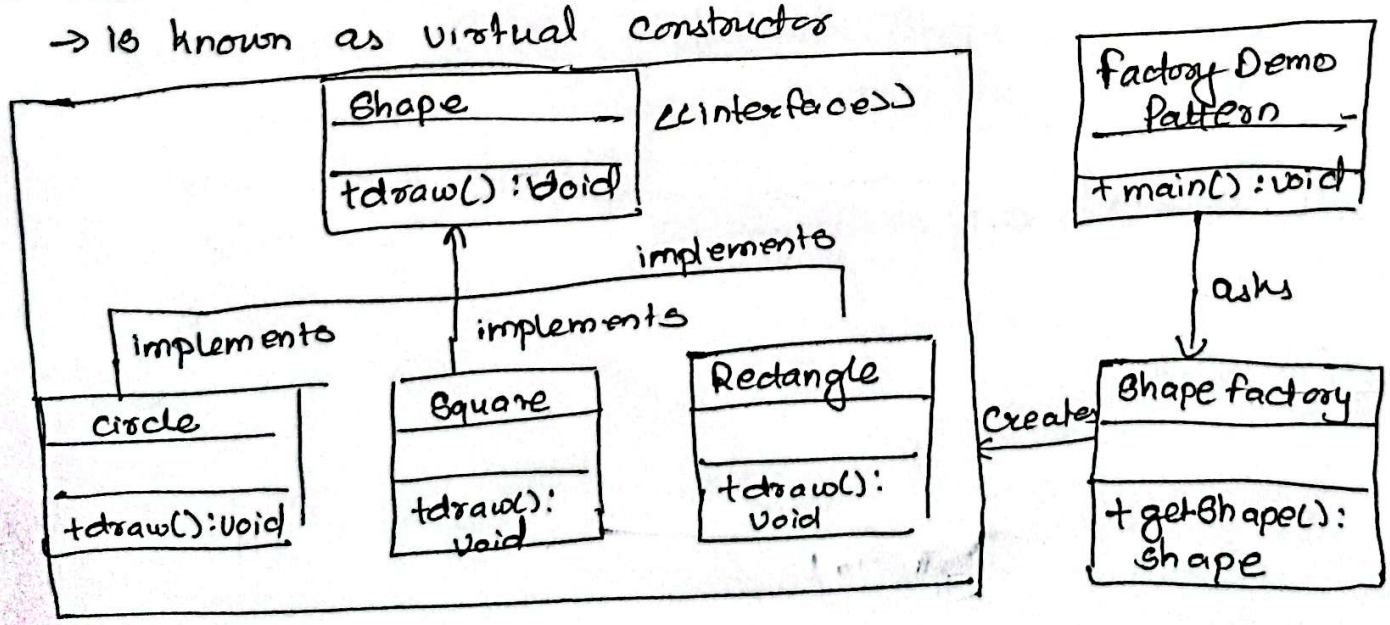
Types of Creational Design pattern

1) factory Method Pattern
2) Abstract factory Pattern
3) Singleton Pattern
4) Prototype Pattern
5) Builder Pattern
6) Object - Pool Pattern

1) factory Method Pattern.

They define abstract class for creating an object but let the subclasses decide which class to instantiate.

→ is known as virtual constructor



Step 1:

Create a interface

Shape.java

```
public interface Shape {
    void draw();
}
```

Step 2:

Rectangle.java

```
public class Rectangle implements Shape {
    @override
    public void draw() {
    System.out.println("Inside Rectangle :: draw() method.");
    }
}
```

Square.java

```
public class Square implements Shape {
    @override
    public void draw() {
    System.out.println("Inside Square ::draw() method.");
    }
}
```

Circle.java

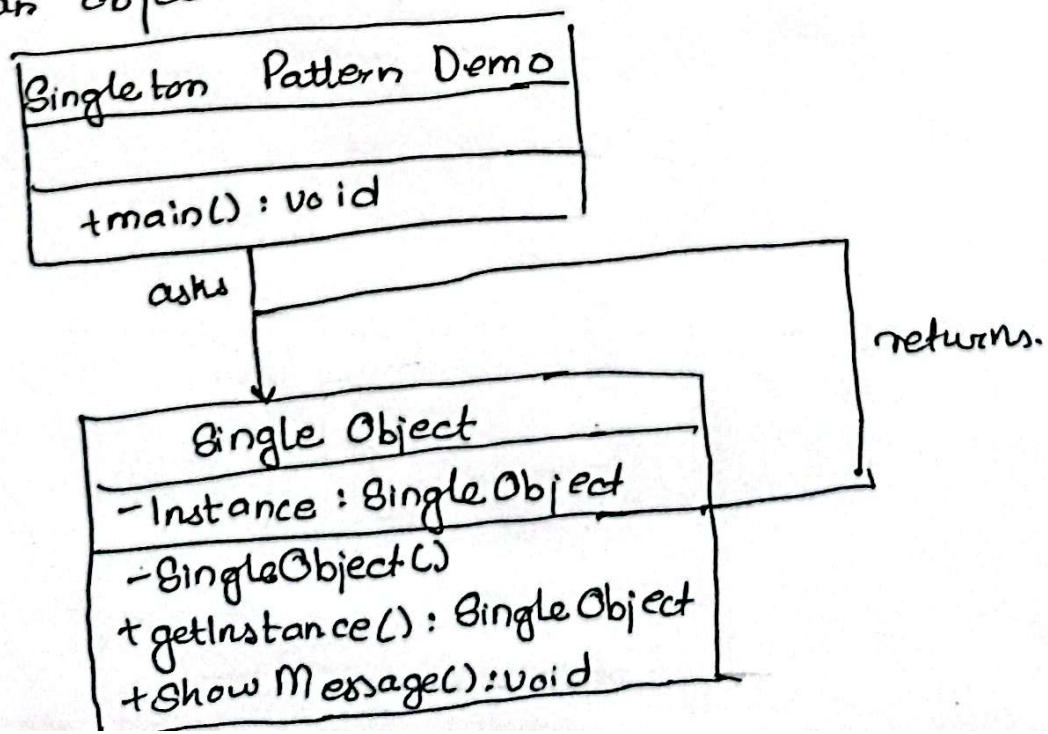
```
public class Circle implements Shape {
    @override
    public void draw() {
    system.out.println("Inside Circle ::draw() method");
    }
}
```

ShapeFactory.java

```
public class ShapeFactory {
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      }
      else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }
      else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

② Singleton pattern.

It is the simplest design patterns in Java. It comes under creational pattern because it is the best way to create an object



Step 1:

SingleObject.java

```
public class SingleObject {
   private static SingleObject instance = new SingleObject();
private SingleObject() {}
  public static SingleObject getInstance() {
     return instance;
   }
   public void showMessage(){
      System.out.println("Hello World");
   }
}
```

Step 2:

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public ~~class~~ static void main(String[] args){
   SingleObject object = SingleObject.getInstance();
   object.showMessage();
   }
}
```

Lab 4: To know about Behavior Design Pattern

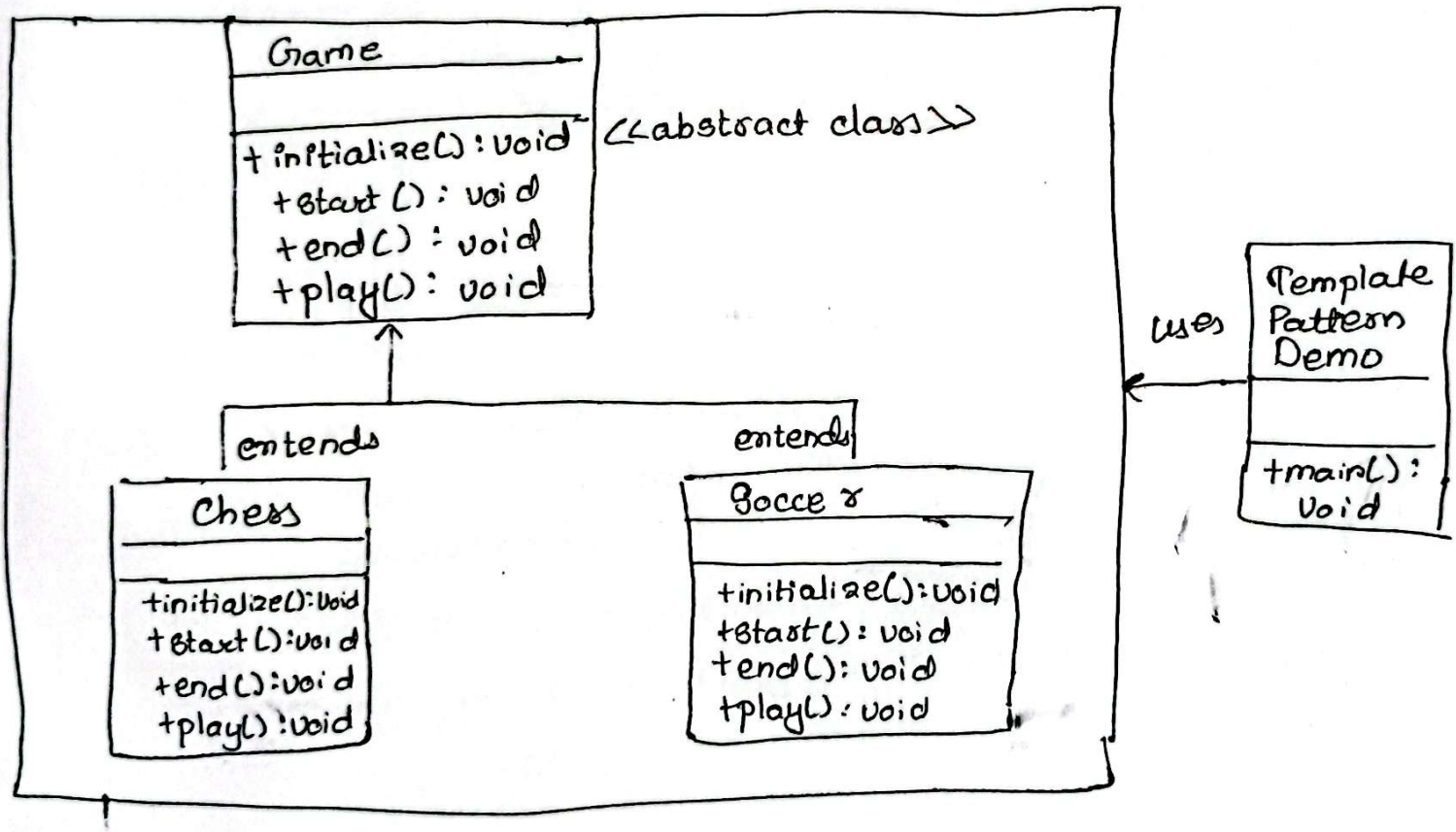
THEORY:

Behavioral Design pattern are design patterns that identify common communication patterns between objects and realise these patterns. By, increasing flexibility in carrying out this communication.

① Observer Pattern

→ It provides the support for broad-cast communication.

→ It describes the coupling betn object and observer.



Step 1:

```
public abstract class Game{
        abstract void initialize();
        abstract void start();
        abstract void end();
public final void play(){
        initialize();
        start();
        end();
    }
}
```

Step 3:

```
public class ChessGame extends Game {
    @Override
    void initialize() {
    System.out.println("Chess game Initialized! Start
                        playing.");
    }
    @Override
    void start() {
    System.out.println("Game Started. Welcome to in
                        the chess game!");
    }
    @Override
    void end() {
    System.out.println("Game finished!");
    }
}
```

TemplatePatternDemo.java.

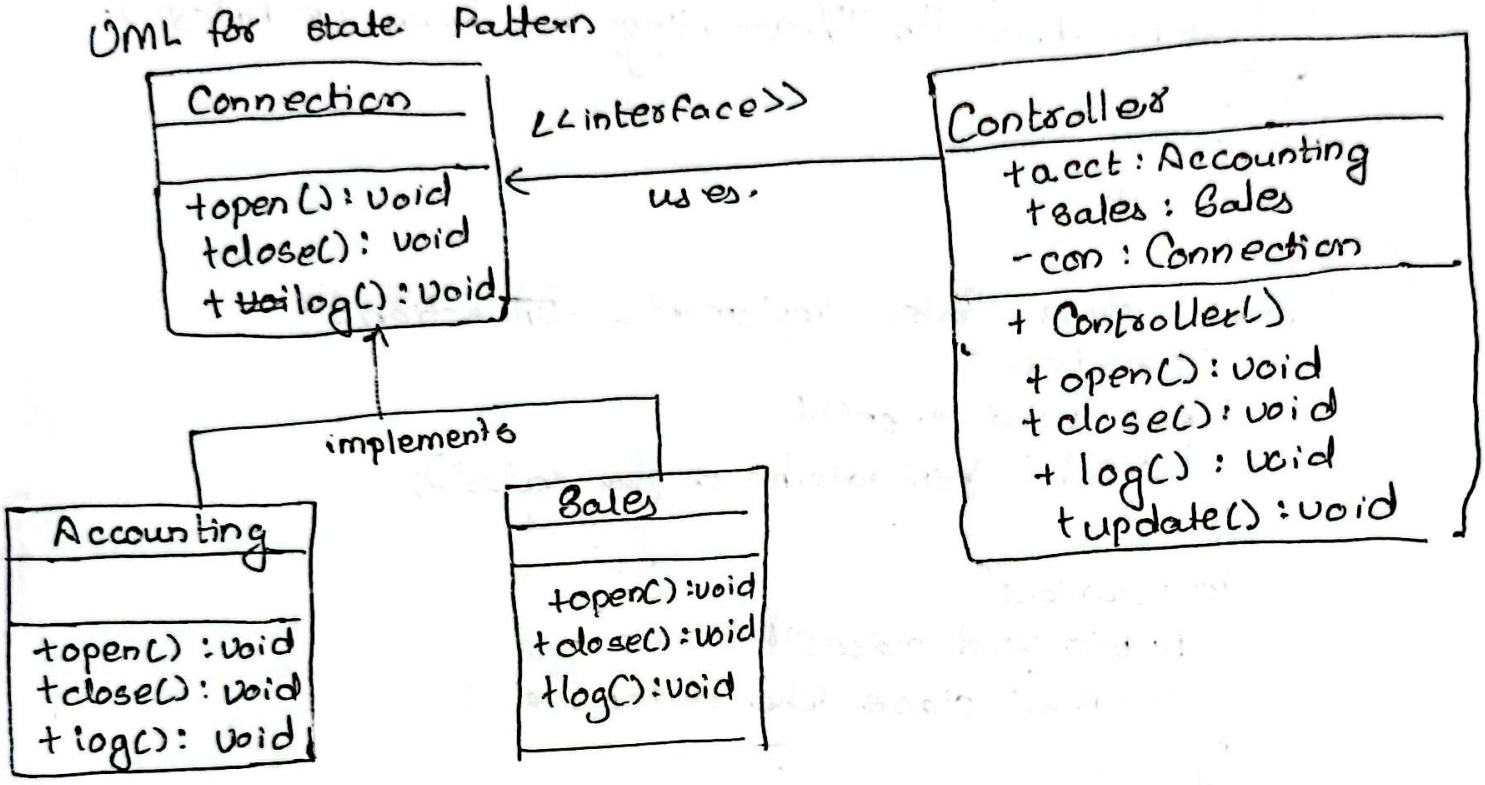
```
public class TemplatePatternDemo {
    public static void main(String[] args) throws
            exception {
        Class c = Class.forName(args[0]);
        Game game = (Game) c.newInstance();
        game.play();
    }
}
```

State Pattern:

→ The class behavior changes based on its state.
In State Pattern we create objects which represents various states and a content object.

→ Object for states.

UML for state Pattern



Step 1:

```
public interface Connection{
    public void open();
    public  "   close();
    public  "   log();
    public void update();
}
```

Step 2:

```
//Create a accounting Class.
public class Accounting implements Connection{
    @override
    public void open(){
     System.out.println("Open database for accounting");
    }
    @override
    public void close(){
     System.out.println("close the database");
    }
```

```
public void log(){
 System.out.println("log activities");
}
@override
public void update(){
 System.out.println("Accounting has been updated");
}
}
```

Step 8:

```
public class Sales implements Connection{
  @Override
  public void open(){
   S.o.p.l("Open database for sales");
  }
  @Override
    public void close(){
    S.o.p.l("close the database");
  }
  @override
     public void log(){
     System.out.println("log activities");
   }
  @-override
     public void update(){
      S.o.p.ln("Sales has been updated");
     }
    }
```

```
public class Controller{
    public static Accounting acct;
    public static Sales sales;
    private static Connection con;
    Controller(){
        acct = new Accounting();
        Sales = new Sales();
        mgmt = new Mgmt();
    }
    public void setAccountingConnection(){
        con = acct;
    }
    public void open(){
        con.open();
    }
    //same as log(), update()
}
```

) TITLE: TO KNOW ABOUT STRUCTURAL DESIGN PATTERN.

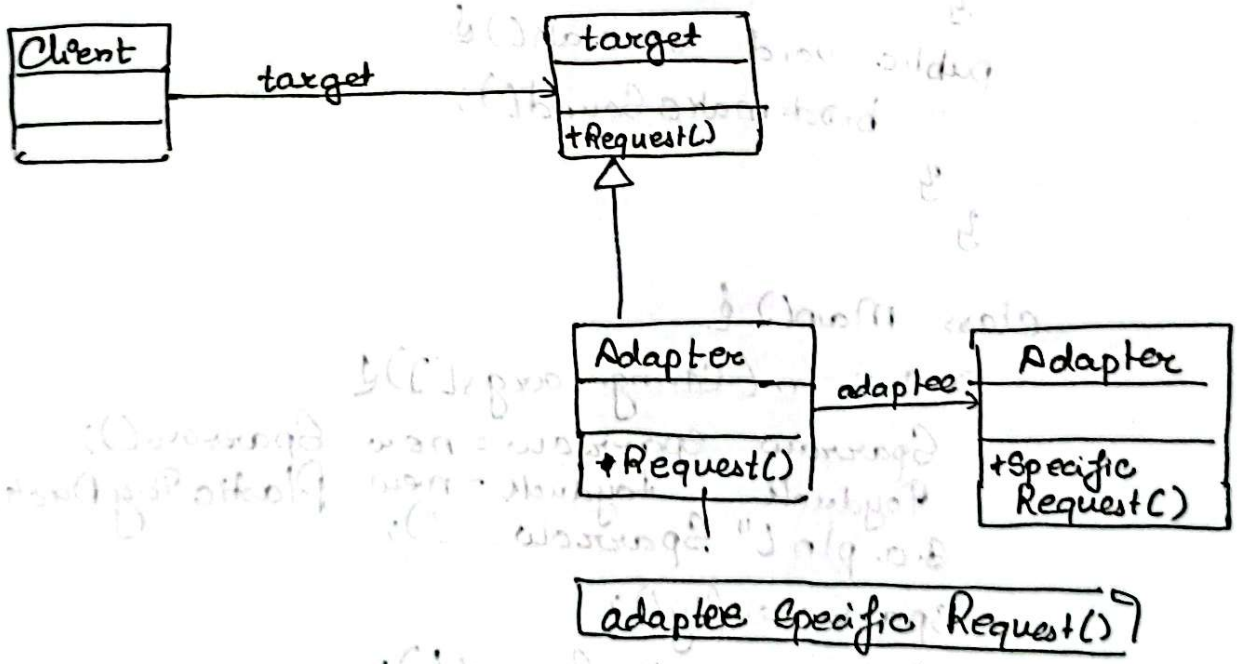
THEORY :

Structural design pattern are concerned with how classes and objects can be composed to form larger structures.

→ Adapter Pattern
→ Proxy Pattern
→ Decorator Pattern

) Adapter Pattern

It allows reusability of existing functionality.
It allows two or more previously incompatible objects to interact.



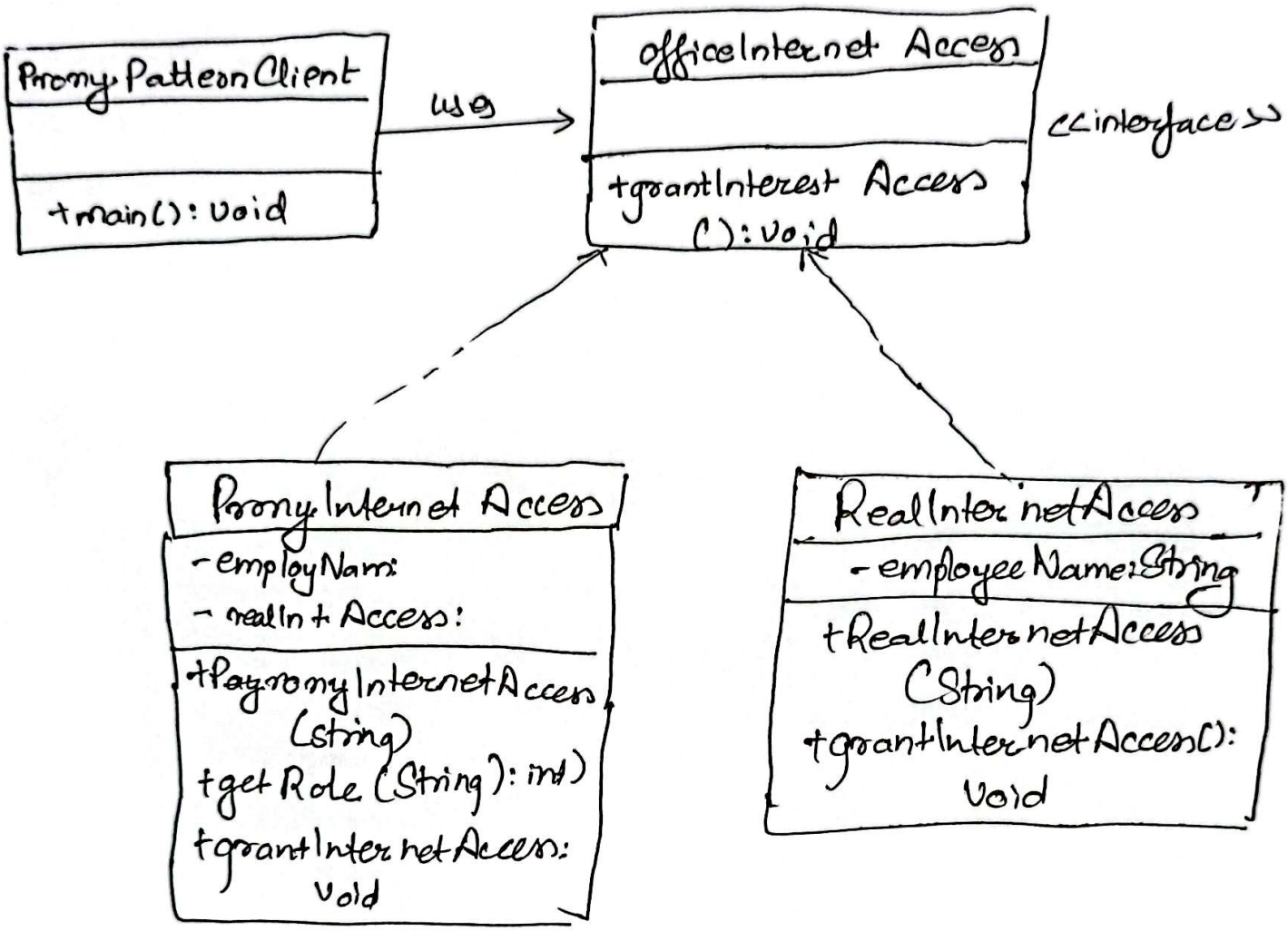

```
Interface Bird {
    public void fly();
    public void makeSound();
}
class Sparrow implements Bird
{
  public void fly() {
    S.O.P.ln ("flying");
  }
  public void makeSound() {
    S.o.pln ("Chirp Chirp");
```

```
interface ToyDuck
{
  public void speak();
                squeak
}
class PlasticToyDuck implements ToyDuck {
    public void squeak()
    {
     s.o.p.ln("Squeak");
    }}
class BirdAdapter implements ToyDuck
  {  Bird bird;
    public BirdAdapter(Bird Bird)
     {
      this.bird = bird;
     }
    public void squeak(){
        bird.makeSound();
     }
    }

  class Main() {
    p.s.v.m.(String args[]){
     Sparrow sparrow = new Sparrow();
     Toyduck toyduck = new PlasticToyDuck();
     s.o.pln(" Sparrow ...");
     sparrow.fly();
     Sparrow.makeSound();
    }
}
```

① Proxy Pattern.

an object representing another object.



ab 6: MVC Implementation.

HEORY:

The Model View Controller design pattern specifies that n application consist of a data model, presentation information nd control information

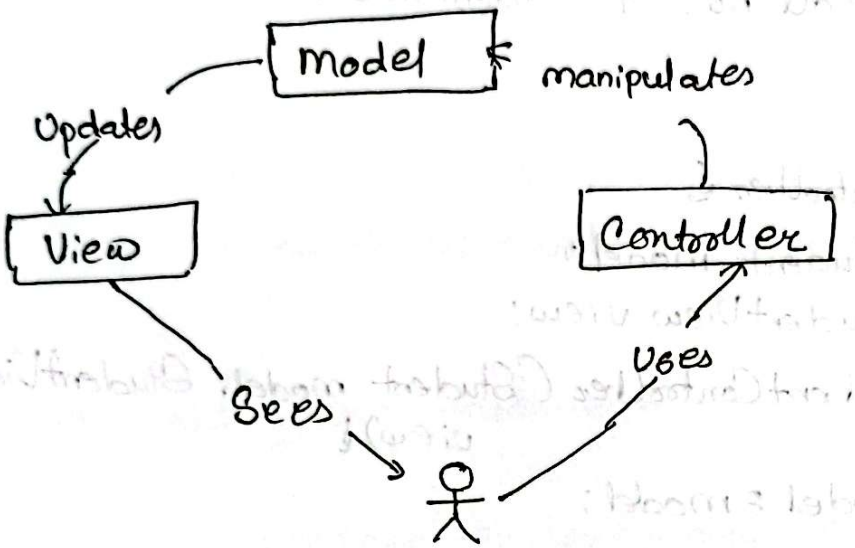


fig: MVC

CODE:

```
class Student {
    private string rollno, string name;
public string getRollNo() {
    return rollNo;
  }
public void setRollNo( string rollNo)
  {
    this.rollNo = rollNo;
  }
public string getName() {
    return name;
}
public void setName (String name)
  {
this.name = name;
  }
}
```

```
class StudentView
{
public void printStudentDetails (String studentName,
    String studentRollNo){
  S.o.pln (" Student :");
  S.o.pln (" Name :" + studentName);
  S.o.p.ln ("Roll no :" + studentRollNo);
}
}

class StudentController {
    private Student model;
    private StudentView view;
    public StudentController (Student model, StudentView
                                        view){
      this.model = model;
      this.view = view;
    }
    public void setStudentName (String name) {
      model.setName(name);
    }
    public String getStudentName()
    {
      return model.getName();
    }
    public void setStudentRollNo (String rollNo)
    {
      model.setRollNo(RollNo);
    }
    public String getStudentRollNo(){
      return model.getRollNo();
    }
    public void updateView(){
      view.printStudentDetails (model.getName(),
      model.getRollNo());
    }}
```

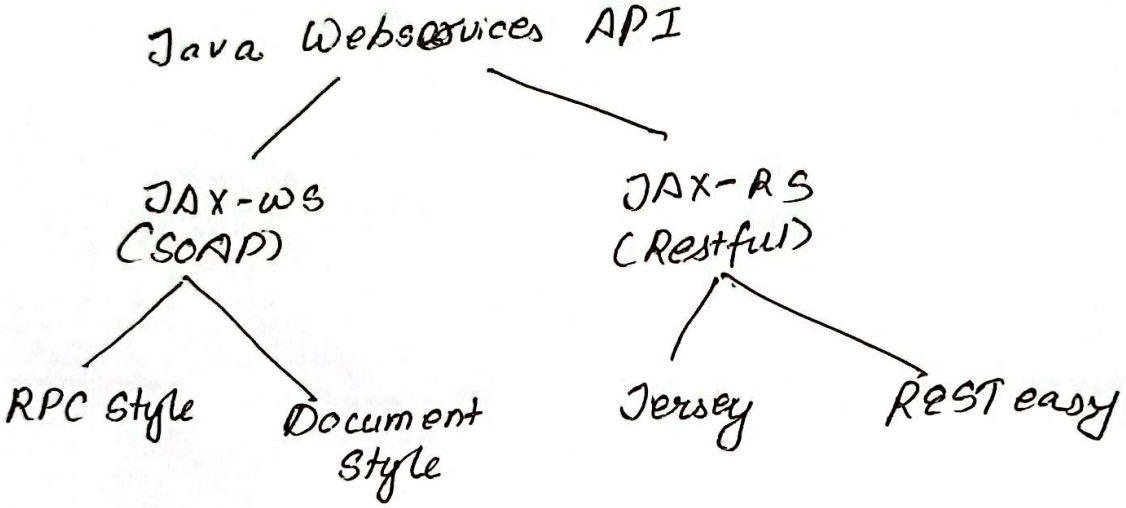
```
class MVCPattern {
    public static void main (String[] args)
{
    Student model = retriveStudentfromDatabase();
    StudentView view = new StudentView();
    StudentController controller = new StudentController
                                    (model.view);
        controller.updateView();
        controller.setStudentName("VS");
        controller.updateView();
    }
private static Student retrivesStudent From Database()
        {
        Student student = new Student();
        Student.setName("Lokesh S");
        Student.setRollNo("1__ __");
        return student;
    }
}
```

Lab 7 : To implement about Web Services.

THEORY:

Web Services are set of protocols and standards used to exchange data and functionality over the internet. It allows different s/w app. to communicate and interact with each other regardless of programming languages or platforms they are built upon.

There are 2 main API's defined by Java for developing web service application. They are:



① JAX-WS : for SOAP web services. There are 2 ways to write JAX-WS application code : by RPC-Style and Document Style.

② JAX-RS : for RESTful web services. There are mainly 2 implementation currently in use for creating JAX-RS application : Jersey and RESTeasy.

Conclusion :
Through this lab. we gain valuable knowledge in implementing Web Services with OOSD.