Experiment#		Student ID	
Date		Student Name	[@KLWKS_BOT]

1. Creational Design Patterns

Aim/Objective: To analyse the implementation of Singleton, Factory and Abstract Design Patterns for the real time scenarios.

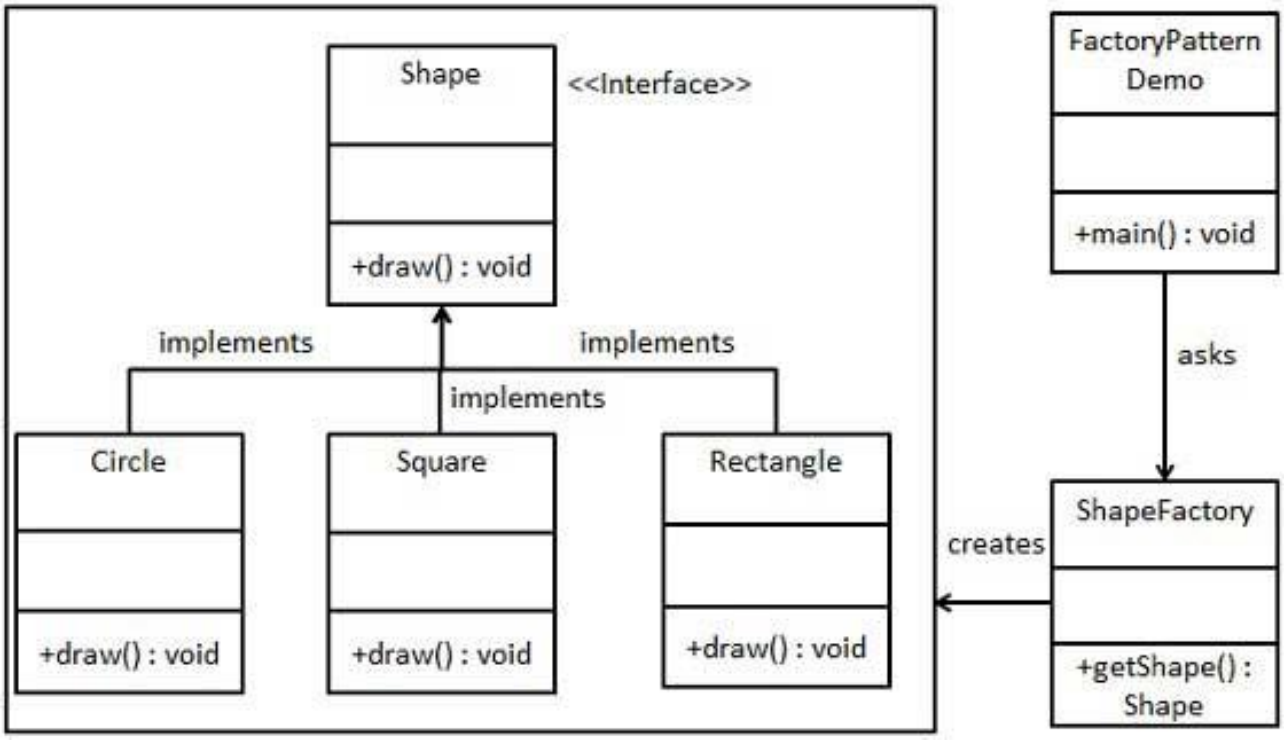
Description: The student will understand the concept of Creational Design Patterns (Singleton, Factory and Abstract Factory)

Pre-Requisites: Classes and Objects in Java

Tools: Eclipse IDE for Enterprise Java and Web Developers

Pre-Lab:

1) Draw the UML Relationship Diagram for Factory Design Pattern for any customized scenario.



In-Lab:

1) Develop a game application with multiple levels and varying difficulty settings. Implement the following design patterns to manage different aspects of the game:

Singleton Pattern: Use this pattern to manage the game state, ensuring that there is only one instance of the game state throughout the application.

Factory Method Pattern: Apply this pattern to create different types of enemies for each level.

Abstract Factory Pattern: Utilize this pattern to create various types of weapons and power-ups based on the level and difficulty settings.

Ensure that the design of your game is flexible and can easily accommodate new levels, enemies, weapons, and power-ups in the future.

Procedure/Program:

```
public class Game {

  static class GameState {
    private static GameState instance = new GameState();
    private int score = 0, level = 1;

    private GameState() {}

    public static GameState getInstance() {
      return instance;
    }

    public int getScore() {
      return score;
    }

    public void increaseScore(int points) {
      score += points;
    }

    public int getLevel() {
      return level;
    }
```

```
    public void setLevel(int level) {
        this.level = level;
    }
}

abstract static class Enemy {
    public abstract void attack();
}

static class EasyEnemy extends Enemy {
    @Override
    public void attack() {
        System.out.println("Easy Enemy attacks!");
    }
}

static class MediumEnemy extends Enemy {
    @Override
    public void attack() {
        System.out.println("Medium Enemy attacks!");
    }
}

static class HardEnemy extends Enemy {
    @Override
    public void attack() {
        System.out.println("Hard Enemy attacks!");
    }
}

static class EnemyFactory {
    public static Enemy createEnemy(int level) {
        if (level == 1) return new EasyEnemy();
        else if (level == 2) return new MediumEnemy();
        return new HardEnemy();
    }
}

interface GameComponentFactory {
    Weapon createWeapon();
    PowerUp createPowerUp();
```

```
    }

    static class EasyGameComponentFactory implements
GameComponentFactory {
        public Weapon createWeapon() {
            return new EasyWeapon();
        }

        public PowerUp createPowerUp() {
            return new EasyPowerUp();
        }
    }

    static class MediumGameComponentFactory implements
GameComponentFactory {
        public Weapon createWeapon() {
            return new MediumWeapon();
        }

        public PowerUp createPowerUp() {
            return new MediumPowerUp();
        }
    }

    static class HardGameComponentFactory implements
GameComponentFactory {
        public Weapon createWeapon() {
            return new HardWeapon();
        }

        public PowerUp createPowerUp() {
            return new HardPowerUp();
        }
    }

    abstract static class Weapon {
        public abstract void attack();
    }

    static class EasyWeapon extends Weapon {
        @Override
```

```
    public void attack() {
        System.out.println("Easy Weapon Attack!");
    }
}

static class MediumWeapon extends Weapon {
    @Override
    public void attack() {
        System.out.println("Medium Weapon Attack!");
    }
}

static class HardWeapon extends Weapon {
    @Override
    public void attack() {
        System.out.println("Hard Weapon Attack!");
    }
}

abstract static class PowerUp {
    public abstract void activate();
}

static class EasyPowerUp extends PowerUp {
    @Override
    public void activate() {
        System.out.println("Easy Power-Up Activated!");
    }
}

static class MediumPowerUp extends PowerUp {
    @Override
    public void activate() {
        System.out.println("Medium Power-Up Activated!");
    }
}

static class HardPowerUp extends PowerUp {
    @Override
    public void activate() {
        System.out.println("Hard Power-Up Activated!");
```

```
    }
  }

  public void play(String difficulty) {
    GameState gameState = GameState.getInstance();
    GameComponentFactory factory;

    if (difficulty.equalsIgnoreCase("easy")) {
      factory = new EasyGameComponentFactory();
    } else if (difficulty.equalsIgnoreCase("medium")) {
      factory = new MediumGameComponentFactory();
    } else {
      factory = new HardGameComponentFactory();
    }

    for (int i = 1; i <= 3; i++) {
      gameState.setLevel(i);

      Enemy enemy = EnemyFactory.createEnemy(gameState.getLevel());
      Weapon weapon = factory.createWeapon();
      PowerUp powerUp = factory.createPowerUp();

      System.out.println("\nLevel: " + gameState.getLevel() + " | Score: " +
gameState.getScore());
      enemy.attack();
      weapon.attack();
      powerUp.activate();

      gameState.increaseScore(100);
    }

    System.out.println("\nGame Over! Final Score: " + gameState.getScore());
  }

  public static void main(String[] args) {
    new Game().play("easy");
  }
}
```

OUTPUT

```
Level: 1 | Score: 0
Easy Enemy attacks!
Easy Weapon Attack!
Easy Power-Up Activated!

Level: 2 | Score: 100
Medium Enemy attacks!
Easy Weapon Attack!
Easy Power-Up Activated!

Level: 3 | Score: 200
Hard Enemy attacks!
Easy Weapon Attack!
Easy Power-Up Activated!

Game Over! Final Score: 300
```

✓ **Data and Results:**

Data

The game features three levels with increasing difficulty progression.

Result

Each level outputs attacks and power-ups, totaling 300 score.

✓ **Analysis and Inferences:**

Analysis

The code employs patterns ensuring modular, scalable game design.

Inferences

Game mechanics adapt easily to future levels and difficulties.

VIVA-VOCE Questions (In-Lab):

1) Which classes are candidates of Singleton? Which kind of class do you make Singleton in Java?

- Classes that need only one instance (e.g., Logger, ConfigurationManager).
- **How to create Singleton:**
  - Private constructor.
  - Static instance.
  - Public `getInstance()` method to return the single instance.

2) Discuss the difference between Factory and Abstract Factory design patterns.

- **Factory:** Creates a single product type (e.g., `CarFactory` creates `Sedan` or `SUV`).
- **Abstract Factory:** Creates families of related products (e.g., `CarFactory` creates `Sedan` and `SUV` for multiple brands).

3) Mention the pros and cons of Factory Design pattern

- **Pros:**
  - Encapsulates object creation.
  - Easy to extend.
  - Flexible and maintainable.
- **Cons:**
  - Can become complex.
  - Increases the number of classes.

4) Mention the cons of Singleton Design pattern.

- Global state makes testing harder.
- Difficult to maintain in multi-threaded environments.
- Hard to extend or modify the pattern.

Post-Lab:

1) Design and implement a ride-sharing application that allows users to request rides from various types of vehicles (cars, bikes, scooters). Utilize the Factory Method pattern to create vehicle instances, the Abstract Factory pattern to implement different payment methods (credit card, PayPal, cash), and the Singleton pattern to manage user authentication securely. Provide a detailed example demonstrating the interaction of these patterns within the application.

Procedure/Program:

```
import java.util.Scanner;

public class RideSharingApp {

   interface Vehicle {
      String ride();
      double getPrice();
   }

   static class Car implements Vehicle {
      public String ride() {
         return "Riding a car.";
      }

      public double getPrice() {
         return 50.0;
      }
   }

   static class Bike implements Vehicle {
      public String ride() {
         return "Riding a bike.";
      }

      public double getPrice() {
         return 35.0;
      }
```

```
}

static class Scooter implements Vehicle {
    public String ride() {
        return "Riding a scooter.";
    }

    public double getPrice() {
        return 25.0;
    }
}

interface PaymentMethod {
    String processPayment(double amount);
}

static class CreditCardPayment implements PaymentMethod {
    public String processPayment(double amount) {
        return "Paid " + amount + " using Credit Card.";
    }
}

static class PayPalPayment implements PaymentMethod {
    public String processPayment(double amount) {
        return "Paid " + amount + " using PayPal.";
    }
}

static class CashPayment implements PaymentMethod {
    public String processPayment(double amount) {
        return "Paid " + amount + " using Cash.";
    }
}

static class UserAuthentication {
    private static UserAuthentication instance;
    private String user;

    private UserAuthentication() {}
```

```
    public static UserAuthentication getInstance() {
        if (instance == null) {
            instance = new UserAuthentication();
        }
        return instance;
    }

    public boolean login(String username, String password) {
        this.user = username;
        return true;
    }

    public void logout() {
        user = null;
    }

    public String getUser() {
        return user;
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    UserAuthentication auth = UserAuthentication.getInstance();
    String username, password;
    boolean loggedIn = false;

    while (!loggedIn) {
        System.out.print("Enter username: ");
        username = scanner.nextLine();
        System.out.print("Enter password: ");
        password = scanner.nextLine();

        if (auth.login(username, password)) {
            loggedIn = true;
            System.out.println("Welcome, " + auth.getUser() + "!");
        } else {
            System.out.println("Invalid credentials! Try again.");
        }
```

```
    }

    String vehicleType;
    Vehicle vehicle = null;

    while (vehicle == null) {
       System.out.print("Enter vehicle type (car/bike/scooter): ");
       vehicleType = scanner.nextLine().toLowerCase();
       switch (vehicleType) {
          case "car":
             vehicle = new Car();
             break;
          case "bike":
             vehicle = new Bike();
             break;
          case "scooter":
             vehicle = new Scooter();
             break;
          default:
             System.out.println("Invalid vehicle type! Please try again.");
       }
    }
    System.out.println(vehicle.ride());

    double vehiclePrice = vehicle.getPrice();
    System.out.print("Enter payment method (credit/paypal/cash): ");
    String paymentType = scanner.nextLine().toLowerCase();
    PaymentMethod paymentMethod = null;

    while (paymentMethod == null) {
       switch (paymentType) {
          case "credit":
             paymentMethod = new CreditCardPayment();
             break;
          case "paypal":
             paymentMethod = new PayPalPayment();
             break;
          case "cash":
             paymentMethod = new CashPayment();
             break;
          default:
```

```
                System.out.println("Invalid payment method! Please try again.");


            System.out.print("Enter payment method (credit/paypal/cash): ");
                paymentType = scanner.nextLine().toLowerCase();
            }
        }

        System.out.println(paymentMethod.processPayment(vehiclePrice));

        auth.logout();
        System.out.println("Logged out successfully.");
        scanner.close();
    }
}
```

OUTPUT

```
Enter username: Thanos
Enter password: 123
Welcome, Thanos!
Enter vehicle type (car/bike/scooter): car
Riding a car.
Enter payment method (credit/paypal/cash): cash
Paid 50.0 using Cash.
Logged out successfully.
```

- ✓ **Data and Results:**

Data:

The data consists of user input, vehicle choice, and payment method.

Results:

The results include the ride type chosen and the processed payment.

- ✓ **Analysis and Inferences:**

Analysis:

We analyze the vehicle types, prices, and payment method choices.

Inferences:

We infer user preferences for vehicle and payment method selections.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.