**Flyweight Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

* **Flyweight Design Pattern :**

The Flyweight Design Pattern is a structural design pattern that focuses on optimizing memory usage by sharing common parts of object state among multiple objects, instead of each object storing its own copy. This pattern is particularly useful when dealing with a large number of similar objects, which would otherwise consume a significant amount of memory if each object maintained its own data.

The key idea behind the Flyweight pattern is to separate intrinsic state (shared state) and extrinsic state (unique state) of an object. The intrinsic state is shared among multiple objects and can be stored externally, while the extrinsic state varies from object to object and must be provided externally when needed.

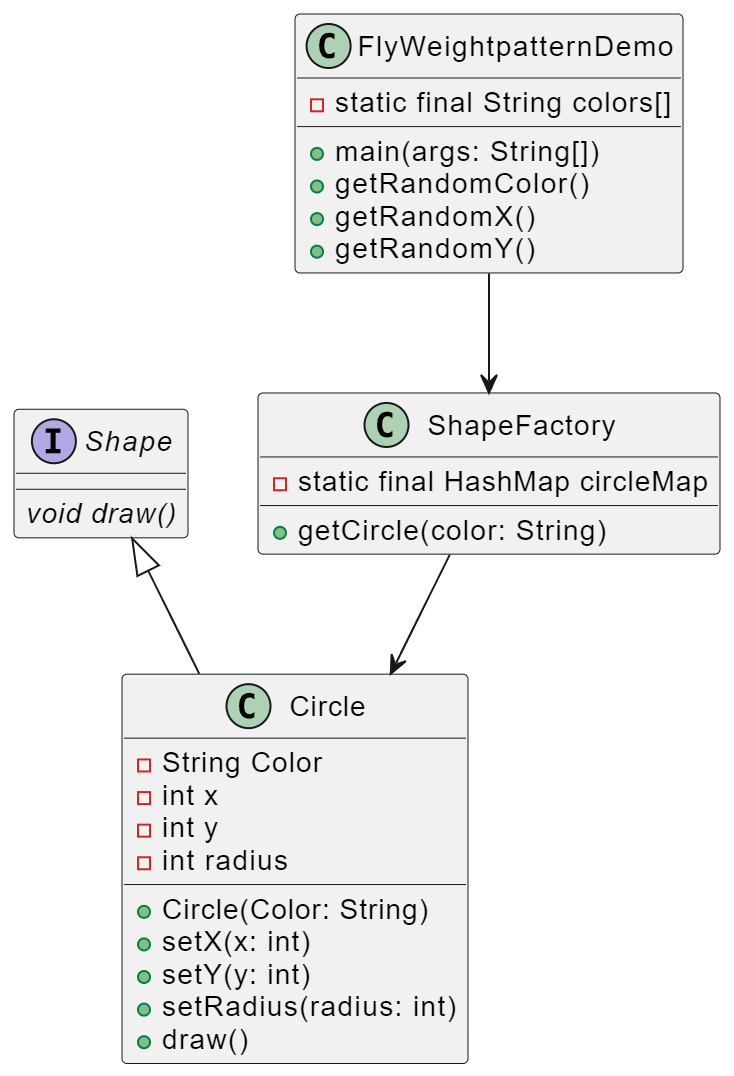
By minimizing memory usage through sharing common state, the Flyweight pattern helps improve performance and reduces memory footprint, especially in scenarios where a large number of similar objects need to be created.

Key components of the Flyweight pattern include:

* Flyweight Interface/Abstract Class: This defines the interface for flyweight objects. It typically includes methods to set and get intrinsic state.
* Concrete Flyweight: This class implements the Flyweight interface and stores the intrinsic state that can be shared among multiple objects.
* Flyweight Factory: This is responsible for managing flyweight objects. It typically maintains a pool of existing flyweight objects and provides a way to retrieve a flyweight object based on certain criteria.
* Client: This is the code that uses flyweight objects. It may need to provide extrinsic state to flyweight objects when interacting with them.

By using the Flyweight pattern, developers can improve the performance and efficiency of their applications, especially in scenarios where large numbers of objects with shared state are involved. However, it's important to carefully consider the trade-offs, as introducing shared state can potentially lead to increased complexity and reduced encapsulation.

* **Program :** Implement flyweight design pattern for Shape example.
* **UML Diagram :**



* **Code :**

public interface Shape

{

void draw();

}

public class Circle implements Shape

{

private String color;

private int x;

private int y;

private int radius;

public Circle(String color)

{

this.color = color;

}

public void setX(int x)

{

this.x = x;

}

public void setY(int y)

{

this.y = y;

}

public void setRadius(int radius)

{

this.radius = radius;

}

@Override

public void draw()

{

System.out.println("Circle: Draw() [Color : "+ color +", x : "+x+", y : "+y+"]");

}

}

import java.util.HashMap;

public class ShapeFactory

{

private static final HashMap circleMap = new HashMap();

public static Shape getCircle(String color)

{

Circle circle = (Circle)circleMap.get(color);

if(circle == null)

{

circle = new Circle(color);

circleMap.put(color, circle);

System.out.println("Creating circle of color : "+color);

}

return circle;

}

}

public class FlyweightPatternDemo

{

private static final String colors[] = { "Red", "Green", "Blue", "White", "Black"};

public static void main(String[] args)

{

for (int i=0; i<20; ++i)

{

Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());

circle.setX(getRandomX());

circle.setY(getRandomY());

circle.setRadius(100);

circle.draw();

}

}

private static String getRandomColor()

{

return colors[(int)(Math.random()\*colors.length)];

}

private static int getRandomX()

{

return (int)(Math.random()\*100);

}

private static int getRandomY()

{

return (int)(Math.random()\*100);

}

}

* **Output :**

