**Class Diagram**

**Roles of Each Class**

* **Shape**: Abstract class that defines common properties and methods for all shapes.
* **Square & Circle &** **Rectangle**: Concrete classes extending Shape, implementing specific behavior for drawing and getting descriptions.
* **WindowsSquare & LinuxSquare**: Specific implementations of Square for Windows and Linux OS respectively.
* **WindowsCircle & LinuxCircle**: Specific implementations of Circle for Windows and Linux OS respectively.
* **WindowsRectangle & LinuxRectangle:** Specific implementations of Rectangle for Windows and Linux OS respectively.
* **ShapeFactory**: Interface for creating shapes.
* **WindowsShapeFactory & LinuxShapeFactory**: Concrete factories implementing the ShapeFactory interface to create Windows and Linux shapes.
* **ShapeDecorator**: Abstract class for decorators that extends Shape.
* **ColorDecorator, BorderColorDecorator, BorderSizeDecorator**: Concrete decorators that add functionality to the shapes.
* **Command**: Interface for command pattern.
* **DrawCommand**: Concrete command that executes drawing a shape.
* **ShapeFactorySingleton**: Singleton class ensuring a single instance of the shape factory.

**Code Implementation**

Here is the complete code for the implementation.

**Shape (Abstract Class)**

public abstract class **Shape** {

protected String color;

protected String borderColor;

protected double borderSize;

public void setColor(String color) {

this.color = color;

}

public void setBorderColor(String borderColor) {

this.borderColor = borderColor;

}

public void setBorderSize(double borderSize) {

this.borderSize = borderSize;

}

public abstract void draw();

public abstract String getDescription();

}

public class **Square** extends **Shape** {

@Override

public void draw() {

System.out.println("Drawing a square");

}

@Override

public String getDescription() {

return "Square";

}

}

public class **Rectangle** extends **Shape** {

@Override

public void draw() {

System.out.println("Drawing a rectangle");

}

@Override

public String getDescription() {

return "Rectangle";

}

}

public class **Circle** extends **Shape** {

@Override

public void draw() {

System.out.println("Drawing a circle");

}

@Override

public String getDescription() {

return "Circle";

}

}

public class **WindowsSquare** extends **Square** {

@Override

public void draw() {

super.draw();

System.out.println("Executing border light command for square.");

}

@Override

public String getDescription() {

return "Windows Square";

}

}

public class **LinuxSquare** extends **Square** {

@Override

public void draw() {

super.draw();

System.out.println("Executing border light command for square.");

}

@Override

public String getDescription() {

return "Linux Square";

}

}

public class **WindowsRectangle** extends **Rectangle** {

@Override

public void draw() {

super.draw();

System.out.println("Executing border light command for rectangle.");

}

@Override

public String getDescription() {

return "Windows Rectangle";

}

}

public class **LinuxRectangle** extends **Rectangle** {

@Override

public void draw() {

super.draw();

System.out.println("Executing border light command for rectangle.");

}

@Override

public String getDescription() {

return "Linux Rectangle";

}

}

public class **WindowsCircle** extends **Circle** {

@Override

public void draw() {

super.draw();

System.out.println("Executing motion command for circle.");

}

@Override

public String getDescription() {

return "Windows Circle";

}

}

public class **LinuxCircle** extends **Circle** {

@Override

public void draw() {

super.draw();

System.out.println("Executing motion command for circle.");

}

@Override

public String getDescription() {

return "Linux Circle";

}

}

**ShapeFactory (Abstract Factory)**

public interface **ShapeFactory** {

Shape createSquare();

Shape createRectangle();

Shape createCircle();

}

public class **WindowsShapeFactory** implements **ShapeFactory** {

@Override

public Shape createSquare() {

return new WindowsSquare();

}

@Override

public Shape createRectangle() {

return new WindowsRectangle();

}

@Override

public Shape createCircle() {

return new WindowsCircle();

}

}

public class **LinuxShapeFactory** implements **ShapeFactory** {

@Override

public Shape createSquare() {

return new LinuxSquare();

}

@Override

public Shape createRectangle() {

return new LinuxRectangle();

}

@Override

public Shape createCircle() {

return new LinuxCircle();

}

}

**ShapeFactorySingleton (Singleton)**

public class **ShapeFactorySingleton** {

private static ShapeFactorySingleton instance;

private Shape square;

private Shape rectangle;

private Shape circle;

private ShapeFactorySingleton() {}

public static ShapeFactorySingleton getInstance() {

if (instance == null) {

instance = new ShapeFactorySingleton();

}

return instance;

}

public ShapeFactory getFactory(String osType) {

if (osType.equalsIgnoreCase("Windows")) {

return new WindowsShapeFactory();

} else if (osType.equalsIgnoreCase("Linux")) {

return new LinuxShapeFactory();

}

return null;

}

public Shape createShape(String osType, String shapeType) {

ShapeFactory factory = getFactory(osType);

if (factory == null) return null;

if (shapeType.equalsIgnoreCase("Square")) {

return factory.createSquare();

} else if (shapeType.equalsIgnoreCase("Rectangle")) {

return factory.createRectangle();

} else if (shapeType.equalsIgnoreCase("Circle")) {

return factory.createCircle();

}

return null;

}

}

**ShapeDecorator (Decorator)**

public abstract class **ShapeDecorator** extends **Shape** {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape) {

this.decoratedShape = decoratedShape;

}

@Override

public void draw() {

decoratedShape.draw();

}

@Override

public String getDescription() {

return decoratedShape.getDescription();

}

}

public class **ColorDecorator** extends **ShapeDecorator** {

private String color;

public ColorDecorator(Shape decoratedShape, String color) {

super(decoratedShape);

this.color = color;

}

@Override

public void draw() {

super.draw();

System.out.println("Filling shape with color: " + color);

}

@Override

public String getDescription() {

return super.getDescription() + ", filled with color: " + color;

}

}

public class **BorderColorDecorator** extends **ShapeDecorator** {

private String borderColor;

public BorderColorDecorator(Shape decoratedShape, String borderColor) {

super(decoratedShape);

this.borderColor = borderColor;

}

@Override

public void draw() {

super.draw();

System.out.println("Setting border color: " + borderColor);

}

@Override

public String getDescription() {

return super.getDescription() + ", border color: " + borderColor;

}

}

public class **BorderSizeDecorator** extends **ShapeDecorator** {

private double borderSize;

public BorderSizeDecorator(Shape decoratedShape, double borderSize) {

super(decoratedShape);

this.borderSize = borderSize;

}

@Override

public void draw() {

super.draw();

System.out.println("Setting border size: " + borderSize);

}

@Override

public String getDescription() {

return super.getDescription() + ", border size: " + borderSize;

}

}

**Command**

public interface **Command** {

void execute();

}

public class DrawCommand implements Command {

private Shape shape;

public DrawCommand(Shape shape) {

this.shape = shape;

}

@Override

public void execute() {

shape.draw();

}

}

public class **ShapeTest** {

public static void main(String[] args) {

ShapeFactorySingleton factorySingleton = ShapeFactorySingleton.getInstance();

Shape windowsSquare = factorySingleton.createShape("Windows", "Square");

Shape linuxCircle = factorySingleton.createShape("Linux", "Circle");

Shape decoratedWindowsSquare = new BorderSizeDecorator(new ColorDecorator(windowsSquare, "Red"), 2.0);

Shape decoratedLinuxCircle = new BorderColorDecorator(linuxCircle, "Blue");

Command drawWindowsSquare = new DrawCommand(decoratedWindowsSquare);

Command drawLinuxCircle = new DrawCommand(decoratedLinuxCircle);

drawWindowsSquare.execute();

drawLinuxCircle.execute(); }

}

Factory Pattern: يسهل إنشاء كائنات الأشكال دون الحاجة لمعرفة تفاصيل التنفيذ، ويدعم إنشاء الأشكال لأنظمة التشغيل المختلفة.

Singleton Pattern: يضمن وجود نسخة واحدة فقط من المصنع، مما يعزز إدارة الموارد والتأكد من استخدام المصنع بطريقة موحدة.

Decorator Pattern: يتيح إضافة خصائص جديدة للأشكال دون تغيير الكود الأساسي للأشكال، مما يتيح مزيدًا من المرونة في تخصيص الأشكال.

Command Pattern: يعزل طلبات تنفيذ الأوامر عن الكائنات التي تقوم بتنفيذها، مما يسهل إدارة وتنفيذ الأوامر بشكل مرن وموحد.

Project Overview

The project involves creating different shapes (Squares, Rectangles, and Circles) specific to two operating systems: Windows and Linux. Core functionalities include drawing shapes and providing descriptions, with the ability to dynamically extend these shapes' properties using decorators. Additionally, the Command pattern is used to encapsulate drawing commands for shapes.

Design Patterns Used

Abstract Factory Pattern:

Purpose: This pattern is used to create families of related objects (shapes specific to different operating systems) without specifying their concrete classes.

Implementation:

ShapeFactory Interface: Defines methods for creating shapes (createSquare(), createRectangle(), createCircle()).

WindowsShapeFactory and LinuxShapeFactory Classes: Concrete implementations of the ShapeFactory interface that produce Windows-specific and Linux-specific shapes, respectively.

Decorator Pattern:

Purpose: This pattern allows adding new functionality to existing objects without modifying their structure.

Implementation:

ShapeDecorator Abstract Class: Serves as the base class for all decorators and holds a reference to a Shape object.

Concrete Decorators (ColorDecorator, BorderColorDecorator, BorderSizeDecorator): Extend ShapeDecorator to add new properties like color, border color, and border size. They override the draw() and getDescription() methods to include these enhancements.

Singleton Pattern:

Purpose: Ensures that only one instance of the ShapeFactorySingleton exists, providing a global point of access to the factory.

Implementation:

ShapeFactorySingleton Class: Ensures only one instance and provides the method getFactory() to retrieve the appropriate shape factory based on the operating system type.

Command Pattern:

Purpose: Encapsulates a request as an object, allowing parameterization of clients with queues, requests, and operations.

Implementation:

Command Interface: Defines the execute() method that all command classes must implement.

DrawCommand Class: Implements the Command interface and holds a reference to a Shape, calling the draw() method when execute() is invoked.

Class Structure and Relationships

Shape Hierarchy:

Shape (Abstract Class): The base class for all shapes, defining common properties (color, borderColor, borderSize) and methods (draw(), getDescription()).

Concrete Classes (Square, Rectangle, Circle): Extend Shape and provide specific implementations for draw() and getDescription().

OS-Specific Shapes (WindowsSquare, LinuxSquare, WindowsRectangle, LinuxRectangle, WindowsCircle, LinuxCircle): Extend the base shape classes and provide OS-specific implementations.

Shape Factory:

ShapeFactory Interface: Defines methods to create shapes.

WindowsShapeFactory and LinuxShapeFactory Classes: Implement the interface to produce OS-specific shapes.

Decorators:

ShapeDecorator (Abstract Class): Extends Shape and holds a reference to a Shape object.

Concrete Decorators (ColorDecorator, BorderColorDecorator, BorderSizeDecorator): Extend ShapeDecorator to add and describe new properties.

Singleton Factory:

ShapeFactorySingleton Class: Ensures a single instance and provides access to shape factories.

Command Pattern:

Command Interface: Defines the execute() method.

DrawCommand Class: Implements Command and performs the drawing operation.

Class Diagram Explanation

The class diagram represents the relationships between classes:

Inheritance:

Shape is the base class for all shapes (Square, Rectangle, Circle) and is extended by specific implementations (WindowsSquare, LinuxSquare, etc.).

ShapeDecorator is the base class for all decorators that enhance the base Shape functionality.

Interfaces:

ShapeFactory and Command are interfaces, indicating behaviors that concrete classes must implement.

Associations:

DrawCommand has an association with Shape, indicating that it utilizes a shape to perform its operation.

ShapeFactorySingleton provides access to shape factories that create specific shape instances.

Example Execution Flow

Creating Shapes:

The user calls ShapeFactorySingleton.getInstance() to get the singleton instance.

Based on the operating system (Windows or Linux), the appropriate factory is retrieved.

The user creates a shape (e.g., a Windows square) and applies decorations (e.g., color and border size).

Using Command Pattern:

A DrawCommand is created for the shape (e.g., the decorated Windows square).

Calling execute() on the command triggers the drawing of the shape, including all applied decorations.

Output:

The output includes drawing messages and descriptions that reflect the current configuration of the shape (e.g., color, border size).