# Lesson 9: Design Patterns in Java

Marcin Kurzyna

## Lecture Goals

By the end of this lesson, students should be able to:

- Understand what a **design pattern** is.
- Identify main categories of design patterns.
- Implement selected **creational** and **structural** patterns.
- Use design patterns to improve code extensibility.

## 1 What Are Design Patterns?

A **design pattern** is a reusable solution to a commonly occurring problem in software design.

Design patterns:

- are independent of programming language,
- represent best practices,
- help structure object-oriented programs.

## 2 Categories of Design Patterns

Design patterns are divided into three main groups:

- **Creational** – object creation mechanisms
- **Structural** – object composition
- **Behavioral** – object interaction

## 3 Creational Pattern: Singleton

The **Singleton** pattern ensures that a class has:

- only one instance,
- a global access point to it.

### Example: Singleton Configuration

```
class Configuration {
    private static Configuration instance;

    private Configuration() {
    }

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

    public String getAppName() {
        return "My Application";
    }
}
```

## 4 Creational Pattern: Builder

The **Builder** pattern:

- separates object construction from its representation,
- is useful when constructors would have many parameters.

### Example: Builder for SVG Rectangle

```
class SvgRectangle {
    private double x, y, width, height;
    private String fill;
    private String stroke;
    private double strokeWidth;

    private SvgRectangle(Builder builder) {
        this.x = builder.x;
        this.y = builder.y;
        this.width = builder.width;
        this.height = builder.height;
        this.fill = builder.fill;
        this.stroke = builder.stroke;
        this.strokeWidth = builder.strokeWidth;
    }

    public static class Builder {
        private double x, y, width, height;
        private String fill = "none";
        private String stroke = "black";
```

```
        private double strokeWidth = 1;

        public Builder position(double x, double y) {
            this.x = x;
            this.y = y;
            return this;
        }

        public Builder size(double width, double height) {
            this.width = width;
            this.height = height;
            return this;
        }

        public Builder fill(String fill) {
            this.fill = fill;
            return this;
        }

        public Builder stroke(String stroke, double strokeWidth)
           {
            this.stroke = stroke;
            this.strokeWidth = strokeWidth;
            return this;
        }

        public SvgRectangle build() {
            return new SvgRectangle(this);
        }
    }

    public String toSvg() {
        return "<rect x=\"" + x + "\" y=\"" + y +
               "\" width=\"" + width + "\" height=\"" + height +
               "\" fill=\"" + fill +
               "\" stroke=\"" + stroke +
               "\" stroke-width=\"" + strokeWidth + "\" />";
    }
}
```

Usage:

```
SvgRectangle rect = new SvgRectangle.Builder()
        .position(10, 10)
        .size(100, 50)
        .fill("red")
        .stroke("black", 2)
        .build();
```

# 5 Structural Pattern: Decorator

The **Decorator** pattern:

- allows behavior to be added dynamically,
- avoids creating many subclasses.

### Example: Shape Decorator

```
interface Shape {
    String toSvg();
}
```

```
class BasicCircle implements Shape {
    public String toSvg() {
        return "<circle cx=\"50\" cy=\"50\" r=\"40\" />";
    }
}
```

```
abstract class ShapeDecorator implements Shape {
    protected Shape shape;

    ShapeDecorator(Shape shape) {
        this.shape = shape;
    }
}
```

```
class ColoredShape extends ShapeDecorator {
    private String color;

    ColoredShape(Shape shape, String color) {
        super(shape);
        this.color = color;
    }

    public String toSvg() {
        return shape.toSvg()
                .replace("/>", " fill=\"" + color + "\" />");
    }
}
```

Usage:

```
Shape circle = new ColoredShape(
                    new BasicCircle(),
                    "blue");
System.out.println(circle.toSvg());
```

## Summary

In this lesson you learned:

- What design patterns are and how they help in OOP.
- How to use the **Singleton** pattern.

- How the **Builder** pattern simplifies object creation.
- How the **Decorator** pattern extends functionality dynamically.

# 6 Exercises

1. Make the `Singleton` thread-safe.
2. Extend the `Builder` to support SVG circles.
3. Create a `BorderDecorator` that adds stroke to a shape.
4. Combine Builder and Decorator in one example.
5. (Challenge) Implement a text-based menu using the Builder pattern.