COMP 3550

7.1 — WHAT ARE DESIGN PATTERNS & WHEN TO USE THEM

Week 7: Design Patterns & Architecture

DESIGN PATTERNS ≠ CODE RECIPES

It’s about design, not copy-paste coding.

What They Are:

- Proven solutions to recurring design problems
- Described in terms of context, problem, and solution
- Language/framework agnostic

What They Aren’t:

- Copy-paste snippets
- “Cool tricks” to show off
- Mandatory for every project

Origin:

- Popularized by the Gang of Four (GoF) in Design Patterns: Elements of Reusable Object-Oriented Software

PATTERN CLASSIFICATIONS

Creational

Focus: Managing object creation

Goal: Control when and how objects are made

PATTERN CLASSIFICATIONS

Creational

Focus: Managing object creation

Goal: Control when and how objects are made

Structural

Focus: Simplifying relationships between objects

Goal: Organize and connect components cleanly

PATTERN CLASSIFICATIONS

Creational

Focus: Managing object creation

Goal: Control when and how objects are made

Structural

Focus: Simplifying relationships between objects

Goal: Organize and connect components cleanly

Behavioral

Focus: Organizing communication between objects

Goal: Define clear interaction patterns

WHEN TO USE PATTERNS

Patterns are tools — use them when they fit.

You keep writing the same structure or logic in multiple places.

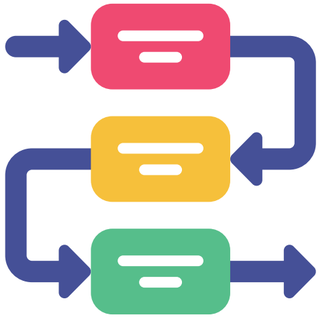
(DRY, OCP)

WHEN TO USE PATTERNS

Patterns are tools — use them when they fit.

You keep writing the same structure or logic in multiple places.

(DRY, OCP)



You want to reduce dependencies and make changes without breaking everything.

(Coupling, SRP, PoLK)

WHEN TO USE PATTERNS

Patterns are tools — use them when they fit.

You keep writing the same structure or logic in multiple places.

(DRY, OCP)

You want to reduce dependencies and make changes without breaking everything.

(Coupling, SRP, PoLK)

You want clearer communication between parts of the system.

(PoLK, PoLA, DIP, Coupling, Cohesion)

WHEN TO USE PATTERNS

Patterns are tools — use them when they fit.

You keep writing the same structure or logic in multiple places.

(DRY, OCP)

You want to reduce dependencies and make changes without breaking everything.

(Coupling, SRP, PoLK)

You want clearer communication between parts of the system.

(PoLK, PoLA, DIP, Coupling, Cohesion)

You want to add new features without rewriting large sections.

(Coupling, Cohesion, OCP, LSP, ISP)

WHEN TO USE PATTERNS

Patterns are tools — use them when they fit.

You keep writing the same structure or logic in multiple places.

(DRY, OCP)

You want to reduce dependencies and make changes without breaking everything.

(Coupling, SRP, PoLK)

You want clearer communication between parts of the system.

(PoLK, PoLA, DIP, Coupling, Cohesion)

You want to add new features without rewriting large sections.

(Coupling, Cohesion, OCP, LSP, ISP)

Some of the most common indicators (not all the indicators though)

PATTERN MISUSE WARNING

Don’t turn your code into pattern soup

PATTERN MISUSE WARNING

Don’t turn your code into pattern soup

Don’t build a rocket ship if all you need is a bicycle
or
KISS (Keep it simple, stupid)

PATTERN MISUSE WARNING

The Danger:

- Overusing patterns for problems that don’t need them
- Mixing too many patterns in one place makes code hard to follow
- “Over-architecting” = adding complexity without real benefit

PATTERN MISUSE WARNING

The Danger:

- Overusing patterns for problems that don’t need them
- Mixing too many patterns in one place makes code hard to follow
- “Over-architecting” = adding complexity without real benefit

Why It’s a Problem:

- Slows development
- Confuses maintainers (including future you)
- Makes debugging harder

PATTERN MISUSE WARNING

The Danger:

- Overusing patterns for problems that don’t need them
- Mixing too many patterns in one place makes code hard to follow
- “Over-architecting” = adding complexity without real benefit

Why It’s a Problem:

- Slows development
- Confuses maintainers (including future you)
- Makes debugging harder

Better Approach:

- Start simple — refactor to a pattern when you see a recurring need
- Match the pattern to the actual problem, not the other way around

PATTERN IN ACTION: REPLACING A SWITCH STATEMENT

Can keep growing, gets messy/hard to follow, big ‘ol OCP violation

```
public double calculate(int shapeType, double value) {
    switch (shapeType) {
        case 1: // square
            return value * value;
        case 2: // circle
            return Math.PI * value * value;
        case 3: // equilateral triangle
            return (Math.sqrt(3) / 4) * value * value;
        case 4: // hexagon
            return (3 * Math.sqrt(3) / 2) * value * value;
        default:
            throw new IllegalArgumentException("Unknown shape type: " + shapeType);
    }
}
```

PATTERN IN ACTION: REPLACING A SWITCH STATEMENT

Can keep growing, gets messy/hard to follow, big ‘ol OCP violation

```
interface Shape {
    double calculate(double value);
}

class Square implements Shape {
    public double calculate(double value) { return value * value; }
}

class Circle implements Shape {
    public double calculate(double value) { return Math.PI * value * value; }
}

class ShapeCalculator {
    private Shape shape;
    ShapeCalculator(Shape shape) { this.shape = shape; }
    public double calculate(double value) { return shape.calculate(value); }
}
```

PATTERN IN ACTION: REPLACING A SWITCH STATEMENT

Can keep growing, gets messy/hard to follow, big ‘ol OCP violation

```
class ShapeFactory {
    /* Factories are centralized places
     * where things are made
     * so we make a factory which
     * centralizes object creation
      * and keeps main code clean
      */
    public Shape createShape(int shapeType) {
        return switch (shapeType) {
            case 1 -> new Square();
            case 2 -> new Circle();
            default -> throw new IllegalArgumentException();
        };
    }
}
```

PROJECT PAUSE & REFLECT

Review the Refactoring Guru patterns and consider a recently implemented feature you or a group mate have completed.

Could a pattern have helped make it cleaner? Which one and how? Does it make sense to go back and refactor your code?