CEN206 Object-Oriented Programming

Week-9 (OO Design Principles & Design Patterns)

Spring Semester, 2024-2025

Download DOC-PDF, DOC-DOCX, SLIDE, PPTX

OO Design Principles & Design Patterns

Outline

- Design Patterns
- SOLID Principles
- Dependency Injection & Inversion of Control
- Practical Applications in Java

Introduction to Design Patterns

Design patterns are typical solutions to common problems in software design. They represent best practices evolved over time by experienced software developers.

- **Definition**: Reusable solution template for common design problems
- **Benefits**: Accelerate development, improve code quality and maintainability
- **Origins**: Inspired by architectural patterns (Christopher Alexander)

First Design Pattern book in architecture: https://www.amazon.com/Pattern-Language-Buildings-Construction-Environmental/dp/0195019199

The Gang of Four (GoF) Book

The seminal work in the field of design patterns is "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gang of Four).

This book categorizes design patterns into:

- **Creational Patterns**: Object creation mechanisms
- **Structural Patterns**: Object composition and relationships
- **Behavioral Patterns**: Object interaction and responsibility distribution

Reference:
https://www.amazon.com/gp/product/0201633612/

Factory Method Pattern

The Factory Method defines an interface for creating objects but lets subclasses decide which classes to instantiate.

```
// Product interface
interface Product {
    void operation();
}

// Concrete products
class ConcreteProductA implements Product {
    @Override
    public void operation() {
        System.out.println("ConcreteProductA operation");
    }
}

// Creator abstract class
abstract class Creator {
    public abstract Product createProduct();

    public void someOperation() {
        Product product = createProduct();
        product.operation();
    }
}

// Concrete creator
class ConcreteCreator extends Creator {
    @Override
    public Product createProduct() {
        return new ConcreteProductA();
    }
}
```

SOLID Principles

SOLID is a set of five design principles that help make software designs more understandable, flexible, and maintainable.

The five principles are:

1. **S**ingle Responsibility Principle
2. **O**pen/Closed Principle
3. **L**iskov Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

Resources:

- https://www.monterail.com/blog/solid-principles-cheatsheet-printable
- https://www.monterail.com/hubfs/PDF content/SOLID_cheatsheet.pdf
- https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

> "A class should have only one reason to change."

Each class should have a single responsibility or purpose. It should encapsulate only one aspect of the software's functionality.

```java
// Violates SRP
class Employee {
    public void calculatePay() { /* ... */ }
    public void saveToDatabase() { /* ... */ }
    public void generateReport() { /* ... */ }
}

// Follows SRP
class Employee {
    private String name;
    private double salary;
    // Employee properties and behavior only
}

class PayrollCalculator {
    public double calculatePay(Employee employee) { /* ... */ }
}

class EmployeeRepository {
    public void save(Employee employee) { /* ... */ }
}

class ReportGenerator {
    public void generateReport(Employee employee) { /* ... */ }
```

Open/Closed Principle (OCP)

> "Software entities should be open for extension but closed for modification."

You should be able to extend a class's behavior without modifying it.

```java
// Violates OCP
class Rectangle {
    public double width;
    public double height;
}

class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Rectangle) {
            Rectangle rect = (Rectangle) shape;
            return rect.width * rect.height;
        }
        // Add more conditions for new shapes
        return 0;
    }
}

// Follows OCP
interface Shape {
    double calculateArea();
}

class Rectangle implements Shape {
    private double width;
    private double height;

    @Override
    public double calculateArea() {
        return width * height;
    }
}

class Circle implements Shape {
    private double radius;

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

"Subtypes must be substitutable for their base types."

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

```
// Violates LSP
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;  // Square changes both dimensions
    }

    @Override
    public void setHeight(int height) {
        this.width = height;  // Square changes both dimensions
        this.height = height;
    }
}

// LSP violation example
void testRectangle(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert r.getArea() == 20; // Fails for Square
}
```

Interface Segregation Principle (ISP)

> "Clients should not be forced to depend on interfaces they do not use."

Many client-specific interfaces are better than one general-purpose interface.

```
// Violates ISP
interface Worker {
    void work();
    void eat();
    void sleep();
}

class Robot implements Worker {
    public void work() { /* ... */ }
    public void eat() { /* Not applicable */ }
    public void sleep() { /* Not applicable */ }
}

// Follows ISP
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

interface Sleepable {
    void sleep();
}

class Human implements Workable, Eatable, Sleepable {
    public void work() { /* ... */ }
    public void eat() { /* ... */ }
    public void sleep() { /* ... */ }
}

class Robot implements Workable {
    public void work() { /* ... */ }
}
```

> "High-level modules should not depend on low-level modules. Both should depend on abstractions."

> "Abstractions should not depend on details. Details should depend on abstractions."

```java
// Violates DIP
class LightBulb {
    public void turnOn() {
        // Turn on the light
    }

    public void turnOff() {
        // Turn off the light
    }
}

class Switch {
    private LightBulb bulb;

    public Switch() {
        this.bulb = new LightBulb();
    }

    public void operate() {
        // Logic to operate the switch
        bulb.turnOn();
    }
}

// Follows DIP
interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable {
    public void turnOn() {
        // Turn on the light
    }

    public void turnOff() {
        // Turn off the light
    }
}

class Fan implements Switchable {
    public void turnOn() {
        // Turn on the fan
    }

    public void turnOff() {
        // Turn off the fan
    }
}

class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }
```

Inversion of Control (IoC) and Dependency Injection (DI)

Inversion of Control is a design principle in which custom-written portions of a program receive the flow of control from a generic framework.

Dependency Injection is a specific form of IoC where the dependencies of a class are "injected" from the outside.

Resources:

- http://www.dotnet-stuff.com/tutorials/dependency-ınjection/understanding-and-implementing-inversion-of-control-container-ioc-container-using-csharp
- https://stackify.com/dependency-injection/
- https://www.tutorialsteacher.com/ioc/inversion-of-control
- https://www.wikiwand.com/en/Dependency_injection
- https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

```
    public Service(Repository repository) {
        this.repository = repository;
    }
}
```

2. Setter Injection

Dependencies are provided through setter methods.

```
class Service {
    private Repository repository;

    public void setRepository(Repository repository) {
        this.repository = repository;
    }
}
```

3. Interface Injection

Dependencies are provided through an interface method.

```
interface RepositoryInjector {
```

Benefits of Design Patterns and SOLID

- **Improved Code Quality**: More maintainable, flexible, and robust code
- **Reduced Complexity**: Break down complex problems into smaller, manageable parts
- **Better Communication**: Common vocabulary for discussing design solutions
- **Faster Development**: Reuse proven solutions rather than reinventing
- **Easier Testing**: More modular code is easier to test
- **Reduced Technical Debt**: Future changes require less rework

Security Best Practices in Design

When applying design patterns, also consider security aspects:

https://www.cisecurity.org/controls/cis-controls-list

- Ensure authentication and authorization are properly encapsulated
- Apply principle of least privilege
- Consider data validation at every boundary
- Implement proper error handling that doesn't leak information
- Design for security from the beginning

References

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Martin, R. C. (2003). Agile Software Development, Principles, Patterns, and Practices. Pearson.
- Freeman, E., Robson, E., Bates, B., Sierra, K. (2004). Head First Design Patterns. O'Reilly Media.
- Refactoring Guru. (n.d.). Design Patterns. https://refactoring.guru/design-patterns
- Martin, R. C. (n.d.). The Principles of OOD. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

And all the references linked throughout the presentation.

Recommended Practice

1. Implement the Factory Method pattern in a simple application
2. Refactor an existing codebase to apply SOLID principles
3. Create a small application using Dependency Injection
4. Identify design patterns in existing frameworks (Spring, JavaFX, etc.)
5. Practice explaining when and why to use specific patterns

Next Week

We'll continue exploring more design patterns and their practical implementations in Java.