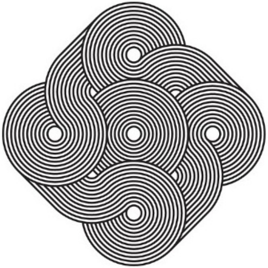
Design Patterns & SOLID Principles

Professional Software Architecture Guide



SE3140: System Design and Modeling | Fall 2025 | Presentation Date: January 7, 2026

Presentation Agenda

Design Patterns

- **Creational Patterns**
  - Singleton Pattern
  - Factory Method Pattern
- **Structural Patterns**
  - Adapter Pattern
  - Decorator Pattern
- **Behavioral Patterns**
  - Observer Pattern

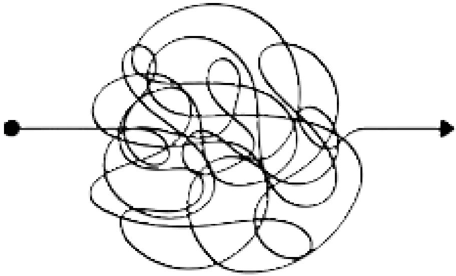
SOLID Principles

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation
- Dependency Inversion

Conclusion

Key Takeaways & Further Reading

The Challenge: Software Complexity



The Problem

Software systems become increasingly complex, making them:

- Difficult to maintain
- Hard to extend
- Costly to modify
- Error-prone

What Are Design Patterns?

Definition

General reusable solutions to common software design problems.

Not finished code, but templates/guidelines.

Key Characteristics

- **Best Practices**: Collective experience refined over time
- **Readability**: Improves code understanding
- **Reusability**: Promotes code reuse
- **Language Independent**

Patterns vs. Algorithms

Design Patterns	Algorithms
Solve design problems	Solve computational problems
High-level architecture	Low-level logic
Singleton, Factory	Sorting, Searching
Structural relationships	Step-by-step procedures

Patterns address *how* to structure code, not *what* to compute

Three Categories of Design Patterns

Creational

- Object creation mechanisms
- Increase flexibility
- Reuse existing code

```
Singleton
Factory Method
Builder
```

"How objects are created"

Structural

- Class/object composition
- Form larger structures
- Simplify relationships

```
Adapter
Decorator
Composite
```

"How objects are composed"

Behavioral

- Object communication
- Responsibility assignment
- Algorithm delegation

```
Observer
Strategy
Command
```

"How objects interact"

Creational Pattern: Singleton

Intent

Ensure a class has only **one instance** and provide a global access point.

Use Cases

- Configuration managers
- Logging services
- Database connections
- Caching systems

Key Characteristics:

- Private constructor
- Static instance
- Thread-safe access

```
class Singleton {
private:
    static Singleton* instance;

    // Private constructor
    Singleton() {
        cout << "Instance created";
    }

public:
    // Public access method
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton(
        }
        return instance;
    }

    void doSomething() {
        cout << "Singleton working";
    }
};
```

SOLID Principles: The Foundation

Five Principles for Maintainable Software

Letter	Principle	Core Idea
S	Single Responsibility	A class should have only **one reason** to change
O	Open/Closed	Open for **extension**, closed for **modification**
L	Liskov Substitution	Subtypes must be **replaceable** for their base types
I	Interface Segregation	Prefer focused interfaces over broad ones
D	Dependency Inversion	Depend on **abstractions**, not concretions

Goal

Build **Understandable**, **Maintainable**, and **Extensible** softwares.

SOLID: Single Responsibility Principle (SRP)

BAD Design - Violates SRP

```
class EmployeeBad {
    // Data management
    void setSalary(double s);
    double getSalary();

    // Report generation
    void generateReport();

    // Data persistence
    void saveToFile(string filename);
};
```

One class does everything → Hard to maintain

GOOD Design - Follows SRP

```
// Employee - only data
class Employee { ... };

// ReportGenerator - only reports
class ReportGenerator {
    static void generateReport(Employee e);
};

// Repository - only persistence
class EmployeeRepository {
    static void saveToFile(Employee e);
};
```

Each class has one responsibility → Easy to maintain

A class should have only ONE reason to change

SOLID: Open/Closed Principle (OCP)

Definition

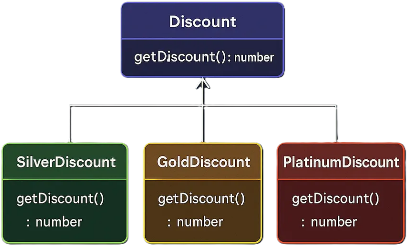
Software entities should be **open for extension** but **closed for modification**.

Key Insight

Add new functionality by **adding new code**, not by **changing existing code**.

How to achieve OCP:

- Use abstraction (interfaces, abstract classes)
- Leverage polymorphism
- Apply design patterns



Example: Shape System

- Shape (abstract class)
- `Circle`, `Rectangle`, `Triangle`
- Add Hexagon without modifying existing code

Key Takeaways

Design Patterns

- Reusable solutions
- Improve structure
- Creational, Structural, Behavioral

SOLID Principles

- Maintainable code
- Reduce technical debt
- Clean architecture

Further Reading

- Gamma et al. - Design Patterns (1994)
- R. C. Martin - Agile Principles (2002)
- Freeman et al. - Head First (2004)
- R. C. Martin - Clean Code (2008)

Use wisely, not rigidly

Thank You!

Questions?