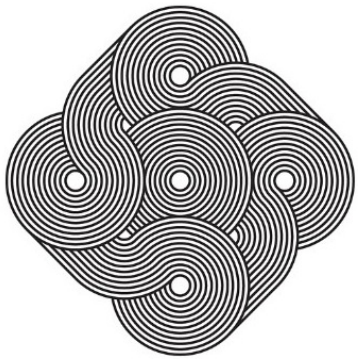

Design Patterns and SOLID Principles

Comprehensive Documentation

SE3140: System Design and Modeling

Faculty of Information and Communication Technologies

Fall 2025

Lecturer: Engr. Tekoh Palma

Presentation Date: 7th January 2026

Group Members

1. Kamdeu Yamdjeuson Neil Marshall [ICTU20241386]
2. Tuheu Tchoubi Pempem Moussa Fahdil [ICTU2024]
3. Kwete Ngnouba Rayan [ICTU2024]
4. Chijioke Emmanuel Izuchukwu [ICTU2024]

Contents

Key Concepts:
- Design Patterns
- SOLID Principles
- Software Architecture
- Maintainability
- Reusability

1 Introduction

Software systems inevitably grow in complexity as requirements evolve. Without proper design approaches, software becomes difficult to maintain, extend, and reuse. **Design Patterns** and **SOLID Principles** provide proven solutions to recurring software design problems and help developers create robust, flexible, and maintainable systems.

This document provides a comprehensive explanation of `Design Patterns` and **SOLID Principles**. Each design pattern is explained using:

- **Intent** - The purpose of the pattern
- **Motivation** - Why the pattern is needed
- **Structure** - UML or class relationships
- **Code Examples** - Practical implementations

Each SOLID principle is demonstrated with **clear, well-commented code examples**.

Note: *Design patterns are not silver bullets. They should be applied judiciously based on the specific context and requirements of your project.*

2 Design Patterns Overview

2.1 What Are Design Patterns?

Design patterns are **general reusable solutions** to commonly occurring problems in software design. They are not finished designs that can be directly converted into code; instead, they are **templates or guidelines** that describe how to solve a problem in different situations.

2.2 Difference Between Design Patterns and Algorithms

2.3 Types of Design Patterns

Design patterns are classified into three main categories:

Characteristic	Description
Best Practices	Represent solutions refined over time through collective experience
Readability	Improve code readability and maintainability
Reusability	Promote reusability and scalability of code
Language Independence	Can be implemented in any object-oriented language
Problem-Solving	Address specific design problems, not algorithmic problems

Table 1: Key Characteristics of Design Patterns

Aspect	Design Patterns	Algorithms
Purpose	Solve software **design** problems	Solve computational **processing** problems
Level	High-level system design	Low-level step-by-step logic
Output	Structure and relationships between classes	A specific result or computation
Reusability	Conceptual and architectural	Code-oriented and mathematical
Example	`Singleton`, `Factory`, `Observer`	Sorting, Searching, Graph traversal

Table 2: Design Patterns vs. Algorithms

3 Creational Design Patterns

Creational patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented.

3.1 Singleton Pattern

Intent: Ensure a class has only one instance and provide a global point of access to it.

Motivation: Singleton is useful when exactly one object is needed to coordinate actions across the system (e.g., configuration manager, logging service).

Structure: **Private** static instance, **private** constructor, **public** static access method.

```cpp
#include <iostream>

class Singleton {
private:
    // 1. Private static instance
    static Singleton* instance;

```

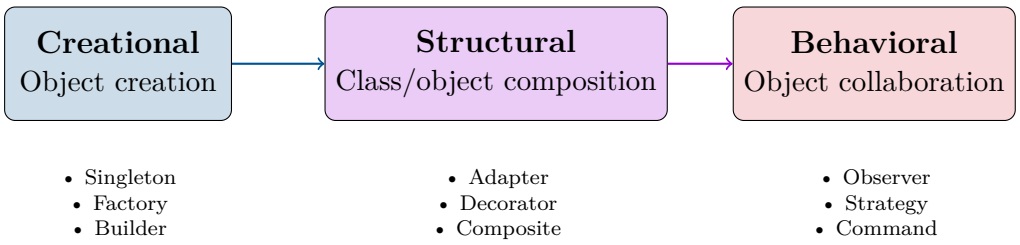

Figure 1: Three Categories of Design Patterns

```
    // 2. Private constructor
    Singleton() {
        std::cout << "Singleton instance created." << std::endl;
    }

    // Prevent copying
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

public:
    // 3. Public static access method
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }

    void doSomething() {
        std::cout << "Singleton is doing something." << std::endl;
    }
};

// Initialize static member
Singleton* Singleton::instance = nullptr;

// Usage example
int main() {
    // Get the singleton instance
    Singleton* s1 = Singleton::getInstance();
    s1->doSomething();

    // This will return the same instance
    Singleton* s2 = Singleton::getInstance();

    // s1 and s2 are the same instance
    std::cout << "Are they same? " << (s1 == s2 ? "Yes" : "No") << std
    ::endl;

    return 0;
```

```
}
```

Listing 1: Singleton Pattern Implementation in C++

3.2 Factory Method Pattern

Intent: Define an interface for creating objects but allow subclasses to alter the type of objects created.
Motivation: Factory Method promotes loose coupling by eliminating the need to bind application-specific classes into the code.
Structure: Product interface → ConcreteProduct → Creator (factory) class.

```java
// Product interface
interface Shape {
    void draw();
}

// Concrete Products
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}

// Creator (Factory) class
class ShapeFactory {
    // Factory method
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }

        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
```

```
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        }

        return null;
    }
}

// Client code
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        // Get objects without knowing the concrete classes
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();

        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        shape2.draw();

        Shape shape3 = shapeFactory.getShape("SQUARE");
        shape3.draw();
    }
}
```

Listing 2: Factory Method Pattern Implementation in Java

4 Structural Design Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures.

4.1 Adapter Pattern

Intent: Convert the interface of a class into another interface that clients expect.
Motivation: Adapter allows reuse of existing classes whose interfaces do not match the system requirements.
Structure: Target interface ← Adapter ← Adaptee.

```
#include <iostream>
#include <string>

// Legacy class with incompatible interface
class LegacyRectangle {
private:
```

```
    int x1, y1, x2, y2;

public:
    LegacyRectangle(int x1, int y1, int x2, int y2)
        : x1(x1), y1(y1), x2(x2), y2(y2) {
        std::cout << "LegacyRectangle: " << x1 << "," << y1
                  << " to " << x2 << "," << y2 << std::endl;
    }

    void oldDraw() {
        std::cout << "LegacyRectangle: oldDraw()." << std::endl;
    }
};

// New interface expected by client
class Rectangle {
public:
    virtual void draw(int x, int y, int width, int height) = 0;
    virtual ~Rectangle() {}
};

// Adapter class
class RectangleAdapter : public Rectangle {
private:
    LegacyRectangle* legacyRect;

public:
    RectangleAdapter(LegacyRectangle* rect) : legacyRect(rect) {}

    void draw(int x, int y, int width, int height) override {
        // Convert new interface to legacy interface
        int x2 = x + width;
        int y2 = y + height;

        // Create legacy rectangle with converted parameters
        LegacyRectangle adaptedRect(x, y, x2, y2);
        adaptedRect.oldDraw();

        std::cout << "Adapter: Translated new interface to legacy." <<
    std::endl;
    }
};

// Client code
int main() {
    LegacyRectangle* legacy = new LegacyRectangle(0, 0, 20, 30);

    // Using adapter to make legacy compatible with new interface
    Rectangle* adapter = new RectangleAdapter(legacy);
    adapter->draw(5, 10, 50, 60);  // Uses new interface
```

```
56
57      delete legacy;
58      delete adapter;
59
60      return 0;
61  }
```

Listing 3: Adapter Pattern Implementation in C++

4.2 Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically.
Motivation: Decorator provides a flexible alternative to subclassing for extending functionality.
Structure: Component interface → ConcreteComponent → Decorator → ConcreteDecorator.

```
1  #include <iostream>
2  #include <string>
3
4  // Component interface
5  class Coffee {
6  public:
7      virtual std::string getDescription() const = 0;
8      virtual double cost() const = 0;
9      virtual ~Coffee() {}
10 };
11
12 // Concrete Component
13 class SimpleCoffee : public Coffee {
14 public:
15     std::string getDescription() const override {
16         return "Simple Coffee";
17     }
18
19     double cost() const override {
20         return 2.0;
21     }
22 };
23
24 // Decorator base class
25 class CoffeeDecorator : public Coffee {
26 protected:
27     Coffee* coffee;
28
29 public:
30     CoffeeDecorator(Coffee* c) : coffee(c) {}
31
32     std::string getDescription() const override {
```

```
        return coffee->getDescription();
    }

    double cost() const override {
        return coffee->cost();
    }
};

// Concrete Decorators
class MilkDecorator : public CoffeeDecorator {
public:
    MilkDecorator(Coffee* c) : CoffeeDecorator(c) {}

    std::string getDescription() const override {
        return coffee->getDescription() + ", Milk";
    }

    double cost() const override {
        return coffee->cost() + 0.5;
    }
};

class SugarDecorator : public CoffeeDecorator {
public:
    SugarDecorator(Coffee* c) : CoffeeDecorator(c) {}

    std::string getDescription() const override {
        return coffee->getDescription() + ", Sugar";
    }

    double cost() const override {
        return coffee->cost() + 0.2;
    }
};

// Client code
int main() {
    // Start with simple coffee
    Coffee* coffee = new SimpleCoffee();
    std::cout << "Order: " << coffee->getDescription()
              << " Cost: $" << coffee->cost() << std::endl;

    // Add milk
    coffee = new MilkDecorator(coffee);
    std::cout << "Order: " << coffee->getDescription()
              << " Cost: $" << coffee->cost() << std::endl;

    // Add sugar
    coffee = new SugarDecorator(coffee);
    std::cout << "Final Order: " << coffee->getDescription()
```

```
                << " Cost: $" << coffee->cost() << std::endl;

    delete coffee;
    return 0;
}
```

Listing 4: Decorator Pattern Implementation in C++

5 Behavioral Design Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

5.1 Observer Pattern

Intent: Define a one-to-many dependency so that when one object changes state, all its dependents are notified automatically.
Motivation: Useful in event-driven systems where multiple components must react to changes in another component without being tightly coupled.
Structure: Subject interface → ConcreteSubject ↔ Observer interface → ConcreteObserver.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

// Forward declaration
class Observer;

// Subject interface
class Subject {
public:
    virtual void attach(Observer* observer) = 0;
    virtual void detach(Observer* observer) = 0;
    virtual void notify() = 0;
    virtual ~Subject() {}
};

// Observer interface
class Observer {
public:
    virtual void update(const std::string& message) = 0;
    virtual ~Observer() {}
};

// Concrete Subject
```

```cpp
class NewsPublisher : public Subject {
private:
    std::vector<Observer*> observers;
    std::string latestNews;

public:
    void attach(Observer* observer) override {
        observers.push_back(observer);
        std::cout << "Observer attached." << std::endl;
    }

    void detach(Observer* observer) override {
        observers.erase(std::remove(observers.begin(), observers.end(), observer),
                        observers.end());
        std::cout << "Observer detached." << std::endl;
    }

    void notify() override {
        std::cout << "Notifying " << observers.size() << " observers..." << std::endl;
        for (Observer* observer : observers) {
            observer->update(latestNews);
        }
    }

    void publishNews(const std::string& news) {
        latestNews = news;
        std::cout << "\n=== Publishing News: " << news << " ===" << std::endl;
        notify();
    }
};

// Concrete Observer
class NewsSubscriber : public Observer {
private:
    std::string name;

public:
    NewsSubscriber(const std::string& name) : name(name) {}

    void update(const std::string& message) override {
        std::cout << "Subscriber [" << name << "] received: " << message << std::endl;
    }
};

// Client code
int main() {
```

```
    // Create publisher
    NewsPublisher publisher;

    // Create subscribers
    NewsSubscriber subscriber1("Alice");
    NewsSubscriber subscriber2("Bob");
    NewsSubscriber subscriber3("Charlie");

    // Attach subscribers
    publisher.attach(&subscriber1);
    publisher.attach(&subscriber2);
    publisher.attach(&subscriber3);

    // Publish news (observers get notified)
    publisher.publishNews("Breaking: New Design Pattern Course
    Available!");
    publisher.publishNews("Update: SOLID Principles Workshop Tomorrow")
    ;

    // Detach one subscriber
    publisher.detach(&subscriber2);

    // Publish more news
    publisher.publishNews("Reminder: Project Submission Deadline
    Extended");

    return 0;
}
```

Listing 5: Observer Pattern Implementation in C++

6 SOLID Principles

The SOLID principles are five design principles that help create maintainable, understandable, and flexible software. They were introduced by Robert C. Martin (Uncle Bob).

6.1 Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility.

```
#include <iostream>
#include <string>
#include <fstream>

// BAD DESIGN: Class with multiple responsibilities
```

Letter	Principle	Core Idea
S	Single Responsibility	A class should have only one reason to change
O	Open/Closed	Software entities should be open for extension but closed for modification
L	Liskov Substitution	Objects of a superclass should be replaceable with objects of a subclass
I	Interface Segregation	Clients should not be forced to depend on interfaces they don't use
D	Dependency Inversion	Depend on abstractions, not on concretions

Table 3: SOLID Principles Overview

```cpp
class EmployeeBad {
private:
    std::string name;
    std::string position;
    double salary;

public:
    EmployeeBad(const std::string& n, const std::string& p, double s)
        : name(n), position(p), salary(s) {}

    // Responsibility 1: Employee data management
    void setSalary(double s) { salary = s; }
    double getSalary() const { return salary; }

    // Responsibility 2: Report generation (should be separate)
    void generateReport() {
        std::cout << "Report for " << name << std::endl;
        std::cout << "Position: " << position << std::endl;
        std::cout << "Salary: $" << salary << std::endl;
    }

    // Responsibility 3: Data persistence (should be separate)
    void saveToFile(const std::string& filename) {
        std::ofstream file(filename);
        file << name << "," << position << "," << salary;
        file.close();
    }
};

// GOOD DESIGN: Separated responsibilities
class Employee {
private:
    std::string name;
    std::string position;
    double salary;

```

```
public:
    Employee(const std::string& n, const std::string& p, double s)
        : name(n), position(p), salary(s) {}

    // Only employee-related methods
    void setSalary(double s) { salary = s; }
    double getSalary() const { return salary; }
    std::string getName() const { return name; }
    std::string getPosition() const { return position; }
};

// Separate class for report generation
class ReportGenerator {
public:
    static void generateEmployeeReport(const Employee& emp) {
        std::cout << "\n=== Employee Report ===" << std::endl;
        std::cout << "Name: " << emp.getName() << std::endl;
        std::cout << "Position: " << emp.getPosition() << std::endl;
        std::cout << "Salary: $" << emp.getSalary() << std::endl;
        std::cout << "======================" << std::endl;
    }
};

// Separate class for data persistence
class EmployeeRepository {
public:
    static void saveToFile(const Employee& emp, const std::string&
    filename) {
        std::ofstream file(filename);
        file << emp.getName() << "," << emp.getPosition() << "," << emp
    .getSalary();
        file.close();
        std::cout << "Employee data saved to " << filename << std::endl
    ;
    }

    static Employee loadFromFile(const std::string& filename) {
        std::ifstream file(filename);
        std::string name, position;
        double salary;

        std::getline(file, name, ',');
        std::getline(file, position, ',');
        file >> salary;

        return Employee(name, position, salary);
    }
};

// Client code demonstrating SRP
```

```cpp
int main() {
    // Create employee
    Employee emp("John Doe", "Software Engineer", 75000.0);

    // Generate report using separate class
    ReportGenerator::generateEmployeeReport(emp);

    // Save data using separate class
    EmployeeRepository::saveToFile(emp, "employee_data.txt");

    // Load data using separate class
    Employee loadedEmp = EmployeeRepository::loadFromFile("
    employee_data.txt");
    ReportGenerator::generateEmployeeReport(loadedEmp);

    return 0;
}
```

Listing 6: Single Responsibility Principle Example

6.2 Open/Closed Principle (OCP)

Definition: Software entities (classes, modules, functions) should be open for extension but closed for modification.

```cpp
#include <iostream>
#include <vector>
#include <memory>

// BAD DESIGN: Not following OCP
class ShapeBad {
    // Type field to distinguish shapes
    enum Type { CIRCLE, SQUARE, TRIANGLE } type;

    // For circle
    double radius;

    // For square
    double side;

    // For triangle
    double base, height;

public:
    ShapeBad(Type t, double r) : type(t), radius(r) {} // Circle
    ShapeBad(Type t, double s) : type(t), side(s) {}    // Square
    ShapeBad(Type t, double b, double h) : type(t), base(b), height(h)
    {} // Triangle

    double area() const {
```

```
        switch (type) {
            case CIRCLE: return 3.14159 * radius * radius;
            case SQUARE: return side * side;
            case TRIANGLE: return 0.5 * base * height;
            default: return 0;
        }
    }
    // Problem: Adding new shape requires modifying this class
};

// GOOD DESIGN: Following OCP
class Shape {
public:
    virtual double area() const = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }
};

class Square : public Shape {
private:
    double side;

public:
    Square(double s) : side(s) {}

    double area() const override {
        return side * side;
    }
};

class Triangle : public Shape {
private:
    double base, height;

public:
    Triangle(double b, double h) : base(b), height(h) {}

    double area() const override {
        return 0.5 * base * height;
```

```
    }
};

// NEW SHAPE: Can be added without modifying existing code
class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const override {
        return width * height;
    }
};

// AreaCalculator is closed for modification but open for extension
class AreaCalculator {
public:
    static double totalArea(const std::vector<std::shared_ptr<Shape>>&
    shapes) {
        double total = 0;
        for (const auto& shape : shapes) {
            total += shape->area();
        }
        return total;
    }
};

// Client code demonstrating OCP
int main() {
    std::vector<std::shared_ptr<Shape>> shapes;

    // Add various shapes
    shapes.push_back(std::make_shared<Circle>(5.0));
    shapes.push_back(std::make_shared<Square>(4.0));
    shapes.push_back(std::make_shared<Triangle>(3.0, 6.0));

    // New shape added without modifying AreaCalculator
    shapes.push_back(std::make_shared<Rectangle>(4.0, 6.0));

    // Calculate total area
    double total = AreaCalculator::totalArea(shapes);
    std::cout << "Total area of all shapes: " << total << std::endl;

    return 0;
}
```

Listing 7: Open/Closed Principle Example

6.3 Liskov Substitution Principle (LSP)

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

```cpp
#include <iostream>
#include <vector>
#include <memory>

// BAD DESIGN: Violating LSP
class RectangleBad {
protected:
    double width, height;

public:
    RectangleBad(double w, double h) : width(w), height(h) {}

    virtual void setWidth(double w) { width = w; }
    virtual void setHeight(double h) { height = h; }

    double getWidth() const { return width; }
    double getHeight() const { return height; }

    double area() const { return width * height; }
};

class SquareBad : public RectangleBad {
public:
    SquareBad(double side) : RectangleBad(side, side) {}

    void setWidth(double w) override {
        width = w;
        height = w; // Violates LSP: Changes height too
    }

    void setHeight(double h) override {
        height = h;
        width = h; // Violates LSP: Changes width too
    }
};

// Problematic usage
void testRectangleBad(RectangleBad& rect) {
    rect.setWidth(5);
    rect.setHeight(4);
    std::cout << "Expected area: 20, Actual area: " << rect.area() <<
    std::endl;
    // For SquareBad, this prints 16, not 20!
}

// GOOD DESIGN: Following LSP
```

```cpp
class ShapeLSP {
public:
    virtual double area() const = 0;
    virtual ~ShapeLSP() {}
};

class RectangleLSP : public ShapeLSP {
protected:
    double width, height;

public:
    RectangleLSP(double w, double h) : width(w), height(h) {}

    virtual void setWidth(double w) { width = w; }
    virtual void setHeight(double h) { height = h; }

    double getWidth() const { return width; }
    double getHeight() const { return height; }

    double area() const override { return width * height; }
};

class SquareLSP : public ShapeLSP {
private:
    double side;

public:
    SquareLSP(double s) : side(s) {}

    void setSide(double s) { side = s; }
    double getSide() const { return side; }

    double area() const override { return side * side; }
};

// Factory functions to create shapes
std::shared_ptr<RectangleLSP> createRectangle(double width, double
    height) {
    return std::make_shared<RectangleLSP>(width, height);
}

std::shared_ptr<SquareLSP> createSquare(double side) {
    return std::make_shared<SquareLSP>(side);
}

// Client code demonstrating LSP
int main() {
    std::vector<std::shared_ptr<ShapeLSP>> shapes;

    // Add different shapes
```

```
    shapes.push_back(createRectangle(5, 4));
    shapes.push_back(createSquare(5));

    // Calculate total area
    double totalArea = 0;
    for (const auto& shape : shapes) {
        totalArea += shape->area();
        std::cout << "Shape area: " << shape->area() << std::endl;
    }

    std::cout << "\nTotal area: " << totalArea << std::endl;

    // Demonstrate that each shape maintains its own invariants
    auto rect = createRectangle(10, 5);
    std::cout << "\nRectangle area: " << rect->area() << std::endl;

    auto square = createSquare(10);
    std::cout << "Square area: " << square->area() << std::endl;

    return 0;
}
```

Listing 8: Liskov Substitution Principle Example

7 Conclusion

Design Patterns and SOLID Principles are essential tools for professional software development. When applied correctly, they lead to systems that are easier to understand, maintain, test, and extend.

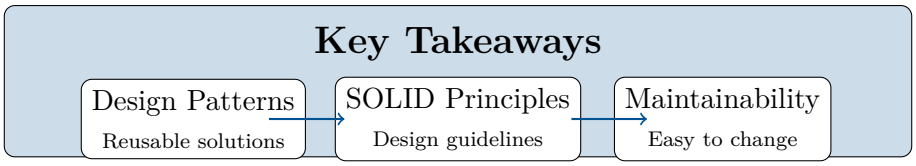

Figure 2: Design Patterns and SOLID Principles Synergy

- **Design Patterns** provide **templates** for solving common problems
- **SOLID Principles** provide **guidelines** for creating maintainable code
- Together, they form a **powerful toolkit** for software architects and developers
- Proper application leads to **reduced technical debt** and **easier maintenance**

Note: *Remember that patterns and principles should be applied thoughtfully, not dogmatically. The goal is to create software that is easy to change and maintain, not to use every pattern possible.*

FURTHER READING:

- Gamma, E., et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*
- Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices*
- Freeman, E., et al. (2004). *Head First Design Patterns*