SOLID Principles: A Comprehensive Course on Object-Oriented Design

Bahey Shalash

February 7, 2025

Contents

1 Introduction

Object-oriented design (OOD) is a paradigm used in modern software development. As projects grow in size and complexity, adhering to design principles becomes essential in ensuring that code is maintainable, flexible, and scalable.

One influential set of guidelines is known as the **SOLID** principles. These five principles help you design robust systems where changes in one part of the code have minimal side effects in others. Although originally popularized in the context of object-oriented programming (and often exemplified in C++ or Java), these principles are *language agnostic* and apply to any paradigm that supports modular design.

The SOLID principles are:

- **S**ingle Responsibility Principle (SRP)
- **O**pen/Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

In this course, we will explore each principle in detail, describe common pitfalls (what you should *not* do), and illustrate best practices (what you should do). We will also provide UML diagrams to help visualize proper design, along with pseudocode and C++ examples where applicable.

2 Single Responsibility Principle (SRP)

2.1 Theory and Explanation

The Single Responsibility Principle states that **a class (or module) should have only one reason to change**. In other words, each class should have a single, well-defined responsibility.

Why it matters:

- *Maintainability:* When a class handles only one responsibility, modifications are easier because changes in one area do not ripple into unrelated areas.

- *Testability:* Smaller, focused classes are simpler to test.
- *Reusability:* Components with a single purpose can be reused in different contexts.

2.2 Common Pitfall (What Not to Do)

A typical violation is a class that handles multiple responsibilities. For example, consider a class that manages both business logic and user interface (UI) code. In pseudocode:

```
class Employee {
    // Business logic: Calculate pay
    method calculatePay() { ... }

    // Presentation logic: Print employee details
    method printEmployee() { ... }

    // Persistence logic: Save to database
    method saveToDatabase() { ... }
}
```

Listing 1: A class with multiple responsibilities (violates SRP)

Any change in presentation or persistence would require modifying the same class that handles core business logic.

2.3 Best Practice (What to Do)

Separate responsibilities into distinct classes or modules. For example:

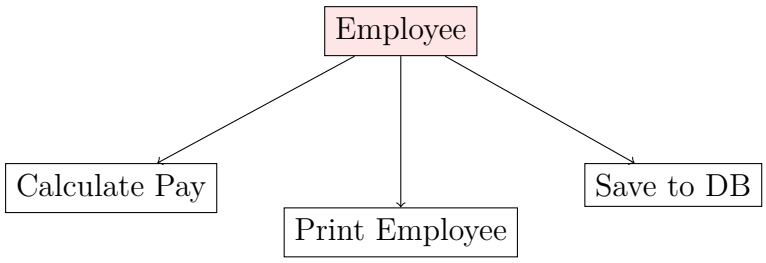
```
// Business logic class
class Employee {
    method calculatePay() { ... }
}

// Presentation class
class EmployeePrinter {
    method print(Employee e) { ... }
}

// Persistence class
class EmployeeRepository {
    method save(Employee e) { ... }
```

```
14 }
```

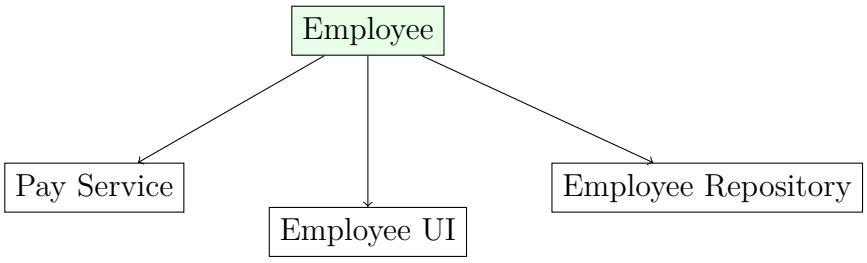
Listing 2: Separate classes adhering to SRP

2.4 UML Diagrams

Bad Design Diagram



Good Design Diagram



3 Open/Closed Principle (OCP)

3.1 Theory and Explanation

The Open/Closed Principle states that **software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**. This means you should be able to add new functionality without changing existing code.

Why it matters:

- *Stability:* Changes or extensions do not break existing code.
- *Flexibility:* New behaviors can be introduced by adding new modules rather than modifying tested code.

3.2 Common Pitfall (What Not to Do)

Using type-checks or conditional logic within a method that forces you to modify the method when adding new cases. For example:

```
class Shape {
    enum Type { CIRCLE, SQUARE };
    attribute type;

    method draw() {
        if (type == CIRCLE) {
            // Draw circle
        } else if (type == SQUARE) {
            // Draw square
        }
        // Adding a new shape requires modifying this method.
    }
}
```

Listing 3: Type-checking within a method (violates OCP)

3.3 Best Practice (What to Do)

Use abstraction and polymorphism to allow extension without modification. For example:

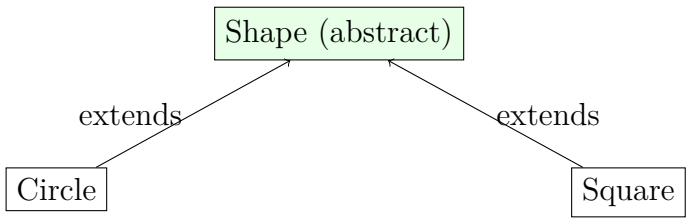
```
// Abstract class
abstract class Shape {
    abstract method draw();
}

class Circle extends Shape {
    method draw() {
        // Draw circle
    }
}

class Square extends Shape {
    method draw() {
        // Draw square
    }
}

// The client code uses the abstract Shape interface.
method render(Shape s) {
```

```
20      s.draw();
21  }
```

Listing 4: Polymorphism allows extension without modification (adheres to OCP)

3.4 UML Diagrams

Good Design UML Diagram



4 Liskov Substitution Principle (LSP)

4.1 Theory and Explanation

The Liskov Substitution Principle states that **objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program**. In short, a derived class must be substitutable for its base class.

Why it matters:

- *Robust Inheritance:* Ensures that a subclass does not alter the expected behavior of its base class.
- *Predictability:* Clients using the base class do not need to know about special behaviors of derived classes.

4.2 Common Pitfall (What Not to Do)

A subclass may override a method in a way that violates the contract of the base class. For instance, consider a class hierarchy for birds:

```
class Bird {
    method fly() { ... }
}

class Ostrich extends Bird {
    // Ostrich cannot fly.
    method fly() {
        throw Error("Ostrich cannot fly");
    }
}
```

Listing 5: Bird class hierarchy with a violation (Ostrich cannot fly)

Substituting an `Ostrich` for a `Bird` in code that expects all birds to fly violates LSP.

4.3 Best Practice (What to Do)

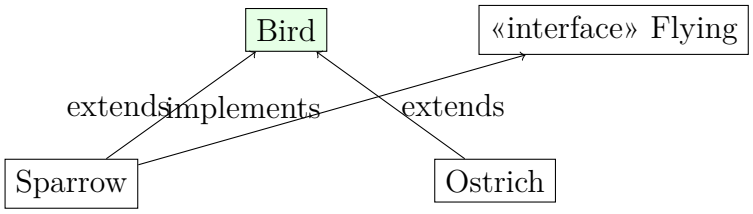
Design the hierarchy so that only birds that can fly implement the flying behavior. For example:

```
class Bird { ... }

interface Flying {
    method fly();
}

class Sparrow extends Bird implements Flying {
    method fly() { ... }
}

class Ostrich extends Bird {
    // Does not implement fly()
}
```

Listing 6: Separate flying functionality using an interface (adheres to LSP)

4.4 UML Diagrams

Good Design UML Diagram



5 Interface Segregation Principle (ISP)

5.1 Theory and Explanation

The Interface Segregation Principle states that **clients should not be forced to depend on methods they do not use**. Instead of one fat interface, it is better to have several smaller, client-specific interfaces.

Why it matters:

- *Decoupling:* Clients are only exposed to the functionality they need.
- *Flexibility:* Changes to one interface have minimal impact on clients that do not use it.

5.2 Common Pitfall (What Not to Do)

Forcing a class to implement a large interface even when it only needs a subset of the functionality. For example:

```
interface Worker {
    method work();
    method eat();
}

class Human implements Worker {
    method work() { ... }
    method eat() { ... }
}

class Robot implements Worker {
    method work() { ... }
```

```
13      // Robot does not need an eat() method, but is forced to
            implement it.
14      method eat() { throw Error("Not applicable"); }
15  }
```

Listing 7: Large interface forcing unnecessary methods (violates ISP)

5.3 Best Practice (What to Do)

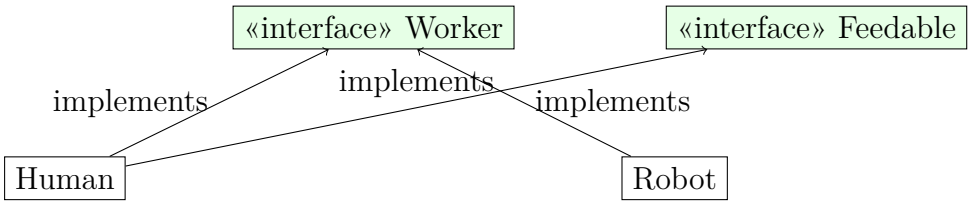
Break the large interface into smaller, more focused ones:

```
1   interface Worker {
2       method work();
3   }
4
5   interface Feedable {
6       method eat();
7   }
8
9   class Human implements Worker, Feedable {
10      method work() { ... }
11      method eat() { ... }
12  }
13
14  class Robot implements Worker {
15      method work() { ... }
16  }
```

Listing 8: Segregated interfaces (adheres to ISP)

5.4 UML Diagrams

Good Design UML Diagram



6 Dependency Inversion Principle (DIP)

6.1 Theory and Explanation

The Dependency Inversion Principle states that **high-level modules should not depend on low-level modules; both should depend on abstractions**. In addition, abstractions should not depend on details; details should depend on abstractions.

Why it matters:

- *Decoupling:* This approach reduces the coupling between components, making the system more modular.
- *Testability:* By programming against abstractions (e.g., interfaces), it is easier to substitute implementations during testing.

6.2 Common Pitfall (What Not to Do)

A high-level module directly instantiates or depends on a low-level concrete class. For example:

```
class Light {
    method turnOn() { ... }
}

class Switch {
    attribute light = new Light(); // Direct dependency on a
        concrete class

    method operate() {
        light.turnOn();
    }
}
```

Listing 9: Direct dependency on a concrete class (violates DIP)

6.3 Best Practice (What to Do)

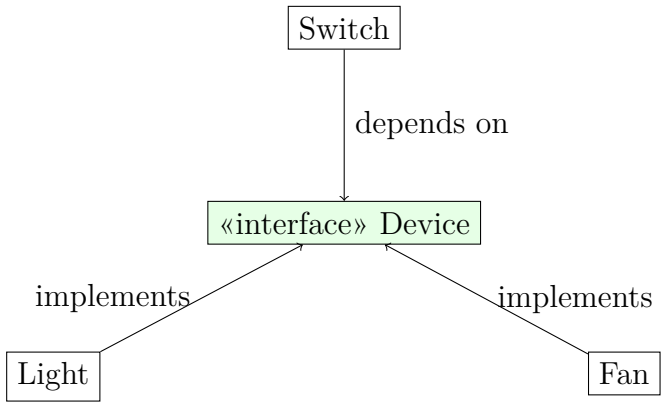
Introduce an abstraction (e.g., an interface) that both the high-level module and low-level modules depend on:

```
interface Device {
    method turnOn();
}

class Light implements Device {
    method turnOn() { ... }
}

class Switch {
    // Dependency injected via the abstraction.
    attribute device : Device;

    method operate() {
        device.turnOn();
    }
}
```

Listing 10: Dependency on an abstraction (adheres to DIP)

6.4 UML Diagrams

Good Design UML Diagram



7 Conclusion

The SOLID principles provide a strong foundation for designing flexible, maintainable, and robust software systems. By applying these guidelines:

- **SRP** ensures that classes have a single responsibility, reducing the chance of unexpected side effects.
- **OCP** promotes extending behavior via new modules rather than modifying existing, tested code.
- **LSP** guarantees that subclasses enhance rather than break the contracts established by their base classes.
- **ISP** minimizes the burden on clients by providing only the interfaces they need.
- **DIP** decouples high-level policies from low-level details, enabling more modular and testable code.

These principles are applicable regardless of the programming language or framework you use. Whether you are writing code in C++, Java, Python, or any other language, embracing SOLID will lead to cleaner, more maintainable software design.

Further Reading:

- *Agile Software Development, Principles, Patterns, and Practices* by Robert C. Martin
- *Clean Architecture: A Craftsman's Guide to Software Structure and Design* by Robert C. Martin