7大软件设计原则

丁宁

February 29, 2024

1 Open-Closed Principle 开闭原则

1.1 定义

一个软件实体应该对扩展开放(Open for extension), 对修改关闭(Closed to modification)。

↪ Add new features or behaviors without altering the existing codebases.

1.2 优势

- 扩展性(Flexible): Accommodates new requirements readily.
- 可维护性(Maintainable): Changes are localized, reducing the risk of bugs.
- 可复用性(Reusable): Components can be used in new contexts.

1.3 实例

反例: Every time you need to handle a new shape (e.g., triangle), you directly modify the `calculateShapeArea` method. This violates OCP.

```
// Bad example - violates OCP
public class AreaCalculator {
    public double calculateShapeArea(String shapeType) {
        if (shapeType.equals("circle")) {
            // Code to calculate circle area
            return 3.14 * radius * radius; // Example
        } else if (shapeType.equals("rectangle")) {
            // Code to calculate rectangle area
            return width * height; // Example
        }
        // ...imagine more shapes here
        return 0;
    }
}
```

正例:

```
class Circle implements Shape {
    private double radius;
    // ...constructor, etc.

    @Override
    public double getArea() {
```

```
        return Math.PI * radius * radius;
    }
}

class Rectangle implements Shape {
    private double width;
    private double height;
    // ...constructor, etc.

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

2 Dependence Inversion Principle 依赖倒置原则

2.1 定义

1. 高层模块不应该依赖低层模块，二者都应该依赖其抽象。
   High-level modules should not depend on low-level modules; both should depend on abstractions.
2. 抽象不应该依赖细节；细节应该依赖抽象。
   Abstractions should not depend on details; details should depend on abstractions.

2.2 优势

- 减少耦合性(Loose coupling): Components become less reliant on each other's concrete implementations, making your code adaptable to changes.
- 扩展性(Flexibility): You can replace low-level components with different implementations as long as they adhere to the shared abstraction.
- 可维护性(Testability): It's easy to substitute real components with test doubles (mocks) during testing.

2.3 实例

反例：`PaymentProcessor` is directly tied to the `PayPalPaymentGateway`. Switching to a different payment provider would require changing this class.

```
class PaymentProcessor {
    private PayPalPaymentGateway payPalGateway = new
   PayPalPaymentGateway();

    public void processPayment(double amount) {
        payPalGateway.pay(amount);
    }
}
```

正例：`PaymentProcessor` now depends on the `PaymentGateway` abstraction, not a specific gateway. You can inject different implementations of `PaymentGateway` (PayPal, Stripe, etc.) into `PaymentProcessor` without changing its code.

```
class PayPalPaymentGateway implements PaymentGateway {
    @Override
    public void pay(double amount) {
        // PayPal-specific logic
    }
}

class StripePaymentGateway implements PaymentGateway {
    @Override
    public void pay(double amount) {
        // Stripe-specific logic
    }
}

//Modified PaymentProcessor (with Dependency Injection):
class PaymentProcessor {
    private PaymentGateway paymentGateway;

    // Constructor injection
    public PaymentProcessor(PaymentGateway paymentGateway) {
        this.paymentGateway = paymentGateway;
    }

    public void processPayment(double amount) {
        paymentGateway.pay(amount);
    }
}
```

2.4 Dependence Injection 依赖注入

一个类不直接创建或查找它所需要的依赖对象，而是由外部提供这些依赖。

Without DI: Classes create their own dependencies - An object that needs to use another object will directly instantiate that object inside itself.

```
    class EmailService {
    private DatabaseConnection databaseConnection;

    public EmailService() {
        this.databaseConnection = new DatabaseConnection("server1",
    "myDatabase"); // Tightly coupled here
    }

    // ... methods to send emails
}
```

Constructor Injection: Dependencies are passed through the class's constructor.

```
class EmailService {
    private DatabaseConnection databaseConnection;

    public EmailService(DatabaseConnection databaseConnection) {
        this.databaseConnection = databaseConnection;
    }
    // ... methods to send emails
}
```

Setter Injection: Dependencies are set through setter methods.

```
class EmailService {
    private DatabaseConnection databaseConnection;

    public void setDatabaseConnection(DatabaseConnection
   databaseConnection) {
        this.databaseConnection = databaseConnection;
    }
    // ... methods to send emails
}
```

Interface Injection：The dependency implements an interface, and is injected through a method accepting that interface

```
interface Logger {
    void log(String message);
}

class EmailService {
    private Logger logger;

    public void setLogger(Logger logger) {
        this.logger = logger;
    }
    // ... methods to send emails
}
```

3 Single Responsibility Principle 单一职责原则

3.1 定义

不要存在多于一个导致类变更的原因
A class should have only one reason to change.

一个类，接口，方法只负责一项职责
Each class should encapsulate a single, well-defined responsibility.

3.2 优势

- 可读性(Readability)，可维护性(Maintainability)：Code with clear responsibilities is easier to understand, modify, and fix.
- 可变更性(Reusability)：Classes designed for a single task are more easily spotted when something went wrong.

3.3 实例

反例：The `Employee` class handles calculating pay, generating payslips, and saving data. This violates SRP.

```
class Employee {
    private double salary;
    private int hoursWorked;
    // ... other employee details

```

```
    public double calculatePay() {
        return salary * hoursWorked;
    }

    public void generatePaySlip() {
        // Logic to format and generate the payslip
    }

    public void savePayDetailsToDatabase() {
        // Logic to save payment details to database
    }
}
```

正例：If the tax laws governing pay calculation change, you only need to modify the `PayCalculator` class.

```
// Employee class
class Employee {
    private double salary;
    private int hoursWorked;
    // ... other employee details
}

// Pay calculations
class PayCalculator {
    public double calculatePay(Employee employee) {
        return employee.getSalary() * employee.getHoursWorked();
    }
}

// Payslip generation
class PayslipGenerator {
    public void generatePayslip(Employee employee, double pay) {
        // Logic to format and generate the payslip
    }
}

// Database interaction
class PaymentDatabase {
    public void savePaymentDetails(Employee employee, double pay) {
        // Logic to save payment details to database
    }
}
```

4 Interface Segregation Principle 接口隔离原则

4.1 定义

用多个专门的接口，而不是使用单一的总接口，客户端不应该依赖它不需要的接口。

clients (classes that use an interface) shouldn't be forced to depend on methods they don't use. Instead of large, bloated interfaces, it's better to create smaller, more focused ones.

4.2 优势

- Flexibility: Classes only have to implement the methods they truly need.

- Maintainability: Changes to one part of the interface won't unnecessarily affect unrelated classes.
- Reduced Coupling: Classes become less dependent on each other.

4.3 实例

反例：A simple printer would be forced to provide dummy implementations for `scanDocument` and `faxDocument` even though it doesn't have those capabilities.

```
interface AllInOneDevice {
    void printDocument();
    void scanDocument();
    void faxDocument();
}
```

正例：Classes now implement only the interfaces relevant to them. Adding a new function, like document stapling, only requires creating a `Stapler` interface and adding it to relevant devices.

```
interface Printer {
    void printDocument();
}

interface Scanner {
    void scanDocument();
}

interface FaxMachine {
    void faxDocument();
}

class MultiFunctionDevice implements Printer, Scanner, FaxMachine {
    // Implement all methods
}

class SimplePrinter implements Printer {
    // Implement only printDocument
}
```

5 Law of Demeter 迪米特法则

5.1 定义

- Each unit (a method or class) should have limited knowledge of other units.
- Only talk to your "immediate friends." Avoid chains of calls like `objectA.getObjectB().getObjectC().doSomething()`.

5.2 优势

- 低耦合性(Low Coupling): Changes in one part of your system are less likely to have ripple effects on distant components.
- 可维护性(Easier Maintenance): Isolated units are simpler to understand and modify.

- 可复用性(Enhanced Reusability): Components with fewer dependencies on the overall system are more reusable.

5.3 实例

反例：The `Car` class knows too much about the internals of `Engine` and `FuelTank`.

```
class Car {
    private Engine engine;

    public void start() {
        engine.getFuelTank().supplyFuel();
    }
}
```

正例：`Car` only interacts with its direct collaborator, `Engine`. It doesn't need to know about the `FuelTank`. If the implementation of `Engine` or `FuelTank` changes, the `Car` class likely remains unaffected.

```
class Car {
    private Engine engine;

    public void start() {
        engine.start();
    }
}

class Engine {
    private FuelTank fuelTank;

    public void start() {
        fuelTank.supplyFuel();
    }
}
```

6 Liskov Substitution Principle 里氏替换原则

6.1 定义

一个软件如果适用于一个父类的话，那一定适用于其子类。所有应用父类的地方必须能透明的使用其子类的对象，子类对象能替换父类对象，而程序逻辑不变。
The LSP states that you should be able to substitute a subclass for its superclass without breaking your program's behavior. This ensures that your code works as intended, even when using different subtypes.

6.2 优势

- 防止继承泛滥(Inheritance Done Right): LSP helps you use inheritance correctly and prevents unexpected behavior.
- 扩展性(Flexible Design): Your code can easily handle new subtypes without modifying code that works with the superclass.
- Reusability: Subclasses become truly interchangeable with the superclass.

6.3 实例

反例：

```
class Rectangle {
    private int width;
    private int height;

    // ...getters and setters

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

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

会出现以下问题：

```
Rectangle rect = new Square();
rect.setWidth(5);
rect.setHeight(10);
// You'd expect area = 50, but a Square breaks this assumption!
```

正例：

```
interface Shape {
    public double getArea();
}

class Rectangle implements Shape {
   // ...
}

class Square implements Shape {
   // ...
}
```

7 Composite/Aggregate Reuse Principle 组合/聚合复用原则

7.1 定义

尽量使用对象组合/聚合，而不是继承关系达到软件复用的目的。
Composing objects into larger, more complex structures, favoring composition over inheritance whenever possible.

黑箱复用：对类以外的对象无法获取到实现细节。
聚合(Composition): A "has-a" relationship where one class contains instances of other classes as members. Strong ownership—the child objects usually can't exist independently of the parent.
组合(Aggregation): A "contains-a" relationship. The child object can exist independently of the parent object.

白箱复用：把所有的实现细节暴露给子类。
继承Inheritance: A "is-a" relationship.

7.2 优势

- 可扩展性(Flexibility): You can modify the behavior of the composite object by changing its component objects at runtime.
- Code Reuse: Maximizes reuse by using existing components rather than creating new subclasses through inheritance.
- Open/Closed Principle (OCP): You can introduce new functionality through new component objects rather than modifying existing classes.

7.3 实例

```
class Employee {
    private String name;
    // ... other employee details
}

class Department {
    private String name;
    private List<Employee> employees;

    public void addEmployee(Employee employee) {
        employees.add(employee);
    }

    // ... other methods
}

class Company {
    private String name;
    private List<Department> departments;

    public void addDepartment(Department department) {
        departments.add(department);
    }

    // ... other methods
}
```

Composition: A `Department` has a list of `Employee` objects, and a `Company` has a list of `Departments`.

Flexibility: You could calculate the overall company salary by iterating through departments and their employees, rather than needing different subclasses of `Company` to handle different calculation types.