SOLID Principles: Violation vs. Fix

March 29, 2025

1 Single Responsibility Principle (SRP)

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

Violation

In the violation example, the `Report` class was responsible for:

- Storing report content
- Printing the report
- Saving the report to a file

This combines multiple responsibilities into a single class, making it harder to maintain and test.

Fix

The responsibilities were separated into distinct classes:

- `Report` stores the content
- `ReportPrinter` handles printing
- `ReportSaver` handles saving to file

This adheres to SRP by ensuring each class has only one reason to change.

2 Open/Closed Principle (OCP)

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

Violation

The `AreaCalculator` class checked the type of `Shape` using if-else statements to calculate the area. To add a new shape, the class had to be modified directly.

Fix

The `Shape` type was turned into an interface with a `calculateArea()` method. Each shape (e.g., `Circle`, `Square`) implements this method. Now, new shapes can be added without modifying the `AreaCalculator` class.

3 Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

Violation

An `Ostrich` class extended a `Bird` class but threw an exception when `fly()` was called, violating expectations of substitutability.

Fix

The hierarchy was redesigned:

- `Bird` became a general interface
- `FlyingBird` extended `Bird` and added a `fly()` method
- `Sparrow` implements `FlyingBird`
- `Ostrich` implements `Bird` only (no `fly()`)

This ensures only birds that can fly implement the `fly()` method.

4 Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they do not use.

Violation

A `Worker` interface defined both `work()` and `eat()`. `RobotWorker` implemented `Worker` but had no need for `eat()`, resulting in a meaningless implementation.

Fix

The interface was split:

- `Workable` defines `work()`

- `Eatable` defines `eat()`

Now, `HumanWorker` implements both, while `RobotWorker` implements only `Workable`, avoiding unnecessary methods.

5 Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions.

Violation

The `App` class directly instantiated and depended on a concrete `MySQLDatabase` class.

Fix

A `Database` interface was introduced. `MySQLDatabase` implements this interface, and `App` depends on the `Database` abstraction. This allows swapping databases without modifying the `App` class.