MAY 3, 2023 / #DESIGN PATTERNS

What is SOLID? Principles for Better Software Design

Ashutosh Krishna

The SOLID principles are a set of guidelines for writing high-quality, maintainable, and scalable software.

They were introduced by Robert C. Martin in his 2000 paper "Design Principles and Design Patterns" to help developers write software that is easy to understand, modify, and extend.

These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym.

The SOLID acronym stands for:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

These principles provide a way for developers to organize their code and create software that is flexible, easy to change, and testable. Applying

create software that is flexible, easy to change, and testable. Applying SOLID principles can lead to code that is more modular, maintainable, and extensible, and it can make it easier for developers to work collaboratively on a codebase.

In this tutorial, we will explore each of the SOLID principles in detail, explain why they are important, and provide examples of how you can apply them in practice. By the end of this tutorial, you should have a good understanding of the SOLID principles and how to apply them to your software development projects.

What is the Single Responsibility Principle?

The Single Responsibility Principle (SRP) states that **a class should have only one reason to change**, or in other words, **it should have only one responsibility**. This means that a class should have only one job to do, and it should do it well.

If a class has too many responsibilities, it can become hard to understand, maintain, and modify. Changes to one responsibility can inadvertently affect another responsibility, leading to unintended consequences and bugs. By following SRP, we can create code that is more modular, easier to understand, and less prone to errors.

Let's take an example that violates the SRP:

```
class Marker {
    String name;
    String color;
    int price;

    public Marker(String name, String color, int price) {
        this.name = name;
        this.color = color;
        this.price = price;
    }
}
```

The above code defines a simple `Marker` class having three instance variables – `name`, `color` and `price`.

```
class Invoice {
    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        return marker.price * this.quantity;
    }

    public void printInvoice() {
        // printing implementation
    }

    public void saveToDb() {
        // save to database implementation
```

```
    }
}
```

The above `Invoice` class violates the SRP because it has multiple responsibilities – it is responsible for calculating the total amount, printing the invoice, and saving the invoice to the database. As a result, if the calculation logic changes, such as the addition of taxes, the `calculateTotal()` method would require modification. Similarly, if the printing or database-saving implementation changes at any point, the class would need to be changed.

There are several reasons for the class to be modified, which could lead to increased maintenance costs and complexity.

Here's how you can modify the code to follow the SRP:

```
class Invoice {
    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        return marker.price * this.quantity;
    }
}
```

```
class InvoiceDao {
    private Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDb() {
        // save to database implementation
    }
}
```

```
class InvoicePrinter {
    private Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void printInvoice() {
        // printing implementation
    }
}
```

In this refactored example, we have split the responsibilities of the `Invoice` class into three separate classes: `Invoice`, `InvoiceDao`, and `InvoicePrinter`.

The `Invoice` class is responsible only for calculating the total amount, and

the printing and saving responsibilities have been delegated to separate classes. This makes the code more modular, easier to understand, and less prone to errors.

What is the Open-Closed Principle?

The Open-Closed Principle (OCP) states that **software entities (classes, modules, functions, and so on) should be open for extension but closed for modification**. This means that the behavior of a software entity can be extended without modifying its source code.

The OCP is essential because it promotes software extensibility and maintainability. By allowing software entities to be extended without modification, developers can add new functionality without the risk of breaking existing code. This results in code that is easier to maintain, extend, and reuse.

Let's take the previous example again.

```
class InvoiceDao {
    private Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDb() {
        // save to database implementation
    }
}
```

The `InvoiceDao` class has a single responsibility of saving the invoice to the database. But, suppose there's a new requirement to save the invoice to a file as well. One way to implement this requirement would be to modify the existing `InvoiceDao` class by adding a `saveToFile()` method. But this violates the Open-Closed Principle because it modifies the existing code that has already been tested and is live in production.

To follow the OCP, a better solution would be to create an `InvoiceDao` interface and implement it separately for database and file saving as shown below:

```
interface InvoiceDao {
    public void save(Invoice invoice);
}

class DatabaseInvoiceDao implements InvoiceDao {
    @Override
    public void save(Invoice invoice) {
        // save to database implementation
    }
}

class FileInvoiceDao implements InvoiceDao {
    @Override
    public void save(Invoice invoice) {
        // save to file implementation
    }
}
```

This way, if there's a new requirement to save the invoice to another data store, you can implement a new `InvoiceDao` implementation without modifying the existing code. Now the `InvoiceDao` interface is open for extension and closed for modification, which follows the OCP.

What is the Liskov Substitution Principle?

The Liskov Substitution Principle (LSP) states that **any instance of a derived class should be substitutable for an instance of its base class without affecting the correctness of the program.**

In other words, a derived class should behave like its base class in all contexts. In more simple terms, if class A is a subtype of class B, you should be able to replace B with A without breaking the behavior of your program.

The importance of the LSP lies in its ability to ensure that the behavior of a program remains consistent and predictable when substituting objects of different classes. Violating the LSP can lead to unexpected behavior, bugs, and maintainability issues.

Let's take an example.

```
interface Bike {
    void turnOnEngine();

    void accelerate();
}
```

In the given example, the interface `Bike` has two methods, `turnOnEngine()` and `accelerate()`. Two classes implement this interface, `Motorbike` and `Bicycle`.

```
class Motorbike implements Bike {

    boolean isEngineOn;
    int speed;

    @Override
    public void turnOnEngine() {
        isEngineOn = true;
    }

    @Override
    public void accelerate() {
        speed += 5;
    }
}
```

`Motorbike` correctly implements the `turnOnEngine()` method, as it sets the `isEngineOn` boolean to true. It also correctly implements the `accelerate()` method by increasing the `speed` by 5.

```
class Bicycle implements Bike {

    boolean isEngineOn;
```

```
    int speed;

    @Override
    public void turnOnEngine() {
        throw new AssertionError("There is no engine!");
    }

    @Override
    public void accelerate() {
        speed += 5;
    }
}
```

However, the `Bicycle` class throws an `AssertionError` in the `turnOnEngine()` method because it has no engine. This means that an instance of `Bicycle` cannot be substituted for an instance of `Bike` without breaking the behavior of the program.

In other words, if the `Bicycle` class is considered a subtype of the `Bike` interface, then according to the LSP, any instance of `Bike` should be replaceable with an instance of `Bicycle` without altering the correctness of the program.

But in this case, it's not true because `Bicycle` throws an `AssertionError` while trying to turn on the engine. Therefore, the code violates the LSP.

What is the Interface Segregation Principle?

The Interface Segregation Principle (ISP) focuses on designing interfaces that are specific to their client's needs. It states that no client should be forced to depend on methods it does not use.

The principle suggests that **instead of creating a large interface that covers all the possible methods, it's better to create smaller, more focused interfaces for specific use cases**. This approach results in interfaces that are more cohesive and less coupled.

Consider a `Vehicle` interface as below:

```
interface Vehicle {
    void startEngine();
    void stopEngine();
    void drive();
    void fly();
}
```

And then you have a class called `Car` that implements the `Vehicle` interface:

```
class Car implements Vehicle {

    @Override
    public void startEngine() {
        // implementation
    }

    @Override
```

```
    public void stopEngine() {
        // implementation
    }

    @Override
    public void drive() {
        // implementation
    }

    @Override
    public void fly() {
        throw new UnsupportedOperationException("This vehicle cannot fly.");
    }
}
```

In this example, the `Vehicle` interface has too many methods. The `Car` class is forced to implement all of them, even though they cannot fly. This violates the ISP because the `Vehicle` interface is not properly segregated into smaller interfaces based on related functionality.

Let's understand how you can follow the ISP here. Suppose you refactor the `Vehicle` interface into smaller, more focused interfaces:

```
interface Drivable {
    void startEngine();
    void stopEngine();
    void drive();
}

interface Flyable {
    void fly();
}
```

Now, you can have a class called `Car` that only implements the `Drivable` interface:

```
class Car implements Drivable {

    @Override
    public void startEngine() {
        // implementation
    }

    @Override
    public void stopEngine() {
        // implementation
    }

    @Override
    public void drive() {
        // implementation
    }
}
```

And, thanks to interface segregation, you can have another class called `Airplane` that implements both the `Drivable` and `Flyable` interfaces:

```
class Airplane implements Drivable, Flyable {

    @Override
    public void startEngine() {
        // implementation
    }

    @Override
```

```
    public void stopEngine() {
        // implementation
    }

    @Override
    public void drive() {
        // implementation
    }

    @Override
    public void fly() {
        // implementation
    }
}
```

In this example, you have properly segregated the `Vehicle` interface into smaller interfaces based on related functionality. This adheres to the ISP and makes your code more flexible and maintainable.

What is the Dependency Inversion Principle?

The Dependency Inversion Principle (DIP) states that **high-level modules should not depend on low-level modules, but both should depend on abstractions**. Abstractions should not depend on details – details should depend on abstractions.

This principle aims to reduce coupling between modules, increase modularity, and make the code easier to maintain, test, and extend.

For example, consider a scenario where you have a class that needs to use an instance of another class. In the traditional approach, the first class would directly create an instance of the second class, leading to a tight coupling between them. This makes it difficult to change the implementation of the second class or to test the first class independently.

But if you apply the DIP, the first class would depend on an abstraction of the second class instead of the implementation. This would make it possible to easily change the implementation and test the first class independently.

Here is an example that violates the DIP:

```
class WeatherTracker {
    private String currentConditions;
    private Emailer emailer;

    public WeatherTracker() {
        this.emailer = new Emailer();
    }

    public void setCurrentConditions(String weatherDescription) {
        this.currentConditions = weatherDescription;
        if (weatherDescription == "rainy") {
            emailer.sendEmail("It is rainy");
        }
    }
}

class Emailer {
    public void sendEmail(String message) {
        System.out.println("Email sent: " + message);
    }
}
```

In this example, the `WeatherTracker` class directly creates an instance of the `Emailer` class, making it tightly coupled to the implementation. This makes it difficult to change the implementation of the `Emailer` class or to test the `WeatherTracker` class independently.

Here is an example of how to apply the DIP to the above code:

```
interface Notifier {
    public void alertWeatherConditions(String weatherDescription);
}

class WeatherTracker {
    private String currentConditions;
    private Notifier notifier;

    public WeatherTracker(Notifier notifier) {
        this.notifier = notifier;
    }

    public void setCurrentConditions(String weatherDescription) {
        this.currentConditions = weatherDescription;
        if (weatherDescription == "rainy") {
            notifier.alertWeatherConditions("It is rainy");
        }
    }
}

class Emailer implements Notifier {
    public void alertWeatherConditions(String weatherDescription) {
        System.out.println("Email sent: " + weatherDescription);
    }
}

class SMS implements Notifier {
    public void alertWeatherConditions(String weatherDescription) {
        System.out.println("SMS sent: " + weatherDescription);
    }
}
```

In this example, we created a `Notifier` interface that defines the `alertWeatherConditions` method. The `WeatherTracker` class now depends on this interface instead of the `Emailer` class, making it possible to easily change the implementation and test the `WeatherTracker` class independently.

We also created two implementations of the `Notifier` interface, `Emailer`, and `SMS`, to demonstrate how you can change the implementation of the `WeatherTracker` class without affecting its behavior.

Conclusion

In this article, you learned about the SOLID principles which are a very important part of general Design Principles.

By applying these principles in your software development projects, you can create code that is easier to maintain, extend, and modify, leading to more robust, flexible, and reusable software. This will also lead to better collaboration among team members, as the code becomes more modular and easier to work with.

> *For more such tutorials, you can follow my personal blog.*

Ashutosh Krishna

Hello! I am Ashutosh and I enjoy creating things that live on the internet. I was first introduced to programming in my freshman year and since then, I started developing Web projects. I am currently working at Thoughtworks India as an Application Developer.

If you read this far, thank the author to show them you care. Say Thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started