SOLID Principles

1. Single Responsibility Principle (SRP)

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

This means every class should have only one responsibility or function in the system, which makes it focused, modular, and easier to maintain. If a class is handling multiple responsibilities, changes to one responsibility can inadvertently affect others, making the code fragile and difficult to extend.

Example: Marker and Invoice System

Let's look at how we can apply SRP using a simple example of a `Marker` and an `Invoice` system.

Before Applying the Single Responsibility Principle

Here's an example where the `Invoice` class handles multiple responsibilities:

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

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

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

```
        // Business logic: Calculates the total price
        public int calculateTotal() {
            return marker.price * quantity;
        }

        // Responsibility 1: Printing the invoice
        public void printInvoice() {
            System.out.println("Invoice: " + quantity + " markers at $" + marker.price + " each. Total: $" + calc
ulateTotal());
        }

        // Responsibility 2: Saving the invoice to the database
        public void saveToDB() {
            System.out.println("Saving invoice to database...");
        }
}
```

Problems

In this design:

- The `Invoice` class handles **three responsibilities**: calculating the total price, printing the invoice, and saving it to the database.
- Any change to one responsibility (e.g., saving data in a new format) might break unrelated functionality, making the class harder to test, debug, and maintain.
- Violates SRP because the class has **multiple reasons to change**.

After Applying the Single Responsibility Principle

We refactor the code by separating responsibilities into different classes. Each class now has a single purpose.

```
// Marker class remains the same
class Marker {
    String name;
    String color;
    int year;
    int price;

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

```
// Invoice class: Handles only business logic (calculateTotal)
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 * quantity;
    }

    public Marker getMarker() {
        return marker;
    }

    public int getQuantity() {
        return quantity;
    }
}

// InvoicePrinter class: Handles printing functionality
class InvoicePrinter {
    private Invoice invoice;

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

    public void printInvoice() {
        System.out.println("Invoice: " + invoice.getQuantity() + " markers at $" +
                invoice.getMarker().price + " each. Total: $" + invoice.calculateTotal());
    }
}

// InvoiceDao class: Handles database saving functionality
class InvoiceDao {
    private Invoice invoice;

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

    public void saveToDB() {
        System.out.println("Saving invoice to database...");
    }
}
```

Main Method to Test

```
public class Main {
    public static void main(String[] args) {
        Marker marker = new Marker("Blue Marker", "Blue", 2023, 10);
        Invoice invoice = new Invoice(marker, 5);

        // Printing responsibility
        InvoicePrinter printer = new InvoicePrinter(invoice);
        printer.printInvoice();

        // Database responsibility
        InvoiceDao dao = new InvoiceDao(invoice);
        dao.saveToDB();
    }
}
```

Explanation of Refactored Code

1. **Invoice Class**:
    - Now focused solely on the business logic of calculating the total price. It has no knowledge of how invoices are printed or saved.
2. **InvoicePrinter Class**:
    - Handles the responsibility of printing the invoice. If printing functionality changes (e.g., PDF generation or formatting), it affects only this class.
3. **InvoiceDao Class**:
    - Handles the responsibility of saving the invoice to the database. If the storage mechanism changes (e.g., saving to a file or cloud), it affects only this class.

Benefits of Applying SRP

1. **Separation of Concerns**:
    - Each class now has a single, clear responsibility, making the code modular and more understandable.
2. **Maintainability**:
    - Changes in one responsibility (e.g., database saving) don't affect other parts of the system.
3. **Testability**:
    - Each responsibility can be tested in isolation. For example, you can test the `Invoice` class without worrying about printing or database logic.
4. **Scalability**:

- Adding new responsibilities (e.g., sending the invoice via email) is straightforward because it can be implemented as a separate class.

Real-World Analogy

Imagine a chef in a restaurant who is responsible for cooking, taking orders, and cleaning the tables. This is inefficient and prone to errors. Instead, you assign separate responsibilities: the chef focuses on cooking, waiters take orders, and cleaners handle the tables. Each role has a single responsibility, ensuring smoother operations. Similarly, in code, classes with focused responsibilities result in a cleaner and more maintainable system.

2. Open-Closed Principle (OCP)

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

The **Open/Closed Principle (OCP)** is another fundamental design principle in object-oriented programming. It states that:

> A class should be open for extension but closed for modification.

This means that we should be able to add new functionality to a class without altering its existing code. This helps prevent introducing bugs into already working code and makes the system more maintainable and extensible.

To better understand this principle, let's use a **real-world example of markers and invoices**.

Before Applying the Open/Closed Principle

Here's a scenario where the principle is violated. Imagine we want to calculate the total cost of an invoice, but we now have two types of markers: regular markers and permanent markers. Permanent markers have an additional cost due to special ink.

Let's look at the code:

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

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

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

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

    public int calculateTotal(String markerType) {
        if (markerType.equals("Regular")) {
            return marker.price * quantity;
        } else if (markerType.equals("Permanent")) {
            return (marker.price + 5) * quantity; // Extra cost for permanent markers
        }
        return 0;
    }
}
```

Problem with the Above Code

1. Every time a new marker type is added, we must modify the `calculateTotal()` method.
2. This violates OCP because the `Invoice` class is not **closed to modification**. Adding new marker types requires altering the logic in the `calculateTotal()` method, increasing the risk of breaking existing functionality.

After Applying the Open/Closed Principle

To fix this, we can refactor the code to follow OCP by introducing **abstraction**. We'll create a `Marker` base class and extend it for different marker types. Each subclass will handle its pricing logic, allowing us to add new marker types without modifying existing code.

Here's the refactored code:

```
// Base class: Marker
abstract class Marker {
    String name;
    String color;
    int year;

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

    public abstract int getPrice();
}

// RegularMarker class
class RegularMarker extends Marker {
    private int price;

    public RegularMarker(String name, String color, int year, int price) {
        super(name, color, year);
        this.price = price;
    }

    @Override
    public int getPrice() {
        return price;
    }
}

// PermanentMarker class
class PermanentMarker extends Marker {
    private int price;

    public PermanentMarker(String name, String color, int year, int price) {
        super(name, color, year);
        this.price = price;
    }

    @Override
    public int getPrice() {
        return price + 5; // Additional cost for permanent markers
    }
}

// Invoice class
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.getPrice() * quantity;
    }
}
```

Explanation of the Refactored Code

1. **Base Class (`Marker`):**

   The `Marker` class is now abstract and defines the common properties of all markers (`name` , `color` , `year`). It also includes an abstract method `getPrice()` , which forces subclasses to implement their own pricing logic.

2. **Subclasses (`RegularMarker` and `PermanentMarker`):**

   These classes inherit from `Marker` and implement the `getPrice()` method. A `RegularMarker` simply returns its price, while a `PermanentMarker` adds an extra cost to its base price.

3. **Invoice Class**:

   The `Invoice` class now interacts with the `Marker` abstraction, not specific marker types. It calls the `getPrice()` method, which is dynamically resolved based on the type of marker passed to it. This makes it **open for extension** (adding new marker types) and **closed for modification** (no changes to the `Invoice` class).

Adding a New Marker Type

Suppose the business introduces a **HighlighterMarker** that costs 10% more than its base price. You can simply create a new class without modifying the existing ones:

```
class HighlighterMarker extends Marker {
    private int price;

    public HighlighterMarker(String name, String color, int year, int price) {
        super(name, color, year);
        this.price = price;
    }

    @Override
    public int getPrice() {
        return (int) (price * 1.1); // 10% extra cost
    }
}
```

You can now use this new marker type without making any changes to the `Invoice` class.

Main Method to Test the Code

```
public class Main {
    public static void main(String[] args) {
        Marker regularMarker = new RegularMarker("Whiteboard Marker", "Black", 2023, 10);
        Marker permanentMarker = new PermanentMarker("Permanent Marker", "Red", 2023, 15);
        Marker highlighterMarker = new HighlighterMarker("Highlighter", "Yellow", 2023, 20);

        Invoice regularInvoice = new Invoice(regularMarker, 5);
        Invoice permanentInvoice = new Invoice(permanentMarker, 3);
        Invoice highlighterInvoice = new Invoice(highlighterMarker, 7);
        System.out.println("Regular Invoice Total: " + regularInvoice.calculateTotal());
        System.out.println("Permanent Invoice Total: " + permanentInvoice.calculateTotal());
        System.out.println("Highlighter Invoice Total: " + highlighterInvoice.calculateTotal());
    }
}
```

Advantages of Applying OCP

1. Adding new marker types is now **seamless**. You only need to create a new subclass.
2. The `Invoice` class remains untouched, reducing the risk of introducing bugs in existing functionality.
3. The design is **scalable** and adheres to good object-oriented practices.

This is how the **Open/Closed Principle** ensures that our code is flexible, maintainable, and easy to extend.

3. Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

The **Liskov Substitution Principle (LSP)** is the 'L' in SOLID principles, formulated by Barbara Liskov. It states:

> If S is a subtype of T, then objects of type T can be replaced with objects of type S without altering the correctness of the program.

In simpler terms, it means that derived classes (subclasses) should be able to substitute their base class without breaking the behavior of the program. Subtypes must uphold the behavior expected of the base type, ensuring consistency and reliability.

To understand LSP, let's consider a real-world example of **Markers and Discounts**.

Before Applying the Liskov Substitution Principle

Imagine you have a base class `Marker` and a subclass `DiscountedMarker`. The subclass introduces a discount calculation for the marker. However, this implementation violates LSP because the behavior of the subclass breaks the expectations of the base class.

Here's the problematic code:

```
// Base Marker class
class Marker {
    String name;
    int price;

    public Marker(String name, int price) {
        this.name = name;
        this.price = price;
    }
    public int getPrice() {
        return price;
    }
}

// Subclass with discount logic
class DiscountedMarker extends Marker {
    int discount;

    public DiscountedMarker(String name, int price, int discount) {
        super(name, price);
        this.discount = discount;
    }

    @Override
    public int getPrice() {
        return price - discount; // Apply discount
    }
}

// Invoice class
class Invoice {
    public int calculateTotal(Marker marker, int quantity) {
        return marker.getPrice() * quantity;
    }
}
```

Problem with the Above Code

1. The `DiscountedMarker` class changes the behavior of the `getPrice()` method. Instead of returning the base price, it applies a discount.
2. If `DiscountedMarker` is substituted for `Marker` in the `Invoice` class, it could lead to unexpected results because the base class behavior has been overridden improperly.
3. This breaks LSP because the subclass (`DiscountedMarker`) does not fully adhere to the expectations set by the base class (`Marker`).

After Applying the Liskov Substitution Principle

To follow LSP, subclasses should extend the base class without altering its expected behavior. In this case, we can refactor the design to separate the discount calculation into its own class or method, keeping the `Marker` class intact.

Here's the corrected code:

```
// Base Marker class
class Marker {
    String name;
    int price;

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

    public int getPrice() {
        return price;
    }
}

// DiscountCalculator class
class DiscountCalculator {
    private Marker marker;
    private int discount;

    public DiscountCalculator(Marker marker, int discount) {
        this.marker = marker;
        this.discount = discount;
    }

    public int getDiscountedPrice() {
        return marker.getPrice() - discount;
    }
}

// Invoice class
class Invoice {
    public int calculateTotal(Marker marker, int quantity) {
        return marker.getPrice() * quantity;
    }

    public int calculateTotalWithDiscount(DiscountCalculator discountCalculator, int quantity) {
        return discountCalculator.getDiscountedPrice() * quantity;
    }
}
```

Explanation of the Refactored Code

1. `Marker` **Class Behavior is Preserved**:

   The `Marker` class now retains its original behavior (`getPrice()` simply returns the base price). No subclass alters this behavior.

2. **Separate Discount Logic**:

   The `DiscountCalculator` class is introduced to handle discount calculations. This ensures that `Marker` does not deviate from its defined behavior, preserving the consistency expected by other parts of the code.

3. **Invoice Class Compatibility**:

   The `Invoice` class can now calculate totals for both regular markers and discounted markers using `DiscountCalculator`. Subtypes do not interfere with the logic for regular markers.

Adding a New Feature

Suppose you want to introduce a new type of discount (e.g., percentage discount). You can now extend the `DiscountCalculator` class or create another calculator class without modifying existing code.

```
class PercentageDiscountCalculator extends DiscountCalculator {
    private int percentage;

    public PercentageDiscountCalculator(Marker marker, int percentage) {
        super(marker, 0); // Pass 0 for base class compatibility
        this.percentage = percentage;
    }

    @Override
    public int getDiscountedPrice() {
        return marker.getPrice() - (marker.getPrice() * percentage / 100);
    }
}
```

Now, this new behavior can be used with `Invoice` without altering its existing functionality.

Main Method to Test the Code

```
public class Main {
    public static void main(String[] args) {
        Marker regularMarker = new Marker("Whiteboard Marker", 10);
        Marker permanentMarker = new Marker("Permanent Marker", 20);

        DiscountCalculator discountCalculator = new DiscountCalculator(regularMarker, 2);
        PercentageDiscountCalculator percentageDiscountCalculator = new PercentageDiscountCalculator(per

        Invoice invoice = new Invoice();

        System.out.println("Regular Marker Total: " + invoice.calculateTotal(regularMarker, 5));
        System.out.println("Discounted Marker Total: " + invoice.calculateTotalWithDiscount(discountCalculator
        System.out.println("Percentage Discount Marker Total: " + invoice.calculateTotalWithDiscount(percenta
    }
}
```

Benefits of Applying LSP

1. **Consistency**: The `Marker` class and its subtypes behave as expected, ensuring no surprises when substituting subclasses.
2. **Extensibility**: New functionality like discounts can be added without altering existing classes.
3. **Robustness**: The program avoids bugs caused by unexpected behavior changes in subclasses.

By adhering to the Liskov Substitution Principle, we ensure that the program remains consistent, extensible, and maintainable while avoiding violations of expected behavior.

4. Interface Segregation Principle (ISP)

The **Interface Segregation Principle (ISP)** is the 'I' in the SOLID principles. It states:

> A class should not be forced to implement interfaces it does not use.

The idea behind ISP is that large, monolithic interfaces should be divided into smaller, more specific interfaces so that implementing classes only need to be concerned with the methods that are relevant to them. This reduces unnecessary dependencies and makes the code easier to maintain and extend.

Before Applying Interface Segregation Principle

Let's consider an example of a `Printer` interface for different types of printing devices.

```
// Large monolithic interface
interface Printer {
    void printDocument(String content);
    void scanDocument(String content);
    void faxDocument(String content);
    void stapleDocument(String content);
}

// Implementing class
class BasicPrinter implements Printer {
    @Override
    public void printDocument(String content) {
        System.out.println("Printing: " + content);
    }

    @Override
    public void scanDocument(String content) {
        throw new UnsupportedOperationException("Scan not supported.");
    }

    @Override
    public void faxDocument(String content) {
        throw new UnsupportedOperationException("Fax not supported.");
    }
    @Override
    public void stapleDocument(String content) {
        throw new UnsupportedOperationException("Staple not supported.");
    }
}
```

Problem

The `BasicPrinter` class is forced to implement methods like `scanDocument`, `faxDocument`, and `stapleDocument`, which are irrelevant to its functionality. This violates the Interface Segregation Principle because the interface is too broad, and the class is burdened with implementing methods it does not use.

After Applying Interface Segregation Principle

To apply ISP, we divide the large `Printer` interface into smaller, more specific interfaces. Each class will implement only the interfaces it actually needs.

```
// Smaller, specific interfaces
interface Printable {
    void printDocument(String content);
}

interface Scannable {
    void scanDocument(String content);
}

interface Faxable {
    void faxDocument(String content);
}

interface Stapleable {
    void stapleDocument(String content);
}

// Implementing class for a basic printer
class BasicPrinter implements Printable {
    @Override
    public void printDocument(String content) {
        System.out.println("Printing: " + content);
    }
}

// Implementing class for an advanced printer
class AdvancedPrinter implements Printable, Scannable, Faxable {
    @Override
    public void printDocument(String content) {
        System.out.println("Printing: " + content);
    }

    @Override
    public void scanDocument(String content) {
        System.out.println("Scanning: " + content);
    }

    @Override
    public void faxDocument(String content) {
        System.out.println("Faxing: " + content);
    }
}
```

Explanation of the Refactored Code

1. **Small, Specific Interfaces**: Instead of a single `Printer` interface with all methods, we now have multiple interfaces like `Printable`, `Scannable`, `Faxable`, and `Stapleable`. Each interface represents a specific capability.
2. **Implement Only What's Needed**:
    - The `BasicPrinter` class implements only the `Printable` interface because it only needs the ability to print.
    - The `AdvancedPrinter` class implements `Printable`, `Scannable`, and `Faxable` because it supports printing, scanning, and faxing.
3. **No Unused Methods**: Classes are no longer forced to implement irrelevant methods, eliminating the need for `UnsupportedOperationException`.

Benefits of ISP

- **Reduced Complexity**: Classes remain simpler as they only deal with the methods they actually use.
- **Improved Maintainability**: Smaller interfaces are easier to understand and modify without affecting unrelated parts of the system.
- **Extensibility**: Adding new functionality is easier since it does not require changes to existing interfaces or classes.

Main Method to Test the Code

Here's how you can use the refactored classes:

```
public class Main {
    public static void main(String[] args) {
        Printable basicPrinter = new BasicPrinter();
        basicPrinter.printDocument("Hello, Basic Printer!");

        AdvancedPrinter advancedPrinter = new AdvancedPrinter();
        advancedPrinter.printDocument("Hello, Advanced Printer!");
        advancedPrinter.scanDocument("Scan this document.");
        advancedPrinter.faxDocument("Fax this document.");
    }
}
```

Real-World Analogy

Think of a restaurant. A "Waiter" interface might include methods like `serveFood()`, `takeOrder()`, and `cleanTable()`. If a robot waiter is only responsible for serving food, it should not be forced to implement methods for taking orders or cleaning tables. Instead, you can split the responsibilities into smaller, more focused interfaces (`FoodServer`, `OrderTaker`, `TableCleaner`) and let the robot implement only the `FoodServer` interface.

By following the Interface Segregation Principle, we design systems that are more modular, reusable, and easier to maintain, aligning perfectly with the goals of clean and efficient software architecture.

5. Dependency Inversion Principle (DIP)

The **Dependency Inversion Principle (DIP)** is the 'D' in SOLID principles and states:

> High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

This principle emphasizes decoupling high-level (business logic) and low-level (implementation) modules by introducing an abstraction layer. By doing this, changes in the low-level modules don't affect high-level modules, making the code more flexible and easier to extend.

Before Applying the Dependency Inversion Principle

Let's look at an example where a high-level module is tightly coupled to a low-level module.

```
// Low-level module
class Keyboard {
    public String getInput() {
        return "Keyboard Input";
    }
}

// Low-level module
class Monitor {
    public void display(String content) {
        System.out.println("Displaying: " + content);
    }
}

// High-level module
class Computer {
    private Keyboard keyboard;
    private Monitor monitor;

    public Computer() {
        this.keyboard = new Keyboard(); // Tightly coupled
        this.monitor = new Monitor();  // Tightly coupled
    }

    public void operate() {
        String input = keyboard.getInput();
        monitor.display(input);
    }
}
```

Problem

In this design:

- The `Computer` class (high-level module) directly depends on the `Keyboard` and `Monitor` classes (low-level modules).
- Any change in the `Keyboard` or `Monitor` implementation (e.g., introducing a wireless keyboard or touchscreen monitor) requires changes in the `Computer` class.
- The code is hard to test and extend due to tight coupling.

After Applying the Dependency Inversion Principle

We introduce abstractions (interfaces) so that the high-level module depends on abstractions instead of concrete implementations.

```
// Abstraction for input devices
interface InputDevice {
    String getInput();
}

// Abstraction for output devices
interface OutputDevice {
    void display(String content);
}

// Low-level module: Keyboard implements InputDevice
class Keyboard implements InputDevice {
    @Override
    public String getInput() {
        return "Keyboard Input";
    }
}

// Low-level module: Monitor implements OutputDevice
```

```
class Monitor implements OutputDevice {
    @Override
    public void display(String content) {
        System.out.println("Displaying: " + content);
    }
}

// High-level module
class Computer {
    private InputDevice inputDevice;
    private OutputDevice outputDevice;

    // Depend on abstractions
    public Computer(InputDevice inputDevice, OutputDevice outputDevice) {
        this.inputDevice = inputDevice;
        this.outputDevice = outputDevice;
    }

    public void operate() {
        String input = inputDevice.getInput();
        outputDevice.display(input);
    }
}
```

Main Method to Test

```
public class Main {
    public static void main(String[] args) {
        // Inject dependencies
        InputDevice keyboard = new Keyboard();
        OutputDevice monitor = new Monitor();

        // High-level module depends on abstractions
        Computer computer = new Computer(keyboard, monitor);
        computer.operate();
    }
}
```

Explanation of the Refactored Code

1. **Abstractions**:
    - Introduced `InputDevice` and `OutputDevice` interfaces to act as contracts for input and output devices.
2. **High-Level Module**:
    - The `Computer` class depends on the `InputDevice` and `OutputDevice` interfaces, not their concrete implementations (`Keyboard` or `Monitor`).
3. **Low-Level Modules**:

    - The `Keyboard` and `Monitor` classes implement the `InputDevice` and `OutputDevice` interfaces, respectively.

4. **Dependency Injection**:
    - Dependencies (`Keyboard` and `Monitor`) are injected into the `Computer` class through its constructor, making it flexible to work with any implementation of `InputDevice` and `OutputDevice`.

Benefits of Applying DIP

- **Decoupling**: The `Computer` class no longer depends on specific implementations, making it easier to replace or modify the `Keyboard` and `Monitor` without changing the `Computer` class.
- **Testability**: Abstractions allow us to mock dependencies for unit testing the `Computer` class.
- **Extensibility**: Adding new input or output devices (e.g., touchscreen or voice input) requires minimal changes.

Real-World Analogy

Imagine a power socket and a device (like a laptop). Instead of a socket that directly provides power in only one format, we use a universal adapter (abstraction) to allow devices to work with various socket types (details). The devices only depend on the adapter, and the adapter ensures compatibility with specific power sockets.

In software, abstractions play the role of the universal adapter, ensuring that high-level modules work independently of low-level module details.