Principios S.O.L.I.D.

Los principios S.O.L.I.D. son un conjunto de cinco principios de diseño orientados a objetos que ayudan a los desarrolladores a crear software que sea más fácil de mantener, entender y extender. Estos principios fueron introducidos por Robert C. Martin, también conocido como "Uncle Bob". A continuación se presenta una explicación detallada de cada uno de estos principios:

1. Single Responsibility Principle (SRP)

Principio de Responsabilidad Única

Definición: Una clase debe tener una, y solo una, razón para cambiar. Esto significa que cada clase debe tener una única responsabilidad o propósito.

Explicación: El principio de responsabilidad única se enfoca en separar las preocupaciones en el diseño del software. Cada clase debe manejar una única parte de la funcionalidad del sistema, y todas las operaciones de esa clase deben estar alineadas con esa responsabilidad. Esto facilita el mantenimiento del código, ya que los cambios en una parte del sistema afectan solo a la clase que se encarga de esa parte específica.

Ejemplo: Una clase que maneja la lógica de negocios de una factura no debe encargarse también de la impresión de la misma. Estas responsabilidades deben estar separadas en clases distintas.

2. Open/Closed Principle (OCP)

Principio Abierto/Cerrado

Definición: Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para su extensión, pero cerradas para su modificación.

Explicación: Este principio sugiere que el diseño de las clases debe permitir la extensión del comportamiento de las mismas sin necesidad de modificar su código fuente. Esto se logra utilizando la herencia y la composición, permitiendo que las nuevas funcionalidades se añadan mediante nuevas clases o métodos que amplíen la funcionalidad existente sin alterar el código ya escrito.

Ejemplo: En lugar de modificar una clase existente para añadir una nueva funcionalidad, se crea una nueva subclase que extiende la funcionalidad de la clase original.

3. Liskov Substitution Principle (LSP)

Principio de Sustitución de Liskov

Definición: Las subclases deben ser sustituibles por sus clases base sin alterar el correcto funcionamiento del programa.

Explicación: Este principio, formulado por Barbara Liskov, establece que si una clase S es una subclase de una clase T, los objetos de la clase T deben poder ser reemplazados por objetos de la clase S sin afectar el comportamiento del programa. Esto garantiza que una subclase puede usarse en cualquier lugar donde se espere un objeto de la clase base, asegurando que el código que usa la clase base no necesita saber de la existencia de la subclase.

Ejemplo: Una subclase PájaroQueNoVuela debe poder ser utilizada en lugar de su clase base Pájaro sin que el comportamiento del programa sea incorrecto o inesperado.

4. Interface Segregation Principle (ISP)

Principio de Segregación de Interfaces

Definición: Los clientes no deben estar obligados a depender de interfaces que no utilizan.

Explicación: Este principio establece que es mejor tener varias interfaces específicas y pequeñas que una sola interfaz general y grande. Los clientes de una interfaz deben depender solo de los métodos que realmente utilizan. Esto evita que los clientes estén obligados a implementar métodos que no necesitan y promueve un diseño de interfaces más limpio y enfocado.

Ejemplo: En lugar de tener una única interfaz Trabajador con métodos trabajar y comer, se pueden crear dos interfaces separadas Trabajable y Comible, para que las clases que implementen estas interfaces solo tengan los métodos que realmente necesitan.

5. Dependency Inversion Principle (DIP)

Principio de Inversión de Dependencias

Definición: Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

Explicación: Este principio sugiere que las dependencias en un programa deben invertirse. En lugar de que las clases de alto nivel dependan de clases de bajo nivel, ambas deben depender de interfaces o abstracciones. Esto reduce el acoplamiento entre las clases y mejora la flexibilidad y la mantenibilidad del código.

Ejemplo: En lugar de que una clase Interruptor dependa directamente de una clase Bombilla, ambas deben depender de una interfaz Encendible. Esto permite que el interruptor trabaje con cualquier dispositivo que implemente la interfaz Encendible, como una bombilla, un ventilador, etc.

Implementación

1. Single Responsibility Principle (SRP)

Principio: Una clase debe tener una única responsabilidad y sólo una razón para cambiar.

Explicación: Aquí tenemos una clase Invoice que representa una factura. Las operaciones relacionadas con la impresión de facturas y el almacenamiento de facturas se han delegado a las clases InvoicePrinter y InvoiceRepository, respectivamente.

```java
package solid;

class Invoice {
    private double amount;

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}

class InvoicePrinter {
    public void printInvoice(Invoice invoice) {
        System.out.println("Imprimiendo factura con monto: " + invoice.getAmount());
    }
}

class InvoiceRepository {
    public void saveInvoice(Invoice invoice) {
        System.out.println("Guardando factura con monto: " + invoice.getAmount());
    }
}
```

```java
public class SingleResponsibility {
    public static void main(String[] args) {
        Invoice invoice = new Invoice();
        invoice.setAmount(100);

        InvoicePrinter printer = new InvoicePrinter();
        InvoiceRepository repository = new InvoiceRepository();

        printer.printInvoice(invoice);
        repository.saveInvoice(invoice);
    }
}
```

Explicación del Código:

- Invoice se encarga únicamente de manejar la información de la factura.
- InvoicePrinter se encarga de la lógica de impresión.
- InvoiceRepository se encarga de la lógica de almacenamiento.

2. Open/Closed Principle (OCP)

Principio: Las clases deben estar abiertas para la extensión, pero cerradas para la modificación.

Explicación: Aquí tenemos una clase abstracta Shape y dos subclases Circle y Square. Podemos añadir nuevas formas sin modificar el código existente.

```
package solid;

abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Dibujando Círculo");
    }
}

class Square extends Shape {
    @Override
    void draw() {
        System.out.println("Dibujando Cuadrado");
    }
}

class ShapeDrawer {
    public void drawShape(Shape shape) {
        shape.draw();
    }
}


public class OpenClosed {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape square = new Square();

        ShapeDrawer drawer = new ShapeDrawer();
        drawer.drawShape(circle);
        drawer.drawShape(square);
    }
}
```

Explicación del Código:

- Shape es una clase abstracta con el método abstracto draw.

- Circle y Square son subclases que implementan draw.
- ShapeDrawer puede dibujar cualquier forma sin conocer sus detalles internos.

3. Liskov Substitution Principle (LSP)

Principio: Las subclases deben ser sustituibles por sus clases base sin alterar el comportamiento esperado del programa.

Explicación: Aquí tenemos una clase abstracta Bird con el método abstracto move. FlyingBird y NonFlyingBird son subclases que implementan move de manera apropiada.

```java
package solid;

abstract class Bird {
    abstract void move();
}

class FlyingBird extends Bird {
    @Override
    void move() {
        System.out.println("Volando");
    }
}

class NonFlyingBird extends Bird {
    @Override
    void move() {
        System.out.println("Caminando");
    }
}

public class LiskovSubstitution {
    public static void main(String[] args) {
        Bird flyingBird = new FlyingBird();
        Bird nonFlyingBird = new NonFlyingBird();

        flyingBird.move();
        nonFlyingBird.move();
    }
}
```

Explicación del Código:

- Bird es una clase abstracta con el método move.
- FlyingBird implementa move para volar.
- NonFlyingBird implementa move para caminar.
- Podemos usar FlyingBird y NonFlyingBird donde sea que se espere un Bird.

4. Interface Segregation Principle (ISP)

Principio: Los clientes no deben estar forzados a depender de interfaces que no utilizan.

Explicación: Aquí tenemos dos interfaces Workable y Eatable. HumanWorker implementa ambas interfaces, mientras que RobotWorker sólo implementa Workable.

```java
package solid;

interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class HumanWorker implements Workable, Eatable {
    @Override
    public void work() {
        System.out.println("Humano trabajando");
    }

    @Override
    public void eat() {
        System.out.println("Humano comiendo");
    }
}

class RobotWorker implements Workable {
    @Override
    public void work() {
        System.out.println("Robot trabajando");
    }
}

public class InterfaceSegregation {
    public static void main(String[] args) {
        HumanWorker humanWorker = new HumanWorker();
        RobotWorker robotWorker = new RobotWorker();

        humanWorker.work();
        humanWorker.eat();

        robotWorker.work();
    }
}
```

Explicación del Código:

- Workable define el método work.

- Eatable define el método eat.
- HumanWorker implementa ambas interfaces.
- RobotWorker implementa solo Workable.

5. Dependency Inversion Principle (DIP)

Principio: Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

Explicación: Aquí tenemos una interfaz Switchable que es implementada por LightBulb. La clase Switch depende de la abstracción Switchable en lugar de una implementación concreta.

```
package solid;

interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("Bombilla encendida");
    }

    @Override
    public void turnOff() {
        System.out.println("Bombilla apagada");
    }
}

class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    public void operate() {
        device.turnOn();
    }
}
```

```java
public class DependencyInversion {
    public static void main(String[] args) {
        Switchable lightBulb = new LightBulb();
        Switch lightSwitch = new Switch(lightBulb);

        lightSwitch.operate();
    }
}
```

Explicación del Código:

- Switchable define los métodos turnOn y turnOff.
- LightBulb implementa Switchable.
- Switch depende de la interfaz Switchable y puede trabajar con cualquier implementación de Switchable.