¿Qué son los Principios SOLID?

Los Principios SOLID son fundamentales en el desarrollo de software orientado a objetos, proporcionando un marco para crear código más limpio, mantenible y escalable. Cada letra del acrónimo representa un principio clave:

Single Responsibility: Cada componente debe tener una única responsabilidad.

```
==============================================
class User {
    private String name;
    private String email;

    // Getters y setters
}

class UserRepository {
    public void save(User user) {
        // Código para guardar al usuario en la base de datos
    }
}

class EmailService {
    public void sendEmail(String email, String message) {
        // Código para enviar un correo electrónico
    }
}
==============================================
```

Las responsabilidades se dividen en tres clases por aparte: User para representar al usuario, UserRepository para manejar la persistencia y EmailService para manejar el envío de correos electrónicos.

Open/Closed: Los componentes deben estar abiertos para la extensión, pero cerrados para la modificación.

==========

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

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

==========

En este ejemplo, Shape es una clase abstracta y sus subclases Circle y Rectangle implementan el método area. Si necesitamos agregar una nueva forma, podemos crear una nueva subclase sin modificar las clases existentes.

Liskov Substitution: Los objetos de una clase base deben poder ser reemplazados por objetos de una clase derivada sin afectar la funcionalidad.

==========

```
abstract class Bird {
    public abstract void makeSound();
}

class Sparrow extends Bird {
    @Override
    public void makeSound() {
        System.out.println("Chirp chirp");
    }
}

class Ostrich extends Bird {
    @Override
    public void makeSound() {
        System.out.println("Boom boom");
    }
}

public class BirdWatcher {
    public void watchBird(Bird bird) {
        bird.makeSound();    }

    public static void main(String[] args) {
        BirdWatcher watcher = new BirdWatcher();
        Bird sparrow = new Sparrow();
        Bird ostrich = new Ostrich();

        watcher.watchBird(sparrow); // Output: Chirp chirp
        watcher.watchBird(ostrich); // Output: Boom boom
    }
}
```

==========

En este ejemplo, hemos definido una clase abstracta Bird con un método abstracto makeSound(). Tanto Sparrow como Ostrich implementan este método de manera consistente con la clase base. El código que utiliza la clase base Bird puede trabajar con cualquier subclase de Bird sin romper la funcionalidad.

Interface Segregation: No se debe obligar a los consumidores a depender de interfaces que no utilizan.

```
==============================================
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class HumanWorker implements Workable, Eatable {
    @Override
    public void work() {
        // Código para trabajar
    }

    @Override
    public void eat() {
        // Código para comer
    }
}

class RobotWorker implements Workable {
    @Override
    public void work() {
        // Código para trabajar
    }
}
==============================================
```

Las interfaces se dividen en Workable y Eatable, y las clases HumanWorker y RobotWorker implementan solo las interfaces que necesitan.

Dependency Inversion: Se debe depender de abstracciones, no de implementaciones concretas.

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

class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        // Encender bombilla
    }

    @Override
    public void turnOff() {
        // Apagar bombilla
    }
}

class Fan implements Switchable {
    @Override
    public void turnOn() {
        // Encender ventilador
    }

    @Override
    public void turnOff() {
        // Apagar ventilador
    }
}
class Switch {
    private Switchable device;

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

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

La clase Switch depende de la interfaz Switchable en lugar de una implementación concreta. Esto permite que Switch funcione con cualquier dispositivo que implemente la interfaz Switchable, como LightBulb o Fan.

Referencias

Fusiona CL(Enero 10 de 2024) Principios SOLID: *¿Por qué usarlos en tu desarrollo de software?*.https://fusiona.cl/blog/tecnologia/principios-solid-por-que-usarlos-en-tu-desarrollo-de-software/