Actividad

Los Principios SOLID

Nombre de la escuela

Universidad Politécnica de Chiapas

Autores

243763 Martínez Jaimes Rudi Fabricio

Docente

Ali López Zúnun

Fecha

29/07/2025

Lugar

Suchiapa, Chiapas, México

Materia

Programación Orientada a Objetos

1. S – Single Responsibility Principle (SRP)
Una clase debe tener una sola razón para cambiar.

Análisis:
Cada clase debe encargarse de una única responsabilidad dentro del sistema. Si una clase hace más de una cosa (por ejemplo, guardar datos y enviar correos), debe separarse.

Ejemplo en Java:

```
// Clase con responsabilidad única: representar al usuario
public class User {
    private String name;
    private String email;
    // Getters y Setters
}

// Clase con responsabilidad única: guardar el usuario
public class UserRepository {
    public void save(User user) {
        System.out.println("Guardando usuario en base de datos...");
    }
}

// Clase con responsabilidad única: enviar notificación
public class EmailService {
    public void sendWelcomeEmail(User user) {
        System.out.println("Enviando email a: " + user.getEmail());
    }
}
```

2. O – Open/Closed Principle (OCP)
Las clases deben estar abiertas para extensión, pero cerradas para modificación.

Análisis:
Puedes extender el comportamiento de una clase sin modificar su código fuente.

Ejemplo en Java:

```
// Interfaz general
public interface Descuento {
    double aplicar(double precio);
}

// Nuevas clases se agregan sin modificar la lógica base
public class DescuentoPorcentual implements Descuento {
    public double aplicar(double precio) {
        return precio * 0.9; // 10% de descuento
    }
}

public class DescuentoFijo implements Descuento {
    public double aplicar(double precio) {
        return precio - 50; // $50 menos
    }
}

// Clase abierta para extensión, cerrada para modificación
public class CalculadoraDescuento {
    public double calcular(Descuento descuento, double precio) {
        return descuento.aplicar(precio);
    }
}
```

3. L – Liskov Substitution Principle (LSP)
Una clase hija debe poder sustituir a su clase padre sin alterar el comportamiento esperado.

Análisis:
Las subclases deben respetar el contrato de la clase base.

Ejemplo en Java (violación):

```
class Rectangulo {
    protected int ancho, alto;
    public void setAncho(int a) { ancho = a; }
    public void setAlto(int a) { alto = a; }
```

```
    public int getArea() { return ancho * alto; }
}

class Cuadrado extends Rectangulo {
    @Override
    public void setAncho(int a) {
        ancho = alto = a;
    }
    @Override
    public void setAlto(int a) {
        ancho = alto = a;
    }
}
```

Aquí Cuadrado rompe el comportamiento esperado de Rectangulo. No debería heredar de él directamente.

Solución: usar composición en lugar de herencia para Cuadrado.

4. I – Interface Segregation Principle (ISP)
Una clase no debe verse obligada a implementar métodos que no necesita.

Análisis:
Es preferible tener varias interfaces pequeñas que una sola grande.

Ejemplo en Java:

```
// Interfaces específicas
public interface IImpresora {
    void imprimir();
}

public interface IEscaner {
    void escanear();
}

// Implementaciones concretas
public class ImpresoraBasica implements IImpresora {
    public void imprimir() {
        System.out.println("Imprimiendo documento...");
```

```
    }
}

public class Multifuncional implements IImpresora, IEscaner {
    public void imprimir() {
        System.out.println("Impresión multifuncional.");
    }
    public void escanear() {
        System.out.println("Escaneo multifuncional.");
    }
}
```

5. D – Dependency Inversion Principle (DIP)
Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

Análisis:
Depender de interfaces en lugar de implementaciones concretas.

Ejemplo en Java:

```
// Abstracción
public interface Notificador {
    void notificar(String mensaje);
}

// Implementación concreta
public class NotificadorEmail implements Notificador {
    public void notificar(String mensaje) {
        System.out.println("Enviando correo: " + mensaje);
    }
}

// Clase de alto nivel que depende de la abstracción
public class ServicioAlerta {
    private Notificador notificador;

    public ServicioAlerta(Notificador notificador) {
        this.notificador = notificador;
    }
```

```
    public void enviarAlerta(String mensaje) {
        notificador.notificar(mensaje);
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Notificador notificador = new NotificadorEmail();
        ServicioAlerta alerta = new ServicioAlerta(notificador);
        alerta.enviarAlerta("Producto disponible en EquipLink");
    }
}
```

Análisis general:
Los principios SOLID no solo mejoran la calidad del código, sino que hacen que los sistemas sean más:

- Mantenibles
- Extensibles
- Reutilizables
- Testeables

Su objetivo principal es ayudar a escribir código que sea más limpio, comprensible y fácil de mantener. Cada principio tiene una función específica que mejora la estructura del software. Por ejemplo, el principio de responsabilidad única me hizo ver que una clase debe enfocarse en una sola tarea, lo que evita confusión y errores.