DISEÑO DE SISTEMAS

Juan Carlos Molina Lozano

Docente

CONTENIDO

- Introducción
- Objetivos de la Clase
- Propósito del Principio Orientado a Objetos (SOLID)
- Características del Principio Orientado a Objetos (SOLID)
- Ventajas y Desventajas
- Elementos del Principio Orientado a Objetos (SOLID)
- Ejemplos Prácticos

OBJETIVOS DE LA CLASE

- Entender los cinco principios SOLID.
- Analizar su importancia en el diseño de software orientado a objetos.
- Aplicar los principios SOLID mediante ejemplos prácticos de programación.

INTRODUCCIÓN

La creación de software no solo trata de que funcione, sino de que sea fácil de entender, mantener y escalar. A medida que las aplicaciones crecen, un mal diseño puede provocar sistemas frágiles y difíciles de modificar.

Para enfrentar estos desafíos, se proponen buenas prácticas de diseño, entre ellas los principios SOLID, popularizados por Robert C. Martin (Uncle Bob).

Qué problemas han tenido en proyectos grandes de programación?

Qué creen que significa que un código sea "bueno" o "mantenible"?

Frase inspiradora:"**El buen diseño no es gratuito. Pero el mal diseño es mucho más costoso**." — Robert C. Martin

CONTEXTO HISTÓRICO DE SOLID

Quién propuso SOLID?

Fue Robert C. Martin, también conocido como "Uncle Bob", un ingeniero de software muy reconocido en el mundo del desarrollo. Propuso estos principios a principios de los años 2000, aunque en realidad, las ideas detrás de cada principio ya existían mucho antes en la Programación Orientada a Objetos (OOP).

Por qué surgió SOLID?

Para resumir y ordenar las buenas prácticas de diseño de software que ayudaban a construir sistemas más robustos y fáciles de mantener. Uncle Bob agrupó los 5 principios bajo un solo acrónimo fácil de recordar: SOLID.

Raíces históricas

Se inspira en ideas del libro clásico "Design Patterns" (1994) de Gamma, Helm, Johnson, Vlissides (los "Gang of Four"). También toma ideas de teorías de diseño de software como el Principio de Responsabilidad Única o el Principio de Abierto/Cerrado de Bertrand Meyer.

IMPORTANCIA DE SOLID EN EL MUNDO REAL

Aplicaciones prácticas:

- Desarrollo de software empresarial: Sistemas ERP, CRMs, backends de servicios web.
- Desarrollo de APIs y microservicios: Organizar servicios independientes y bien desacoplados.
- Apps móviles: Aplicaciones escalables y fáciles de actualizar.
- Desarrollo de videojuegos: Código de motores de juegos basado en entidades y comportamientos.
- Arquitecturas modernas: MVC, MVP, MVVM, Clean Architecture aplican SOLID directamente.

IMPORTANCIA DE SOLID EN EL MUNDO REAL

En la industria:

- Muchas empresas esperan que un desarrollador aplique SOLID sin que se lo pidan explícitamente.
- Los códigos bien diseñados:
  - Son más fáciles de probar (unit tests).
  - Son más fáciles de mantener (agregar nuevas funciones).
  - Son más fáciles de escalar (soportar más usuarios, más procesos).

¿QUÉ ES SOLID?

SOLID es un conjunto de cinco principios fundamentales de diseño orientado a objetos, que buscan mejorar la calidad, mantenibilidad y flexibilidad del software. Fueron propuestos por Robert C. Martin (Uncle Bob) en los años 2000 y son ampliamente utilizados por desarrolladores de software para crear sistemas más robustos, escalables y fáciles de entender.

¿Qué significa SOLID?

S — Single Responsibility Principle (SRP) Principio de Responsabilidad Única

Una clase debe tener una única razón para cambiar. Esto significa que cada clase debe estar enfocada en una sola responsabilidad, o tarea. Si una clase tiene más de una razón para cambiar, se vuelve difícil de mantener.

O — Open/Closed Principle (OCP) Principio de Abierto/Cerrado

El código debe estar abierto para la extensión pero cerrado para la modificación. Esto implica que podemos agregar nuevas funcionalidades sin modificar el código existente, favoreciendo el uso de herencia y polimorfismo.

¿QUÉ ES SOLID?

L — Liskov Substitution Principle (LSP) Principio de Sustitución de Liskov

Las subclases deben ser intercambiables por sus superclases sin alterar el comportamiento esperado del programa. Es decir, cualquier instancia de una clase derivada debe poder ser utilizada en lugar de su clase base sin causar errores.

I — Interface Segregation Principle (ISP) Principio de Segregación de Interfaces

Es mejor tener varias interfaces específicas en lugar de una sola interfaz general. Esto asegura que las clases no dependan de métodos que no van a utilizar, lo que reduce el acoplamiento entre componentes.

D — Dependency Inversion Principle (DIP) Principio de Inversión de Dependencias

Las clases deben depender de abstracciones (interfaces o clases abstractas), no de implementaciones concretas. Esto permite que las clases sean fácilmente modificables sin afectar a otras partes del sistema.

¿QUÉ ES SOLID?

Letra	Sí (✅)	No (❌)
S	Una responsabilidad por clase	Muchas responsabilidades mezcladas
O	Extiende, no modifiques	Cambiar código viejo todo el tiempo
L	Subclases que respetan la superclase	Subclases que rompen reglas
I	Interfaces específicas	Interfaces gigantes que obligan
D	Depender de abstracciones	Depender de clases concretas

CARACTERÍSTICAS DEL PRINCIPIO SOLID

- Modularidad: Cada módulo tiene una función clara y bien definida.
- Extensibilidad: Permiten agregar nuevas funcionalidades sin romper el sistema.
- Flexibilidad: Facilitan cambios de requerimientos.
- Reutilización: Favorecen la creación de componentes reutilizables.
- Bajo acoplamiento: Los componentes no dependen fuertemente unos de otros.
- Alta cohesión: Cada componente está enfocado en una única responsabilidad.

VENTAJAS

☑ Mejor mantenibilidad: El código puede actualizarse sin necesidad de rehacer todo.

☑ Facilita la comprensión: El código es más claro y fácil de entender para otros desarrolladores.

☑ Reduce la complejidad: Evita "God classes“ (demasiadas cosas al mismo tiempo.) o "código espagueti“ (es muy difícil de entender y de modificar).

☑ Incrementa la posibilidad de reutilización: Módulos y clases son más independientes y reaprovechables.

☑ Facilita el testing: Al tener componentes pequeños y aislados, se facilita la escritura de pruebas unitarias.

DESVENTAJAS

⚠ Sobrediseño: Si se aplica sin necesidad real, puede generar arquitecturas innecesariamente complicadas.

⚠ Mayor curva de aprendizaje: Requiere experiencia para aplicar los principios de manera adecuada.

⚠ Tiempo inicial de desarrollo: Diseñar correctamente desde el inicio puede ser más costoso en términos de tiempo.

⚠ Exceso de clases o interfaces: Dividir demasiado puede hacer que el proyecto tenga demasiados componentes pequeños.

EJEMPLO

S - Single Responsibility Principle (SRP)

Una clase debe tener una única razón para cambiar, es decir, debe tener una sola responsabilidad.

```java
class ManejadorEmpleados {

    public void AgregarEmpleado(String name) {
        System.out.println("Agregar Empleado: " + name);
    }

    public void CalcularSalario(String name) {
        System.out.println("Calculando Salario para: " + name);
    }

    public void GenerarReporte(String name) {
        System.out.println("Generating report for: " + name);
    }
}

public class S {
    public static void main(String[] args) {
        ManejadorEmpleados empleado = new ManejadorEmpleados();
        empleado.AgregarEmpleado("Juan Perez");
        empleado.CalcularSalario("Juan Perez");
        empleado.GenerarReporte("Juan Perez");
    }
}
```

```java
// Clase para agregar empleados
class EmpleadoAdder {
    public void AgregarEmpleado(String name) {
        System.out.println("Agregar Empleado: " + name);
    }
}

// Clase para calcular salarios
class SalarioCalculator {
    public void CalcularSalario(String name) {
        System.out.println("Calculando Salario para: " + name);
    }
}

// Clase para generar reportes
class ReporteGenerator {
    public void GenerarReporte(String name) {
        System.out.println("Generando reporte para: " + name);
    }
}

// Clase principal
public class S {
    public static void main(String[] args) {
        // Crear instancias de las clases que tienen una única responsabilidad
        EmpleadoAdder empleadoAdder = new EmpleadoAdder();
        SalarioCalculator salarioCalculator = new SalarioCalculator();
        ReporteGenerator reporteGenerator = new ReporteGenerator();

        String nombreEmpleado = "Juan Perez";
        // Usar cada clase para su responsabilidad
        empleadoAdder.AgregarEmpleado(nombreEmpleado);
        salarioCalculator.CalcularSalario(nombreEmpleado);
        reporteGenerator.GenerarReporte(nombreEmpleado);
    }
}
```

EJEMPLO

O - Open/Closed Principle (OCP)

Las clases deben estar abiertas para su extensión, pero cerradas para su modificación.

```java
class CalculadoraBonos {
    public double calcularBono(String tipoEmpleado, double salario) {
        if (tipoEmpleado.equals("Junior")) {
            return salario * 0.05;
        } else if (tipoEmpleado.equals("Senior")) {
            return salario * 0.10;
        } else if (tipoEmpleado.equals("Manager")) {
            return salario * 0.20;
        } else {
            return 0;
        }
    }
}

public class O {
    public static void main(String[] args) {
        CalculadoraBonos calculadora = new CalculadoraBonos();

        double bono1 = calculadora.calcularBono("Junior", 1000);
        double bono2 = calculadora.calcularBono("Senior", 2000);
        double bono3 = calculadora.calcularBono("Manager", 3000);

        System.out.println("Bono Junior: " + bono1);
        System.out.println("Bono Senior: " + bono2);
        System.out.println("Bono Manager: " + bono3);
    }
}
```

```java
interface BonoEmpleado {
    double calcularBono(double salario);
}
// Implementaciones específicas
class BonoJunior implements BonoEmpleado {
    @Override
    public double calcularBono(double salario) {
        return salario * 0.05;
    }
}
class BonoSenior implements BonoEmpleado {
    @Override
    public double calcularBono(double salario) {
        return salario * 0.10;
    }
}
class BonoManager implements BonoEmpleado {
    @Override
    public double calcularBono(double salario) {
        return salario * 0.20;
    }
}
// Calculadora que usa polimorfismo
class CalculadoraBonos {
    public double calcular(BonoEmpleado bono, double salario) {
        return bono.calcularBono(salario);
    }
}
public class O {
    public static void main(String[] args) {
        CalculadoraBonos calculadora = new CalculadoraBonos();
        double bono1 = calculadora.calcular(new BonoJunior(), 1000);
        double bono2 = calculadora.calcular(new BonoSenior(), 2000);
        double bono3 = calculadora.calcular(new BonoManager(), 3000);
        System.out.println("Bono Junior: " + bono1);
        System.out.println("Bono Senior: " + bono2);
        System.out.println("Bono Manager: " + bono3);
    }
```

EJEMPLO

L - Liskov Substitution Principle (LSP)

Los objetos de una clase base deben ser reemplazables por objetos de una clase derivada sin alterar el comportamiento correcto del programa.

```java
class Vehiculo {
    public void arrancar() {
        System.out.println("El vehículo está arrancando...");
    }
}
// Clase derivada que no sigue la lógica de la clase base
class Coche extends Vehiculo {
    @Override
    public void arrancar() {
        throw new UnsupportedOperationException("Los coches no pue
    }
}
public class L {
    public static void main(String[] args) {
        Vehiculo vehiculo = new Vehiculo();
        vehiculo.arrancar();  // Correcto
        Coche coche = new Coche();
        coche.arrancar();  // Incorrecto, rompe el comportamiento
    }
}
```

```java
class Vehiculo {
    public void arrancar() {
        System.out.println("El vehículo está arrancando...");
    }
}
// Clase derivada que sigue la lógica de la clase base
class Coche extends Vehiculo {
    @Override
    public void arrancar() {
        System.out.println("El coche está arrancando con llave...");
    }
}
class Motocicleta extends Vehiculo {
    @Override
    public void arrancar() {
        System.out.println("La motocicleta está arrancando con botón...");
    }
}
public class L {
    public static void main(String[] args) {
        Vehiculo vehiculo = new Vehiculo();
        vehiculo.arrancar();  // Correcto, comportamiento esperado
        Vehiculo coche = new Coche();  // Se puede usar un Coche donde se e
        coche.arrancar();  // Correcto, comportamiento de coche
        Vehiculo motocicleta = new Motocicleta();  // Se puede usar una Mot
        motocicleta.arrancar();  // Correcto, comportamiento de motocicleta
    }
}
```

EJEMPLO

I - Interface Segregation Principle (ISP)

Los clientes no deben verse forzados a depender de interfaces que no usan.

```java
interface Trabajador {
    void trabajar();
    void comer();
    void dormir();
}
// Clase que implementa todo correctamente
class EmpleadoOficina implements Trabajador {
    @Override
    public void trabajar() {
        System.out.println("Trabajando en la oficina...");
    }
    @Override
    public void comer() {
        System.out.println("Hora del almuerzo.");
    }
    @Override
    public void dormir() {
        System.out.println("Durmiendo después del trabajo.");
    }
}
// Clase que se ve forzada a implementar métodos que no necesita
class Robot implements Trabajador {
    @Override
    public void trabajar() {
        System.out.println("Robot trabajando...");
    }
    @Override
    public void comer() {
        // No aplica para robots
        throw new UnsupportedOperationException("Los robots no comen.");
    }
    @Override
    public void dormir() {
        // Tampoco aplica
        throw new UnsupportedOperationException("Los robots no duermen.");
    }
}
public class I {
    public static void main(String[] args) {
        Trabajador humano = new EmpleadoOficina();
        humano.trabajar();
        humano.comer();
        humano.dormir();
        Trabajador robot = new Robot();
        robot.trabajar();
        robot.comer();  // ¡Rompe el programa!
    }
}
```

```java
interface ITrabajador {
    void trabajar();
}
interface IComedor {
    void comer();
}
interface IDurmiente {
    void dormir();
}
// Ahora cada clase implementa solo lo que necesita
class EmpleadoOficina implements ITrabajador, IComedor, IDurmiente {
    @Override
    public void trabajar() {
        System.out.println("Trabajando en la oficina...");
    }
    @Override
    public void comer() {
        System.out.println("Hora del almuerzo.");
    }
    @Override
    public void dormir() {
        System.out.println("Durmiendo después del trabajo.");
    }
}
class Robot implements ITrabajador {
    @Override
    public void trabajar() {
        System.out.println("Robot trabajando...");
    }
}
public class I {
    public static void main(String[] args) {
        ITrabajador humano = new EmpleadoOficina();
        humano.trabajar();

        IComedor comedor = new EmpleadoOficina();
        comedor.comer();

        ITrabajador robot = new Robot();
        robot.trabajar();
    }
}
```

EJEMPLO

D - Dependency Inversion Principle (DIP)

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

```java
class EmailService {
    public void enviarEmail(String destinatario, String mensaje) {
        System.out.println("Enviando correo a " + destinatario + ": " + mensaje);
    }
}

class FacturaService {
    private EmailService emailService;

    public FacturaService() {
        this.emailService = new EmailService();  // Dependencia directa de la clase EmailService
    }

    public void generarFactura(String destinatario) {
        // Lógica para generar factura...
        System.out.println("Generando factura...");
        emailService.enviarEmail(destinatario, "Tu factura ha sido generada.");
    }
}

public class D {
    public static void main(String[] args) {
        FacturaService facturaService = new FacturaService();
        facturaService.generarFactura("cliente@correo.com");
    }
}
```

```java
interface Notificador {
    void enviarNotificacion(String destinatario, String mensaje);
}
// Implementación concreta para enviar emails
class EmailService implements Notificador {
    @Override
    public void enviarNotificacion(String destinatario, String mensaje) {
        System.out.println("Enviando correo a " + destinatario + ": " + mensaje);
    }
}
// Implementación concreta para enviar mensajes por SMS (puede ser otro servicio)
class SMSService implements Notificador {
    @Override
    public void enviarNotificacion(String destinatario, String mensaje) {
        System.out.println("Enviando SMS a " + destinatario + ": " + mensaje);
    }
}
// Clase de alto nivel que ahora depende de la abstracción Notificador
class FacturaService {
    private Notificador notificador;

    // Inyección de dependencia a través del constructor
    public FacturaService(Notificador notificador) {
        this.notificador = notificador;
    }
    public void generarFactura(String destinatario) {
        // Lógica para generar factura...
        System.out.println("Generando factura...");
        notificador.enviarNotificacion(destinatario, "Tu factura ha sido generada.");
    }
}
public class D {
    public static void main(String[] args) {
        // Podemos decidir qué implementación usar para la notificación
        Notificador emailService = new EmailService();
        FacturaService facturaService = new FacturaService(emailService);
        facturaService.generarFactura("cliente@correo.com");
        // Si queremos usar SMS en lugar de email, solo cambiamos la implementación
        Notificador smsService = new SMSService();
        facturaService = new FacturaService(smsService);
        facturaService.generarFactura("cliente@correo.com");
    }
```

PREGUNTAS