UNIVERSIDAD DON BOSCO

INGENERIA EN CIENCIAS DE LA COMPUTACION (VIRTUAL)

investigación de los Principios SOLID

Asignatura:

Diseño y Programación de Software Multiplataforma DPS941 G01T.

Alumno:

Dennis Iván Rivas Figueroa RF140580.

Juan Pablo Flores Santos FS142078.

Docente de la materia:

Ing. Alexander Alberto Sigüenza Campos.

Fecha:

09/06/2023

INDICE

Introducción.

En esta investigación, exploraremos en detalle cada uno de los cinco principios SOLID: el Principio de Responsabilidad Única, el Principio de Abierto/Cerrado, el Principio de Sustitución de Liskov, el Principio de Segregación de Interfaces y el Principio de Inversión de Dependencias. Analizaremos en qué consiste cada principio, cómo se aplica en el desarrollo de software y qué beneficios ofrece su aplicación.

Además, examinaremos ejemplos prácticos de implementación de estos principios en diferentes lenguajes de programación y escenarios de desarrollo. Estos ejemplos nos ayudarán a comprender cómo aplicar los principios SOLID en situaciones reales y cómo pueden mejorar la calidad del código, la facilidad de mantenimiento y la capacidad de extensión de un sistema.

Al final de esta investigación, esperamos tener un entendimiento sólido de los principios SOLID y cómo pueden ser utilizados de manera efectiva en el diseño y desarrollo de software. Estos principios nos permitirán escribir código más modular, flexible y de mayor calidad, lo que resultará en sistemas más robustos y sostenibles a largo plazo.

Los cinco principios SOLID:

- Principio de Responsabilidad Única,
- Principio de Abierto/Cerrado,
- Principio de Sustitución de Liskov,
- Principio de Segregación de Interfaces
- Principio de Inversión de Dependencias

Los Principios SOLID **son un conjunto de principios utilizados en el desarrollo de software para hacer que los diseños sean más comprensibles, flexibles, escalables y resistentes a los cambios** .

Estos principios son:

- **Principio de Responsabilidad Única**: Una clase debe tener una sola razón para cambiar.
- **Principio Abierto/Cerrado**: Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para su extensión, pero cerradas para su modificación.
- **Principio de Sustitución de Liskov**: Las clases derivadas deben ser sustituibles por sus clases base.
- **Principio de Segregación de Interfaces**: Los clientes no deben verse obligados a depender de interfaces que no usan.
- **Principio de Inversión de Dependencias**: Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

Ejemplos de cómo aplicar los principios SOLID:

- **Principio de Responsabilidad Única**: Una clase debe tener una sola razón para cambiar. Por ejemplo, si tienes una clase que se encarga de la lógica de negocios y también se encarga de la persistencia de datos, podrías dividirla en dos clases separadas: una para la lógica de negocios y otra para la persistencia de datos.
- **Principio Abierto/Cerrado**: Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para su extensión, pero cerradas para su modificación. Por ejemplo, si tienes una clase que se encarga de la lógica de negocios y necesitas agregar una nueva funcionalidad, en lugar de modificar la clase existente, podrías crear una nueva clase que extienda la clase existente y agregue la nueva funcionalidad.
- **Principio de Sustitución de Liskov**: Las clases derivadas deben ser sustituibles por sus clases base. Por ejemplo, si tienes una clase base que define un método que acepta un objeto de tipo Animal y una clase derivada que define un método que acepta un objeto de tipo Perro, el método en la clase derivada debe ser compatible con el método en la clase base.
- **Principio de Segregación de Interfaces**: Los clientes no deben verse obligados a depender de interfaces que no usan. Por ejemplo, si tienes una interfaz que define varios

métodos y una clase que solo necesita implementar algunos de esos métodos, podrías dividir la interfaz en varias interfaces más pequeñas para que las clases solo implementen los métodos que necesitan.

- **Principio de Inversión de Dependencias**: Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Por ejemplo, si tienes una clase que depende directamente de otra clase concreta, podrías crear una interfaz entre las dos clases para reducir el acoplamiento y aumentar la flexibilidad.

¿Cómo puedo aplicar los principios SOLID en mi código?

Para aplicar los principios SOLID en tu código, puedes seguir los siguientes pasos:

- Aprende los principios SOLID: El primer paso es aprender los cinco principios SOLID y comprender cómo se aplican en el desarrollo de software.
- Identifica las responsabilidades de cada componente: Cada componente de tu sistema debe tener una única responsabilidad. Identifica las responsabilidades de cada componente y asegúrate de que no estén sobrecargados con tareas adicionales.

- Mantén el acoplamiento bajo: El acoplamiento se refiere a la dependencia entre diferentes componentes de un sistema. Para mantener el acoplamiento bajo, asegúrate de que cada componente tenga una única responsabilidad y que no dependa directamente de otros componentes.

- Asegúrate de que tus componentes sean cohesivos: La cohesión se refiere a la relación entre las diferentes partes de un componente. Asegúrate de que cada componente tenga una alta cohesión y que sus partes estén relacionadas entre sí.

- Crea pruebas unitarias y de integración: Los principios SOLID hacen que el código sea más modular y fácil de probar. Crea pruebas unitarias y de integración para asegurarte de que tu código funciona correctamente y cumple con los requisitos del negocio.

- Refactoriza tu código: Si encuentras áreas en tu código que no cumplen con los principios SOLID, tómate el tiempo para refactorizarlo y hacerlo más modular y fácil de mantener.

¿Cuáles son las ventajas de aplicar los principios SOLID?

Las ventajas de aplicar los principios SOLID son muchas y variadas. Algunas de las más importantes son:

- **Software más flexible**: Los principios SOLID mejoran la cohesión y disminuyen el acoplamiento entre los componentes de un sistema, lo que hace que el software sea más fácil de mantener y modificar.
- **Mejor comprensión de la arquitectura del software**: Los principios SOLID ayudan a los desarrolladores a comprender mejor la arquitectura del software y cómo se relacionan los diferentes componentes entre sí.
- **Simplificación de la creación de pruebas**: Los principios SOLID hacen que el código sea más modular y fácil de probar, lo que simplifica la creación de pruebas unitarias y de integración.
- **Mayor calidad del software**: Al seguir los principios SOLID, se puede crear un software más robusto, escalable y fácil de mantener, lo que se traduce en una mayor calidad del software en general.

Ejemplos de implementación de estos principios.

Principio de Responsabilidad Única (SRP):

Supongamos que tenemos una clase llamada Usuario que se encarga de gestionar la autenticación y el almacenamiento de información de un usuario. Sin embargo, esta clase también maneja la lógica relacionada con la generación de reportes. Para cumplir con el principio de responsabilidad única, podemos separar estas responsabilidades en dos clases diferentes: Usuario para la autenticación y almacenamiento de información, y GeneradorReportes para la generación de reportes.

Código de ejemplo en java

```
// Antes sin Solid
class Usuario {
public void autenticar() {
// Lógica de autenticación
}
public void almacenarInformacion()
{
// Lógica de almacenamiento de
información
}
public void generarReporte() {
// Lógica de generación de reportes
}
}
```

```
// Después
class Usuario {
  public void autenticar() {
    // Lógica de autenticación
  }

  public void
almacenarInformacion() {
    // Lógica de almacenamiento de
información
  }
}

class GeneradorReportes {
  public void generarReporte() {
    // Lógica de generación de
reportes
  }
}
```

Principio de Abierto/Cerrado (OCP):

Supongamos que tenemos una clase llamada Calculadora que realiza operaciones matemáticas básicas. Ahora, queremos extender la funcionalidad de la calculadora para agregar soporte de nuevas operaciones sin modificar la clase existente. Podemos lograr esto utilizando herencia y polimorfismo.

Código de ejemplo en java

```
class Calculadora {
  public double calcular(double num1, double num2, String
operacion) {
    switch (operacion) {
      case "suma":
        return num1 + num2;
      case "resta":
        return num1 - num2;
      default:
        throw new UnsupportedOperationException("Operación no
soportada");
    }
  }
}

class CalculadoraAvanzada extends Calculadora {
  @Override
  public double calcular(double num1, double num2, String
operacion) {
    switch (operacion) {
      case "multiplicacion":
        return num1 * num2;
      case "division":
        return num1 / num2;
      default:
        return super.calcular(num1, num2, operacion);
    }
  }
}
```

Con esta implementación, podemos agregar nuevas operaciones a la calculadora sin modificar la clase base Calculadora. Simplemente creamos una nueva clase que hereda de Calculadora y sobrescribimos el método calcular() para agregar la nueva operación.

Principio de Sustitución de Liskov (LSP):

Supongamos que tenemos una clase llamada Vehículo y una subclase llamada Coche. El principio de sustitución de Liskov establece que los objetos de la subclase deben poder reemplazar a los objetos de la clase base sin alterar el comportamiento del sistema.

Código de ejemplo en java

```
class Vehiculo {
  public void acelerar() {
    // Lógica de aceleración

}

class Coche extends Vehiculo {
  @Override
  public void acelerar() {
    // Lógica específica del coche
  }
}
     case "division":
       return num1 / num2;
     default:
       return super.calcular(num1, num2, operacion);
    }
  }
}
```

// Ejemplo de uso

Vehiculo vehiculo = new Coche(); // Se crea una instancia de la subclase Coche

vehiculo.acelerar(); // El comportamiento esperado es el de un coche

En este caso, la subclase Coche se puede utilizar en lugar de la clase base vehículo sin introducir comportamientos inesperados. Esto cumple con el principio de sustitución de Liskov.

Principio de Segregación de Interfaces (ISP):

Supongamos que tenemos una interfaz llamada Impresora que define métodos para imprimir, escanear y enviar fax. Sin embargo, no todos los dispositivos de impresión pueden realizar todas estas operaciones. Para cumplir con el principio de segregación de interfaces, podemos dividir esta interfaz en interfaces más pequeñas y específicas.

Código de ejemplo en java

```
interface Impresora {
  void imprimir();
  void escanear();
  void enviarFax();
}

interface ImpresoraSimple {
  void imprimir();
}

interface Escaner {
  void escanear();
}

// Implementación de las interfaces
class ImpresoraMultifuncional implements Impresora, Escaner {
  @Override
  public void imprimir() {
    // Lógica de impresión
  }

  @Override
  public void escanear() {
    // Lógica de escaneo
  }

  @Override
  public void enviarFax() {
    // Lógica de envío de fax
  }
}

class ImpresoraBasica implements ImpresoraSimple {
  @Override
  public void imprimir() {
    // Lógica de impresión
  }
}
```

En este ejemplo, hemos segregado la interfaz Impresora en las interfaces más pequeñas Impresora Simple y Escaner. Esto permite que las clases implementen solo las interfaces que necesitan y evita que dependan de métodos innecesarios.

Principio de Inversión de Dependencias (DIP):

Supongamos que tenemos una clase Cliente que depende de una clase Servicio concreta. Para cumplir con el principio de inversión de dependencias, debemos invertir la dependencia para que Cliente dependa de una abstracción en lugar de una implementación concreta.

Código de ejemplo en java

```
interface Servicio {
  void ejecutar();
}

class ServicioConcreto implements Servicio {
  @Override
  public void ejecutar() {
    // Lógica del servicio concreto
  }
}

class Cliente {
  private Servicio servicio;

  public Cliente(Servicio servicio) {
    this.servicio = servicio;
  }

  public void realizarOperacion() {
    servicio.ejecutar();
  }
}

// Ejemplo de uso
ServicioConcreto servicioConcreto = new ServicioConcreto();
Cliente cliente = new Cliente(servicioConcreto);
cliente.realizarOperacion();
```

En este ejemplo, la clase Cliente depende de la abstracción Servicio, lo que permite que se le pase cualquier implementación de Servicio, como ServicioConcreto. Esto facilita la extensibilidad y el intercambio de implementaciones sin modificar la clase Cliente.