UNIVERSIDAD DE LIMA

FACULTAD DE INGENIERÍA
CARRERA DE INGENIERÍA DE SISTEMAS

# Ingeniería de Software II

Laboratorio N°2: SOLID

---

**Enlace de repositorio de la guía**: https://github.com/anderalarcon/ing-sw-2-2024-1-java

## Single Responsability Principle

Objetivo: *Promover la cohesión y la modularidad en el diseño de software. Este principio establece que una clase debe tener una única razón para cambiar, es decir, que debe tener una sola responsabilidad.*

**Descripción del Problema:**

Se proporciona una implementación inicial de una clase RegistroUsuariosV1 que se encarga tanto de registrar usuarios como de encriptar sus contraseñas. Sin embargo, esta implementación viola el Principio SRP al tener una clase que asume múltiples responsabilidades.

**Tarea:**

1. Analiza la implementación inicial y identifica cómo viola el Principio SRP.
2. Proporciona una nueva implementación que corrija esta violación, dividiendo las responsabilidades en clases separadas de acuerdo con el Principio SRP.
3. La nueva implementación debe tener una clase específica para la lógica de registro de usuarios y otra para la encriptación de contraseñas.
4. Comprueba que la nueva implementación cumple con el Principio SRP explicando cómo cada clase tiene una única responsabilidad y cómo se beneficia el diseño del sistema.

UNIVERSIDAD DE LIMA

## Código

### Clase → RegistroUsuariosV1

```
1
2  import java.security.MessageDigest;
3  import java.security.NoSuchAlgorithmException;
4
5  public class RegistroUsuariosV1 {
6
7      private String nombre;
8      private String contraseña;
9
10     public RegistroUsuariosV1(String nombre, String contraseña) {
11         this.nombre = nombre;
12         this.contraseña = contraseña;
13     }
14
15     public void registrarUsuario() {
16         encriptarContraseña();
17         // Lógica para guardar el usuario en la base de datos
18         System.out.println("Usuario registrado: " + nombre);
19     }
20
21     public void encriptarContraseña() {
22         try {
23             MessageDigest md = MessageDigest.getInstance("SHA-256");
24             byte[] hash = md.digest(contraseña.getBytes());
25             StringBuilder hexString = new StringBuilder();
26             for (byte b : hash) {
27                 String hex = Integer.toHexString(0xff & b);
28                 if (hex.length() == 1) {
29                     hexString.append('0');
30                 }
31                 hexString.append(hex);
32             }
33             this.contraseña = hexString.toString();
34             System.out.println("Contraseña encriptada!: " +
35 this.contraseña);
36         } catch (NoSuchAlgorithmException e) {
37             e.getMessage();
38         }
39     }
40
41 }
```

UNIVERSIDAD DE LIMA

**Clase → Main**

```
1
 2 public class Main {
 3
 4     public static void main(String[] args) {
 5         RegistroUsuariosV1 registro1 = new RegistroUsuariosV1("ander",
 6 "contraseña123");
 7         registro1.registrarUsuario();
 8
 9     }
10 }
```

## Open Close Principle

Objetivo: *Fomentar un diseño de software que sea flexible, fácil de mantener y extender.*

**Descripción del Problema:**

Se proporciona una implementación inicial de una clase GestorDeMensajesV1 que tiene una lógica condicional para enviar mensajes a través de diferentes canales de comunicación (correo electrónico y SMS). Sin embargo, esta implementación viola el Principio OCP al requerir modificaciones directas en la clase GestorDeMensajesV1 cada vez que se quiere agregar un nuevo canal de comunicación.

**Tarea:**

1. Analiza la implementación inicial y cómo viola el Principio OCP al requerir modificaciones directas en la clase GestorDeMensajesV1 para agregar nuevos canales de comunicación.
2. Proporciona una nueva implementación que corrija esta violación, aplicando el Principio OCP para que la clase GestorDeMensajesV2 esté abierta para la extensión, pero cerrada para la modificación.
3. Comprueba que la nueva implementación cumple con el Principio OCP explicando cómo la clase GestorDeMensajesV2 está abierta para la extensión pero cerrada para la modificación, permitiendo agregar nuevos canales de comunicación sin modificar su código

UNIVERSIDAD DE LIMA

## Código

### Clase → GestorDeMensajesV1

```java

public class GestorDeMensajesV1 {

    public void enviarMensaje(String tipo, String contenido) {
        if (tipo.equalsIgnoreCase("correo")) {
            enviarCorreo(contenido);
        } else if (tipo.equalsIgnoreCase("sms")) {
            enviarSMS(contenido);
        }
    }

    private void enviarCorreo(String contenido) {
        System.out.println("Enviando correo V1: " + contenido);
    }

    private void enviarSMS(String contenido) {
        System.out.println("Enviando SMS V1: " + contenido);
    }
}
```

### Clase → Main

```java

public class Main {

    public static void main(String[] args) {
        GestorDeMensajesV1 gestorV1 = new GestorDeMensajesV1();

        gestorV1.enviarMensaje("correo", "¡Hola, esto es un correo!");
        gestorV1.enviarMensaje("sms", "¡Hola, esto es un SMS!");

    }
}
```

UNIVERSIDAD DE LIMA

# Interface Segregation Principle

Objetivo: *Promover un diseño de interfaces de software que sean específicas y cohesivas para los clientes que las utilicen. Este principio se centra en dividir las interfaces grandes y poco específicas en interfaces más pequeñas y específicas, de manera que los clientes solo dependan de las interfaces que necesitan realmente.*

**Descripción del Problema:**

Se proporciona una implementación inicial de clases HumanoV1 y RobotV1, que implementan una interfaz Trabajador con métodos para trabajar, comer, descansar y reunirse. Sin embargo, esta implementación viola el Principio ISP, ya que los robots no necesitan implementar todos los métodos definidos en la interfaz Trabajador.

**Tarea:**

1. Analiza la implementación inicial y cómo viola el Principio ISP al forzar a los robots a implementar métodos que no necesitan.
2. Proporciona una nueva implementación que corrija esta violación, creando interfaces más específicas para los humanos y los robots.
3. Verifica que la nueva implementación cumple con el Principio ISP explicando cómo las interfaces segregadas mejoran la cohesión y la flexibilidad del diseño.

## Código

**Clase → HumanoV1**

```
1
2 public class HumanoV1 implements Trabajador {
3
4     @Override
5     public void trabajar() {
6         System.out.println("Trabajando...");
7     }
8
9     @Override
10    public void comer() {
11        System.out.println("Comiendo...");
12    }
13
14    @Override
15    public void descansar() {
16        System.out.println("Descansando...");
```

```
    }

    @Override
    public void reunirse() {
        System.out.println("Reunión...");
    }
}

```

## Clase → RobotV1

```
public class RobotV1 implements Trabajador {

    @Override
    public void trabajar() {
        System.out.println("Trabajando...");
    }

    @Override
    public void comer() {
        // Los robots no necesitan comer, pero deben implementar este
método debido a la interfaz
        throw new UnsupportedOperationException("Los robots no necesitan
comer");
    }

    @Override
    public void descansar() {
        // Los robots no necesitan descansar, pero deben implementar este
método debido a la interfaz
        throw new UnsupportedOperationException("Los robots no necesitan
descansar");
    }

    @Override
    public void reunirse() {
        // Los robots no necesitan reunirse, pero deben implementar este
método debido a la interfaz
        throw new UnsupportedOperationException("Los robots no necesitan
reunirse");
    }
}
```

## Interfaz

```

public interface Trabajador {

    void trabajar();

```

```
    void comer();

    void descansar();

    void reunirse();
}

```

**Clase → Main**

```

public class Main {

    public static void main(String[] args) {
        Trabajador humano = new HumanoV1();
        humano.trabajar();
        humano.comer();
        humano.descansar();
        humano.reunirse();

        Trabajador robot = new RobotV1();
        robot.trabajar();
        robot.comer(); // Esto lanzará una excepción, ya que los robots no
pueden comer
    }
}
```

## Dependency Inversion Principle

Objetivo: *Reducir el acoplamiento entre módulos de un sistema de software, promoviendo la flexibilidad y la reutilización del código. Este principio establece dos reglas fundamentales:*

1. *Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.*
2. *Las abstracciones no deben depender de los detalles. Los detalles deben depender de abstracciones.*

**Descripción del Problema:**

UNIVERSIDAD DE LIMA

Se proporciona una implementación inicial de una clase InterruptorV1 que depende directamente de la clase concreta LamparaV1. Esto viola el Principio DIP, ya que el InterruptorV1 está fuertemente acoplado a la implementación específica de LamparaV1, lo que dificulta la flexibilidad y la reutilización del código.

**Tarea:**

1. Analiza la implementación inicial y cómo viola el Principio DIP al depender directamente de la clase concreta LamparaV1.
2. Proporciona una nueva implementación que corrija esta violación, aplicando el Principio DIP para que el InterruptorV2 dependa de una abstracción (DispositivoElectrico) en lugar de una implementación concreta.
3. Verifica que la nueva implementación cumple con el Principio DIP explicando cómo el InterruptorV2 depende de una abstracción en lugar de una implementación concreta, lo que mejora la flexibilidad y la mantenibilidad del sistema.

## Código

### Clase → LamparaV1

```
1
2
3  public class LamparaV1 {
4
5      public void encender() {
6          System.out.println("La lámpara se ha encendido");
7      }
8  }
```

### Clase → InterruptorV1

```
1
2  public class InterruptorV1 {
3
4      private LamparaV1 lampara;
5
6      public InterruptorV1() {
7          this.lampara = new LamparaV1();
8      }
9
10     public void presionarInterruptor() {
```

```
11          lampara.encender();
12      }
13 }
14
```

**Clase → Main**

```
1  public class Main {
2
3      public static void main(String[] args) {
4          InterruptorV1 interruptor = new InterruptorV1();
5          interruptor.presionarInterruptor(); // Enciende la lámpara
6  directamente
7
8      }
   }
```