| Laboratorio Nº 01 | |
|---|---|
| **E. P.:** Ingeniería de Sistemas (plan 2018) | **Asignatura:** IS388 Construcción y Evolución de Software |
| **Docente:** Mtro. Ing. Oscar Meneses Yaranga | **Semestre Académico**: 2025-I Presencial |
| **SESIÓN 01: Principios SOLID.** | |

## I. OBJETIVOS

Al término de esta experiencia, el estudiante será capaz de:

- Realizar la Aplicación Práctica de los Principios SOLID
- Conocer las Ventajas de seguir los principios SOLID
- Laboratorio 1 Diagramar el proceso de desarrollo de software
- Tarea 1 Desarrollar un mapa conceptual de Lean, ágil y Devops.

## II. EQUIPOS Y MATERIALES

- ✓ Computadora personal (PC).
- ✓ Compilador/intérprete de lenguajes Orientado a Objetos.
- ✓ Guía de laboratorio.
- ✓ Material bibliográfico con la información en páginas de la sesión de aprendizaje.

## III. METODOLOGÍA Y ACTIVIDADES

a) Teoría de Desarrollo de Software (Construcción y Evolución de Software)
b) Teoría de Sistemas de Información (Ingeniería del Software)
c) Teoría de Proyectos de Desarrollo de Software

**Revisión Teórica:**

- **Origen y significado de SOLID.**
  El acrónimo mnemónico SOLID se originó a comienzos de la década del 2000 a partir de los primeros principios de diseño de software promovidos por *Robert C. Martin* con primer indicio en 1995, también conocido como ***'Uncle Bob'***. Estos principios son una guía para escribir software que sea fácil de entender, mantener y extender. El término SOLID engloba cinco principios fundamentales:
  - Single Responsibility Principle (SRP)
  - Open/Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)

- **Importancia de SOLID en el desarrollo de software**
  Los cinco principios SOLID ayudan a desarrollar código que es modular y adaptable al cambio, lo que permite una evolución continua y simplifica la localización de errores con base en principios de la programación orientada a objetos y el diseño. Siguiendo estos principios básicos, los desarrolladores pueden reducir el riesgo de crear código "frágil", difícil de escalar o propenso a fallos al realizar cambios, consiguiendo un software de calidad, legible, entendible y fácilmente testeable.

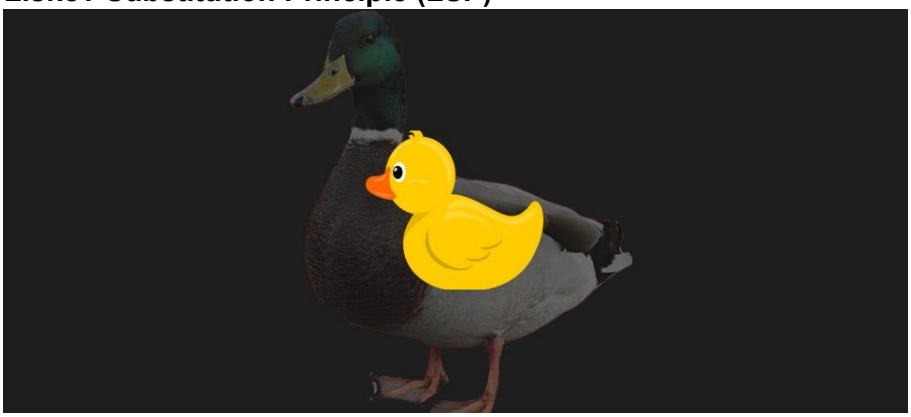
**Single Responsibility Principle (SRP)**


***"A class should have one, and only one, reason to change."***

**Open/Closed Principle (OCP)**

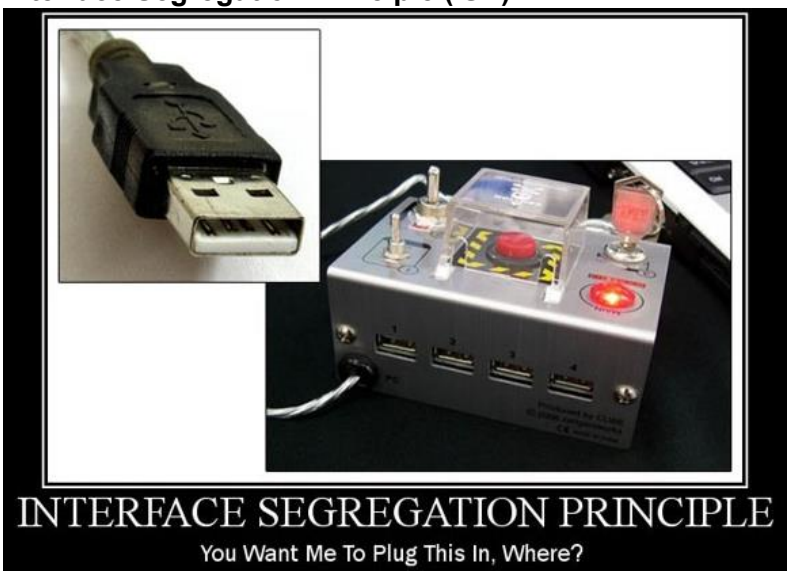
***"You should be able to extend a classes behavior, without modifying it."***

**Liskov Substitution Principle (LSP)**



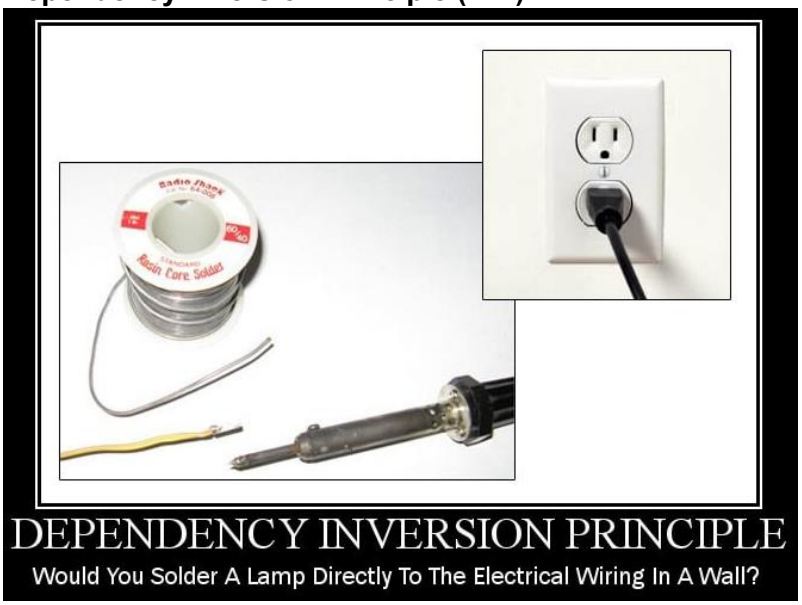
***"Derived classes must be substitutable for their base classes."***

**Interface Segregation Principle (ISP)**




***"Make fine grained interfaces that are client specific."***

**Dependency Inversion Principle (DIP)**




***"Depend on abstractions, not on concretions."***

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos:

- Crear un **software eficaz**: que cumpla con su cometido y que sea **robusto y estable**.
- Escribir un **código limpio y flexible** ante los cambios: que se pueda modificar fácilmente según necesidad, que sea **reutilizable** y **mantenible**.
- Permitir **escalabilidad**: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

En definitiva, desarrollar un **software de calidad**.

En este sentido la aplicación de los principios SOLID está muy relacionada con la comprensión y el uso de patrones de diseño, que nos permitirán mantener una alta **cohesión** y, por tanto, un bajo **acoplamiento** de software.

**Conclusión**

Los principios SOLID no son sólo la teoría, son herramientas prácticas para escribir código limpio, escalable y sostenible.

Usando el contexto del mundo real de una aplicación de pedido de alimentos, hemos visto cómo:

- **SRP** mejora la modularidad.
- **OCP** hace extensible su sistema.
- **LSP** asegura herencias fiables.
- **ISP** mantiene las interfaces limpias y enfocadas.
- **DIP** desvincula la lógica de alto nivel y bajo nivel.

Aplicando SOLID no se trata de añadir complejidad, se trata de escribir un mejor código que resiste la prueba del tiempo.

## IV. INDICACIONES

Antes de empezar con el laboratorio, verificar tenerlos las aplicaciones de Office, puesto los documentos se redactarán en Word o similares, analizar los siguientes:

**¿Qué son la cohesión y el acoplamiento?**

Son dos conceptos muy relevantes a la hora de diseñar y desarrollar software. Veamos en qué consisten.

**Acoplamiento**

El acoplamiento se refiere al **grado de interdependencia que tienen dos unidades de software entre sí**, entendiendo por unidades de software: clases, subtipos, métodos, módulos, funciones, bibliotecas, etc.

Si dos unidades de software son completamente independientes la una de la otra, decimos que están desacopladas.

**Cohesión**

La cohesión de software es el **grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado** que si trabajaran por separado. Se refiere a la forma en que podemos agrupar diversas unidades de software para crear una unidad mayor.

**Críticas a SOLID**

El ámbito del desarrollo de software es un terreno de **continuo debate** y SOLID no se queda fuera de la controversia.

Aunque estos cinco principios son considerados por muchos como una **base fundamental de un buen desarrollo** o al menos como una guía a tener en cuenta, no son pocos los profesionales que critican los principios SOLID.

Los acusan de ambiguos, confusos, de complicar el código, de demorar el proceso de desarrollo y los tildan incluso de totalmente equivocados e innecesarios.

Aquí tienes un par de ejemplos de estas críticas:

The SOLID Design Principles – Absolute Nonsense: https://jamesmccaffrey.wordpress.com/2016/08/24/the-solid-design-principles-absolute-nonsense/

Why Every Element of SOLID is Wrong: https://speakerdeck.com/tastapod/why-every-element-of-solid-is-wrong

Y como a toda acción le corresponde una reacción, las respuestas en defensa de SOLID:

In Defense of the SOLID Principles: https://blog.ndepend.com/defense-solid-principles/

Why Every Single Argument of Dan North is Wrong: https://www.entropywins.wtf/blog/2017/02/17/why-every-single-argument-of-dan-north-is-wrong/

## V. PROCEDIMIENTO

Cumplir con las indicaciones dadas, para esto Ud. deberá seguir los siguientes pasos:

Los principios SOLID son fundamentales en la programación orientada a objetos y pueden aplicarse en cualquier lenguaje que adopte este paradigma, pero siguiendo estos procedimeintos.
Ahora que ya conoces el significado del acrónimo y el origen de los principios, es importante dar un paso atrás para comprender el paradigma de la Programación Orientada a Objetos (POO). Después de todo, como ya sabes, a través de la POO es posible aplicar los principios SOLID.


Observa que la clase “carro” representa un modelo abstracto de lo que un auto necesita tener: el tipo, color, placa y número de puertas.

A partir de este modelo (que es la clase), podemos crear representaciones más específicas de autos, que son los objetos:

*Tipo: Porsche*
*Color: Blanco*
*Placa: MHZ-4345*
*Número de puertas: 4*

o

*Tipo: Ferrari*
*Color: Rojo*
*Placa: JKL-0001*
*Número de puertas: 4*

De esta manera, si es necesario agregar más información sobre los autos, como si tienen aire acondicionado o no, podemos modificar la clase.

A partir de esto, todos los objetos se actualizarán automáticamente con un nuevo campo que debe ser llenado.

En este sentido, el paradigma promueve la reutilización del código y facilita el mantenimiento, tanto en sistemas simples, como el ejemplo, como en los más complejos.

Cabe mencionar que la Programación Orientada a Objetos se utiliza en muchos lenguajes de programación populares, como Java, Python, PHP, C++, C#, entre otros.

Ahora que ya conoces el concepto de POO, es hora de aprender sobre cada uno de los principios SOLID.

**Principio de Responsabilidad Única (S - Single Responsibility Principle).** Para entender el principio de responsabilidad única, pensemos en el desarrollo de un gestor de tareas. Comencemos con el siguiente código:

```
public class GestorTareas {
    public String conectarAPI(){
        //...
    }
    public void crearTarea(){
        //...
    }
    public void actualizarTarea(){
        //...
    }
    public void eliminarTarea(){
        //...
    }
    public void enviarNotificacion(){
        //...
    }
    public void generarInforme(){
        //...
    }
    public void enviarInforme(){
```

```
        //...
    }
}
```

**Problemática**

Intenta enumerar todas las funciones que tiene la clase GestorTareas. Es responsable de manejar todas las operaciones de las tareas en sí, además de consumir una API, enviar notificaciones a los usuarios y generar informes de la aplicación.

Piensa en la Programación Orientada a Objetos. ¿Debería un objeto gestor de tareas enviar correos electrónicos y generar informes? ¡No! Un gestor de tareas se encarga de gestionar las tareas, no de correos electrónicos o informes.

Como en la imagen, no es recomendable que una sola herramienta tenga 1001 utilidades, ya que esto puede causar confusión.

Esto se intensifica cuando hablamos de desarrollo de software. Lo ideal es crear varias herramientas diferentes, cada una con su propia función.

**Solución**

Para resolver este problema, vamos a crear diferentes clases, cada una representando una función.

Nuestra clase GestorTareas solo tendrá el código relacionado con las operaciones de tareas. Otras operaciones estarán en otras clases. Y cada clase será responsable de una parte diferente de la aplicación.

Así, tendremos la clase GestorTareas refactorizada:

```
public class GestorTareas {
    public void crearTarea(){
        //...
    }
    public void actualizarTarea(){
        //...
    }
    public void eliminarTarea(){
        //...
    }
}
```

De esta forma, crearemos una clase para consumir una API externa, otra para enviar notificaciones y una última para gestionar los informes.

```
public class ConectorAPI {
    public String conectarAPI() {
        //...
    }
}
public class Notificador {
    public void enviarNotificacion() {
```

```
        //...
    }
}
public class GeneradorInforme {
    public void generarInforme(){
        //...
    }
    public void enviarInforme(){
        //...
    }
}
```

Quizás te preguntes si las clases son demasiado pequeñas. En este caso, no lo son. Cada clase refleja exactamente la responsabilidad que tiene.

Si necesitamos agregar algún método relacionado con el consumo de la API, sabremos exactamente en qué parte del código hacerlo. Es decir, será mucho más fácil realizar los cambios necesarios.

**Definición del Principio de Responsabilidad Única**

En resumen, el principio de responsabilidad única dice que: "Cada clase debe tener un, y solo un, motivo para cambiar."

Si una clase tiene varias responsabilidades, cambiar un requisito del proyecto puede traer varias razones para modificar la clase. Por lo tanto, las clases deben tener responsabilidades únicas.

Este principio también se puede extender a los métodos que creamos. Cuantas más tareas realiza un método, más difícil es probarlo y garantizar que el programa esté en orden.

Un consejo para aplicar este principio en la práctica es intentar nombrar tus clases o métodos con todo lo que son capaces de hacer.

Si el nombre es demasiado largo, como GestorTareasEmailsInformes, es una señal de que el código puede ser refactorizado.

**Ventajas de aplicar el Principio de Responsabilidad Única**

Existen varios beneficios al aplicar este principio, entre los cuales destacan:

- Facilidad para realizar mantenimientos
- Reutilización de clases
- Facilidad para realizar pruebas
- Simplificación de la legibilidad del código

**Principio Abierto-Cerrado (O - Open-Closed Principle)**

Para entender el Principio Abierto-Cerrado (la letra O del acrónimo), pensemos que estamos trabajando en el sistema de una clínica médica.

En esta clínica, existe una clase que gestiona las solicitudes de exámenes. Inicialmente, el único examen posible es el de sangre. Por eso, tenemos el siguiente código:

```
public class AprobarExamen {
    public void aprobarSolicitudExamen(Examen examen){
        if(verificaCondicionesExamenSangre(examen))
            System.out.println("¡Examen aprobado!");
    }
    public boolean verificaCondicionesExamenSangre(){
        //...
    }
}
```

Ahora necesitamos agregar una nueva funcionalidad al sistema: la clínica empezará a hacer exámenes de Rayos X. ¿Cómo incluimos esto en nuestro código?

Una alternativa sería verificar qué tipo de examen se está realizando para poder aprobarlo:

```
public class AprobarExamen {
    public void aprobarSolicitudExamen(Examen examen){
        if(examen.tipo == SANGRE){
            if(verificaCondicionesExamenSangre(examen))
                System.out.println("¡Examen de sangre aprobado!");
        } else if(examen.tipo == RAYOSX) {
            if (verificaCondicionesRayosX(examen))
                System.out.println("¡Rayos X aprobado!");
        }
    }
    private boolean verificaCondicionesExamenSangre(){
        //...
    }
    private boolean verificaCondicionesRayosX(){
        //...
    }
}
```

**Problemática**

A simple vista, parece que todo está bien, ¿no? Nuestro código funciona correctamente y hemos añadido la funcionalidad de manera adecuada.

Pero, ¿y si además de Rayos X, la clínica también comenzara a hacer ultrasonidos? Siguiendo esta lógica, agregaríamos otro if en el código y otro método para verificar las condiciones específicas del examen.

Esta definitivamente no es una buena estrategia. Cada vez que añadimos una función, la clase (y el proyecto en su conjunto) se vuelve más compleja.

Por eso, es necesario contar con una estrategia para agregar más funcionalidades al proyecto sin modificar y desordenar la clase original.

**Solución**

En este escenario, el proyecto comprende varios tipos de aprobación de exámenes. Por lo tanto, podemos crear una clase o interfaz que represente una aprobación de manera genérica.

Para cada tipo de examen proporcionado por la clínica, es posible crear nuevos tipos de aprobación, más específicos, que implementen la interfaz. Así, podemos tener el siguiente código:

```
public interface AprobarExamen{
    void aprobarSolicitudExamen(Examen examen);
    boolean verificaCondicionesExamen(Examen examen);
}
public class AprobarExamenSangre implements AprobarExamen{
    @Override
    public void aprobarSolicitudExamen(Examen examen){
            if(verificaCondicionesExamen(examen))
                System.out.println("¡Examen de sangre aprobado!");
    }
    @Override
    boolean verificaCondicionesExamen(Examen examen){
        //...
    }
}
public class AprobarRayosX implements AprobarExamen{
    @Override
    public void aprobarSolicitudExamen(Examen examen){
        if(verificaCondicionesExamen(examen))
            System.out.println("¡Rayos X aprobado!");
    }
    @Override
    boolean verificaCondicionesExamen(Examen examen){
        //...
    }
}
```

Ahora, como la interfaz representa la aprobación de un examen, para agregar un nuevo tipo de examen, solo necesitamos crear una nueva clase que implemente la interfaz AprobarExamen. Esta clase representará cómo se aprueba el nuevo examen.

Observa que siempre será posible implementar la interfaz AprobarExamen al añadir nuevas funcionalidades. Sin embargo, la interfaz en sí no cambia. Estamos extendiéndola, pero no modificándola.

**Definición del Principio Abierto-Cerrado**

Podemos definir el Principio Abierto-Cerrado de la siguiente manera: "Las entidades de software (como clases y métodos) deben estar abiertas para la extensión, pero cerradas para la modificación."

Es decir, si una clase está abierta para la modificación, cuantos más recursos añadamos, más compleja se volverá.

Lo ideal es adaptar el código no para modificar la clase, sino para extenderla. En general, esto se logra cuando abstraemos un código en una interfaz.

Aplicando el principio de Open-Closed, es posible hacer que nuestro código se asemeje más al mundo real, practicando de manera sólida la orientación a objetos.

Piensa en un camión: toda su implementación, como el motor, la batería y la cabina, está cerrada para modificaciones.

Sin embargo, podemos extender las tareas que realiza dependiendo del tipo de carrocería que le acoplemos, como muestra la figura a continuación:

**Ventajas de aplicar el Principio Abierto-Cerrado**

Al aplicar este principio, es posible hacer que el proyecto sea mucho más flexible. Agregar nuevas funcionalidades se vuelve una tarea más sencilla.

Además, los códigos se vuelven más fáciles de leer. Con todo esto, el riesgo de introducir errores disminuye significativamente.

Asimismo, este principio nos lleva directamente a la aplicación de algunos patrones de diseño, como el Strategy.

De esta forma, alineamos varias buenas prácticas de desarrollo. El resultado es un código cada vez más limpio y organizado.

**Principio de Sustitución de Liskov (L - Liskov Substitution Principle)**
Para entender el Principio de Sustitución de Liskov (la letra L del acrónimo), pensemos en el siguiente escenario: el desarrollo de un sistema para una universidad.

Dentro del sistema, hay una clase madre Estudiante, que representa a un estudiante de grado, y su clase hija, EstudiantePosGraduacion, con el siguiente código:

```
public class Estudiante {
    String nombre;
    public Estudiante(String nombre) {
        this.nombre = nombre;
    }
    public void estudiar() {
        System.out.println(nombre + " está estudiando.");
    }
}
public class EstudianteDePosGraduacion extends Estudiante {
    @Override
    public void estudiar() {
        System.out.println(nombre + " está estudiando y realizando investigaciones.");
    }
}
```

Para agregar la funcionalidad entregarTesis() al sistema, simplemente colocamos este método en la clase Estudiante. El código quedaría así:

```
class Estudiante {
    String nombre;
    public Estudiante(String nombre) {
        this.nombre = nombre;
    }
    public void estudiar() {
        System.out.println(nombre + " está estudiando.");
    }
    public void entregarTesis(){
        //…
    }
}
```

**Problemática**
Probablemente ya notaste algo incorrecto en el código. Normalmente, los estudiantes de posgrado no entregan tesis de grado (TCC).

Sin embargo, la clase EstudianteDePosGraduacion es hija de Estudiante, y, por lo tanto, debería tener todos los comportamientos de esta.

Una alternativa sería sobrescribir el método entregarTesis() en la clase EstudianteDePosGraduacion lanzando una excepción.

No obstante, esto seguiría siendo problemático: la clase EstudianteDePosGraduacion aún no tendría los mismos comportamientos que la clase Estudiante.

Lo ideal es que, en los lugares donde se utilice la clase Estudiante, sea posible usar una clase EstudianteDePosGraduacion, ya que, por herencia, un estudiante de posgrado es un estudiante.

**Solución**

La solución a este problema es modificar nuestro modelo. Podemos crear una nueva clase EstudianteDeGraduacion, que también heredará de Estudiante. Esta clase tendrá el método entregarTesis():

```
public class EstudianteDeGraduacion extends Estudiante {
    public void estudiar() {
        System.out.println(nombre + " está estudiando en la licenciatura.");
    }
    public void entregarTesis() {
        //…
    }
}
```

Observa que, de esta manera, nuestras clases representan mejor el mundo real. No estamos forzando a una clase a hacer algo que originalmente no hace.

Además, si necesitamos utilizar una instancia de Estudiante, podemos pasar, sin miedo, una instancia de EstudianteDeGraduacion o de EstudianteDePosGraduacion.

Al fin y al cabo, estas clases pueden ejecutar todas las funciones de Estudiante, incluso teniendo funciones más específicas.

**Definición del Principio de Sustitución de Liskov**

Quien propuso el Principio de Sustitución de Liskov de manera formal y matemática fue Bárbara Liskov.

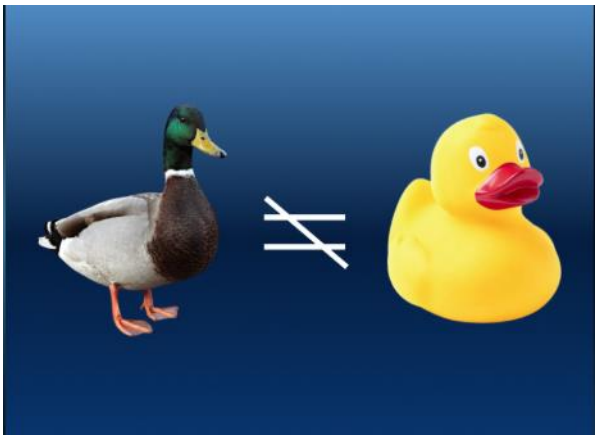
Sin embargo, Robert Martin dio una definición más simple: "Las clases derivadas (o clases hijas) deben poder sustituir a sus clases base (o clases madres)".

Es decir, una clase hija debe ser capaz de ejecutar todo lo que su clase madre hace. Este principio está relacionado con el polimorfismo y refuerza este pilar de la POO.

Es importante notar que, al entender este principio, nos volvemos más atentos al código: si un método de una clase hija tiene un retorno muy diferente al de la clase madre o lanza una excepción, por ejemplo, ya es una señal de que algo está mal.

Si en tu programa tienes una abstracción que parece un pato, suena como un pato, nada como un pato, pero necesita baterías, tu abstracción está equivocada.

Imagina que en tu proyecto tienes una clase Pato y una clase hija de esta, PatoDeGoma. En esta imagen puedes ver que un pato y un pato de goma son bastante diferentes:



Se en una parte del código necesitas usar un objeto Pato, pero usas un PatoDeGoma en su lugar, podrías tener problemas.

Esto viola el principio de sustitución de Liskov, ya que no podemos sustituir completamente una clase padre por una clase hija.

**Ventajas de aplicar el Principio de Sustitución de Liskov**

Aplicar este principio nos brinda diversos beneficios, especialmente para tener una modelación más fiel a la realidad, reducir errores inesperados en el programa y simplificar el mantenimiento del código.

**Principio de Segregación de Interfaz (I - Interface Segregation Principle)**
Para entender el Principio de Segregación de Interfaz, imaginemos que estamos trabajando en un sistema de gestión de empleados de una empresa.

Vamos a crear una interfaz, como muestra el código a continuación:

**Interfaz Funcionario**

```
public interface Funcionario {
    public BigDecimal salario();
    public BigDecimal generarComision();
}
```

Observa que creamos la interfaz para establecer un “contrato” con las personas que son empleados de esta empresa.

En este contexto, el siguiente código describe dos clases que hacen referencia a dos profesiones en esta empresa: Vendedor y Recepcionista.

Ambas usan la interfaz Funcionario y, por lo tanto, deben implementar los métodos salario() y generarComision().

**Clase Vendedor**

```
import java.math.BigDecimal;
public class Vendedor implements Funcionario {
    @Override
    public BigDecimal salario() {
    }
    @Override
    public BigDecimal generarComision() {
    }
}
```

**Clase Recepcionista**

```
import java.math.BigDecimal;
public class Recepcionista implements Funcionario {
    @Override
    public BigDecimal salario() {
    }
    @Override
    public BigDecimal generarComision() {
    }
}
```

## Problemática

Analizando el código anterior, ¿tiene sentido que una persona que tiene el cargo de vendedora o recepcionista reciba salario? ¡Sí! Después de todo, todos tenemos facturas que pagar.

Siguiendo esta misma lógica, ¿tiene sentido que una persona con el cargo de vendedora o recepcionista reciba comisión? ¡No!

Para una persona que tiene el cargo de vendedora, tiene sentido. Pero para la persona con el cargo de recepcionista, no tiene sentido.

Es decir, la clase Recepcionista fue obligada a implementar un método que no tiene sentido para ella. Aunque sea empleada de esta empresa, ese puesto no recibe comisión.

Por lo tanto, podemos notar que este problema fue causado por tener una interfaz genérica.

## Solución

Para resolver esto, es posible crear interfaces específicas. En lugar de tener una única interfaz Funcionario, podemos tener dos: Funcionario y Comisionable.

**Interfaz Funcionario**

```
import java.math.BigDecimal;
public interface Funcionario {
    public BigDecimal salario();
}
```

Observa que mantenemos la interfaz Funcionario, pero eliminamos el método generarComision(), que es específico de algunos empleados, para agregarlo en una nueva interfaz Comisionable:

**Interfaz Comisionable**

```
import java.math.BigDecimal;
public interface Comisionable {
    public BigDecimal generarComision();
}
```

Ahora, las personas que tienen derecho a recibir comisión implementarán la interfaz Comisionable. Un ejemplo de esto es la clase Vendedor:

**Vendedor**

```
import java.math.BigDecimal;
public class Vendedor implements Funcionario, Comisionable {
    @Override
    public BigDecimal salario() {
    }
    @Override
    public BigDecimal generarComision() {
    }
}
```

Ahora, la clase Recepcionista puede implementar la interfaz Funcionario sin tener la obligación de crear el método generarComision():

**Recepcionista**

```
import java.math.BigDecimal;
public class Recepcionista implements Funcionario {
    @Override
    public BigDecimal salario() {
    }
}
```

## Definición del Principio de Segregación de Interfaz

Como analizamos en el código anterior, podemos notar que:

Debemos crear interfaces específicas en lugar de tener una única interfaz genérica.

Y esto es precisamente lo que dice el Principio de Segregación de Interfaz: "Una clase no debe ser obligada a implementar interfaces y métodos que no serán utilizados".

Es posible que alguna vez hayas comprado un adaptador con múltiples entradas, como en la foto. La mayoría de las veces, las personas no saben para qué sirven todas las conexiones.

Siguiendo esta analogía, si no necesitamos un conector o entrada específica, no tiene sentido incluirlos, como comprar un conector personalizado para un dispositivo específico.

Es decir, tampoco se debe obligar a una clase a implementar métodos que no se utilizarán.

**Ventajas de aplicar el principio de segregación de interfaces**
Seguir el principio de segregación de interfaces ayuda a promover la cohesión y la flexibilidad en nuestros sistemas, haciéndolos fáciles de mantener y ampliar.

**Principio de inversión de dependencia (D - Dependency Inversion Principle)**
Para comprender el Principio de Inversión de Dependencia (letra D del acrónimo), imaginemos que estamos trabajando en una startup de e-commerce y necesitamos desarrollar el sistema de gestión de pedidos.

Sin conocer el Principio de Inversión de Dependencia, es muy probable que desarrollemos una clase PedidoService similar al siguiente código:

**Clase PedidoService**
```
public class PedidoService {
    private PedidoRepository repository;
    public PedidoService() {
        this.repository = new PedidoRepository();
    }
    public void procesarPedido(Pedido pedido) {
        // Lógica de procesamiento del pedido
        repository.guardarPedido(pedido);
    }
}
```

**Problemática**
Aparentemente, el código parece estar correcto. Sin embargo, si algún día necesitamos cambiar el almacenamiento de este pedido a otro lugar (por ejemplo, una API externa), necesitaremos más de una clase para resolver el problema.

Después de todo, la clase PedidoService está directamente acoplada a la implementación concreta de la clase PedidoRepository.

**Solución**
Para resolver este problema, podemos crear una interfaz para la clase de acceso a la base de datos e inyectarla en la clase PedidoService.

De esta manera, estamos dependiendo de abstracciones y no de implementaciones concretas.

**Interfaz PedidoRepository**
```
public interface PedidoRepository {
    void guardarPedido(Pedido pedido);
}
```

**Clase PedidoService**

```
public class PedidoService {
    private PedidoRepository repository;
    public PedidoService(PedidoRepository repository) {
        this.repository = repository;
    }
    public void procesarPedido(Pedido pedido) {
        // Lógica de procesamiento del pedido
        repository.guardarPedido(pedido);
    }
}
```

De esta forma, conseguimos que la clase de alto nivel (PedidoService) sea independiente de los detalles de implementación de la clase de bajo nivel (PedidoRepository).

**Definición del Principio de Inversión de Dependencia**

El Principio de Inversión de Dependencia dice: "depende de abstracciones y no de implementaciones concretas".

Así, se recomienda que los módulos de alto nivel no dependan directamente de los detalles de implementación de los módulos de bajo nivel.

En su lugar, deben depender de abstracciones o interfaces que definan contratos de funcionamiento. Esto promueve una mayor flexibilidad y facilita el mantenimiento del sistema.

Por ejemplo, la funcionalidad de cualquier equipo electrónico está garantizada por la conexión adecuada entre el enchufe y la toma de corriente, ¿no es así?

Siguiendo esta analogía, los módulos de alto nivel representan el enchufe, mientras que los módulos de bajo nivel corresponden al tomacorriente.

Al igual que un enchufe se conecta a una toma de corriente independientemente de sus detalles internos, los módulos de alto nivel deben vincularse a abstracciones o interfaces, estableciendo contratos de funcionamiento.

Este enfoque es similar a utilizar un enchufe estándar, lo que garantiza una conexión flexible y un mantenimiento más fácil.

**Ventajas de aplicar el Principio de Inversión de Dependencia**

Seguir el Principio de Inversión de Dependencia promueve la flexibilidad y extensibilidad de nuestros sistemas.

Esto facilita la realización de pruebas unitarias y ayuda a construir códigos más robustos y duraderos.

**Ahora que hemos cubierto todos los conceptos de SOLID con ejemplos prácticos, veamos cómo este principio está relacionado con el Clean Code.**

**Clean Code**

En el mundo de la programación, frecuentemente encontramos el término "Clean Code" o "Código Limpio".

Pero ¿qué es exactamente un "código limpio"? ¿Qué características son necesarias para lograrlo?

Escribir un código limpio significa escribirlo de tal manera que sea fácil de entender sin complicaciones.

Esto no solo simplifica la manipulación del código, sino que también facilita la colaboración dentro del equipo. Al final, todo el desarrollo y mantenimiento del sistema se vuelve más fácil.

De acuerdo con "Uncle Bob", en su libro "Código Limpio: Habilidades Prácticas del Software Ágil", existen algunas buenas prácticas fundamentales para alcanzar la claridad del código.

Conozcámoslas a continuación:

**1. Utilizar los principios SOLID**

El Clean Code y los principios SOLID comparten el objetivo de mejorar la calidad del software, haciéndolo legible, organizado, extensible y fácil de mantener.

**2. Usar nombres significativos**

Los nombres descriptivos ayudan a entender la finalidad de una parte del código sin necesidad de comentarios explicativos.

Para ilustrar, considera el siguiente código:

```
public static double conv(double tC) {
    double tF = (tC * 9 / 5) + 32;
```

```
    return tF;
}
```

Es necesario hacer un esfuerzo para entender qué hace este código. Podemos mejorar el entendimiento simplemente agregando nombres significativos a las variables y al método:

```
public static double convertirCelsiusAFahrenheit(double temperaturaCelsius) {
    double temperaturaFahrenheit = (temperaturaCelsius * 9 / 5) + 32;
    return temperaturaFahrenheit;
}
```

Ahora está claro cuál es el propósito del código sin necesidad de recordar fórmulas o realizar investigaciones adicionales. Esto ahorra tiempo y evita confusiones innecesarias.

### 3. Priorizar el uso de funciones pequeñas

Escribir métodos o funciones pequeñas y centradas en una sola tarea es fundamental para mantener el código claro y seguir el Principio de Responsabilidad Única (SRP).

Considera el siguiente código:

```
public class Main {
    public static void main(String[] args) {
        int[] numeros = {1, 2, 3, 4, 5};
        int suma = 0;
        for (int numero : numeros) {
            suma += numero;
        }
        double promedio = (double) suma / numeros.length;
        if (promedio > 3) {
            System.out.println("El promedio es mayor que 3");
        } else {
            System.out.println("El promedio es menor o igual a 3");
        }
    }
}
```

Aunque el código usa nombres descriptivos, la legibilidad podría mejorar dividiendo las tareas en funciones separadas, cada una con su propia descripción. Por ejemplo:

```
public class Main {
    public static void main(String[] args) {
        int[] numeros = {1, 2, 3, 4, 5};
        int suma = calcularSuma(numeros);
        double promedio = calcularPromedio(numeros);
        verificarYMostrarResultado(promedio);
    }
    public static int calcularSuma(int[] numeros) {
        int suma = 0;
        for (int numero : numeros) {
            suma += numero;
        }
        return suma;
```

```
    }
    public static double calcularPromedio(int[] numeros) {
        return (double) calcularSuma(numeros) / numeros.length;
    }
    public static void verificarYMostrarResultado(double promedio) {
        if (promedio > 3) {
            System.out.println("El promedio es mayor que 3");
        } else {
            System.out.println("El promedio es menor o igual a 3");
        }
    }
}
```

Aunque el código es más extenso, ganamos en legibilidad y segmentación. Cualquier persona que necesite modificar cómo se muestra el promedio a la persona usuaria solo necesitará cambiar el método verificarYMostrarResultado.

Esto demuestra cómo las funciones pequeñas pueden facilitar el mantenimiento y la comprensión del código.

**4. Evitar comentarios innecesarios**

El código debe ser autoexplicativo, con nombres significativos y una estructura lógica clara. Los comentarios excesivos pueden hacer que el código se vea desordenado y difícil de mantener.

Considera el siguiente código:

```
public class R {
    private double w;
    private double h;
    // Método para calcular el área
    public double calc() {
        return w * h;
    }
}
```

Los nombres cortos para las variables dificultan la comprensión, lo que obliga a utilizar comentarios en el código, lo que lo ensucia. Podemos resolverlo agregando nombres descriptivos y eliminando los comentarios:

```
public class Rectangulo {
    private double ancho;
    private double alto;
    public Rectangulo(double ancho, double alto) {
        this.ancho = ancho;
        this.alto = alto;
    }
    public double calcularArea() {
        return ancho * alto;
    }
}
```

Ahora es mucho más fácil entenderlo. Cualquier desarrollador que lea este código podrá asimilar fácilmente lo que hace cada parte.

### 5. Evitar la complejidad innecesaria

La complejidad innecesaria aumenta la probabilidad de errores y dificulta el mantenimiento del código. Un ejemplo de código complejo para hacer algo simple, como sumar dos números, sería:

```
public void sumar() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ingrese el primer número: ");
    String num1String = scanner.nextLine();
    System.out.print("Ingrese el segundo número: ");
    String num2String = scanner.nextLine();
    boolean entradaValida = false;
    double num1 = 0;
    double num2 = 0;
    while (!entradaValida) {
        num1 = Double.parseDouble(num1String);
        num2 = Double.parseDouble(num2String);
        entradaValida = true;
    }
    double suma = num1 + num2;
    System.out.println("La suma de los números es: " + suma);
    scanner.close();
}
```

Hemos hecho una verificación de la entrada, solo para luego convertirla a double.

Podríamos simplemente asumir que el usuario ingresará un double y manejar las excepciones relacionadas con ello:

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class SumaSimpleConManejoDeErrores {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Ingrese el primer número: ");
            double num1 = scanner.nextDouble();
            System.out.print("Ingrese el segundo número: ");
            double num2 = scanner.nextDouble();
            double suma = num1 + num2;
            System.out.println("La suma de los números es: " + suma);
        } catch (InputMismatchException e) {
            System.out.println("Error: Por favor, ingrese números válidos.");
        } finally {
            scanner.close();
        }
    }
}
```

Ejecutamos la lógica deseada de forma rápida y fácil de entender. Imagina que ahora necesitamos sumar tres variables en lugar de dos.

Es mucho más fácil modificar el segundo código que el primero. El segundo código es un ejemplo de código limpio.

### 6. Reducir la cantidad de argumentos

Las funciones y métodos deben tener la menor cantidad de argumentos posible. Esto mejora la legibilidad y la facilidad de uso.

En el siguiente ejemplo, observa cómo registrar a un empleado es complejo debido a la gran cantidad de parámetros:

```
public static void registrarEmpleado(String nombre, int edad, String cargo, double salario, String direccion, String ciudad, String codigoPostal, String telefono, String correoElectronico) {
    // Lógica de registro del empleado aquí...
}
```

Al llamar a este método, es difícil entender qué parámetro usar en cada lugar, lo que puede llevar a confusiones.

Una buena alternativa sería crear una clase para representar al empleado, otra para la dirección y otra para el contacto. Así, podríamos simplificar:

```
public static void registrarEmpleado(Empleado empleado, Direccion direccion, Contacto contacto) {
}
```

De esta forma, agrupamos la información para usar menos argumentos, lo que es una buena práctica.

### 7. Evitar la repetición de código

La repetición de código dificulta el mantenimiento, ya que si hay cambios necesarios, deben aplicarse en múltiples lugares.

Extrae el código repetido en funciones o métodos para promover la reutilización y el mantenimiento eficiente.

Considera el siguiente ejemplo:

```
public static void main
(String[] args) {
    int numero1 = 5;
    int numero2 = 7;
    // Cálculo del factorial para el primer número
    int resultado1 = 1;
    for (int i = 1; i <= numero1; i++) {
        resultado1 *= i;
    }
    System.out.println("Factorial de " + numero1 + ": " + resultado1);
    // Cálculo del factorial para el segundo número
    int resultado2 = 1;
    for (int i = 1; i <= numero2; i++) {
        resultado2 *= i;
    }
    System.out.println("Factorial de " + numero2 + ": " + resultado2);
}
```

Como calculamos el factorial más de una vez, extraemos el código a una función para evitar la repetición:

```
public static void main(String[] args) {
    int numero1 = 5;
    int numero2 = 7;
    // Cálculo e impresión del factorial para el primer número
    calcularEImprimirFactorial(numero1);
    // Cálculo e impresión del factorial para el segundo número
    calcularEImprimirFactorial(numero2);
}
public static void calcularEImprimirFactorial(int numero) {
    int resultado = 1;
    for (int i = 1; i <= numero; i++) {
        resultado *= i;
    }
    System.out.println("Factorial de " + numero + ": " + resultado);
}
```

De esta manera, si necesitamos calcular la factorial de otro número, no tendremos que repetir el código. Simplemente llamamos nuevamente a la función para calcular la factorial. Esto facilita el desarrollo.

**Ventajas de escribir código limpio**

Al implementar cada uno de estos principios y prácticas, no solo mejorarás la calidad de tu código, sino que también facilitarás su comprensión, mantenimiento y la colaboración en el desarrollo de software. Todo lo que hemos discutido aquí se resume en una frase de Martin Fowler:

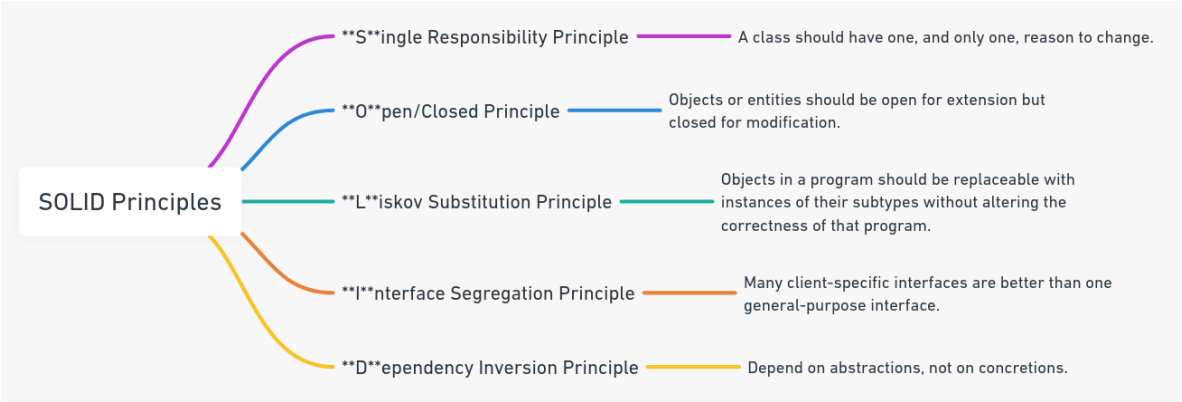
*"Cualquier tonto puede escribir código que una computadora pueda entender. Los buenos programadores escriben código que los humanos puedan entender."*

La programación no se trata solo de hacer que la máquina funcione, sino también de crear soluciones que sean comprensibles y colaborativas.

## VI. CASOS PROPUESTOS:

Revisar lo siguiente:

**SOLID Mind Map:**




1. **Puertos USB en computadoras** 🔌: Los puertos USB son un ejemplo perfecto de OCP. Permiten conectar varios dispositivos (extensión) sin cambiar el hardware de la computadora (modificación). De igual manera, en programación, OCP fomenta el diseño de módulos que se puedan ampliar con nuevas funciones sin modificar el código existente.

2. **Aplicaciones para smartphones** 📱: Considere cómo se actualizan las aplicaciones para smartphones. Los desarrolladores añaden funciones o corrigen errores sin necesidad de modificar el teléfono (el sistema existente). Esto refleja OCP, donde el software puede evolucionar y crecer sin alterar el sistema central.

3. **Televisión con diferentes canales** 📺: Un televisor está diseñado para mostrar diferentes canales sin necesidad de modificar sus circuitos internos para cada nuevo canal (extensión sin modificación). En software, esto es similar a diseñar sistemas que admitan nuevas funciones sin alterar la lógica central.

4. **Tomas de corriente y adaptadores** 🔌: Las tomas de corriente están diseñadas para suministrar electricidad a varios dispositivos. Con adaptadores, pueden alimentar dispositivos de diferentes países, ampliando su funcionalidad sin modificar la toma. En software, esto equivale a crear módulos que se pueden ampliar (por ejemplo, mediante plugins o extensiones) sin modificar el módulo original.

5. **Ropa con bolsillos** 👖: La ropa se diseña con bolsillos, que permiten guardar diversos artículos. El diseño de la ropa permanece igual (cerrado a modificaciones), pero su funcionalidad se extiende al permitir guardar diferentes artículos (abierto a extensiones). En programación, esto sería como una clase que puede gestionar diferentes tipos de datos sin necesidad de modificar su estructura.

**Actividades:**

En grupo de hasta 3 integrantes, crear un ejemplo concreto de principios SOLID en cualquier **lenguaje** y subir al classroom.

**NOTA**: Se puede poner capturas con detalles de cada código o comentarios legibles, agregar conclusiones y recomendaciones.

**RÚBRICA:**

| Inicio<br>00-10 | Proceso<br>11-13 | Logro previsto<br>14-17 | Logro satisfactorio<br>18-20 |
|---|---|---|---|
| Desarrolla correctamente las prácticas en el laboratorio hasta un 50% | Desarrolla correctamente las prácticas en el laboratorio hasta un 60% | Desarrolla correctamente las prácticas en el laboratorio hasta un 80% | Desarrolla correctamente las prácticas en el laboratorio hasta un 100% |

---

**Bibliografía:**

https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start