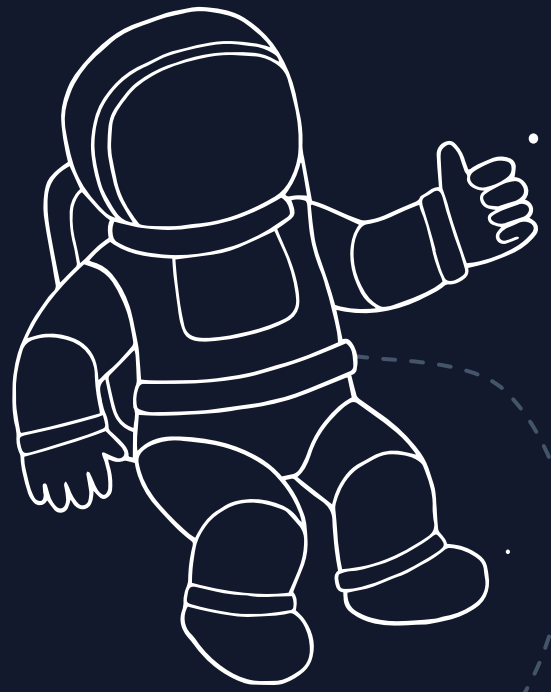
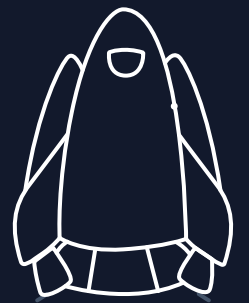


Programa académico CAMPUS

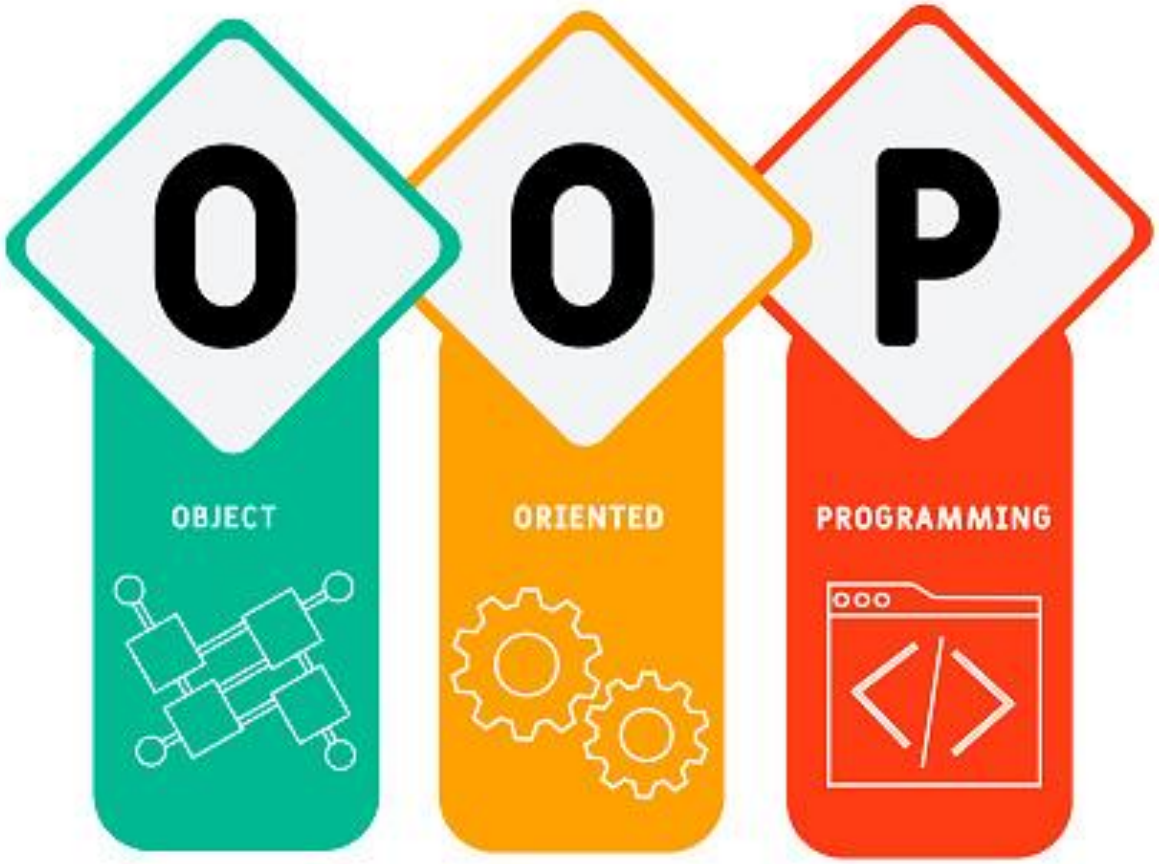
MODULO JAVA
Sesión 10

Principios SOLID

Trainer Carlos H. Rueda C.

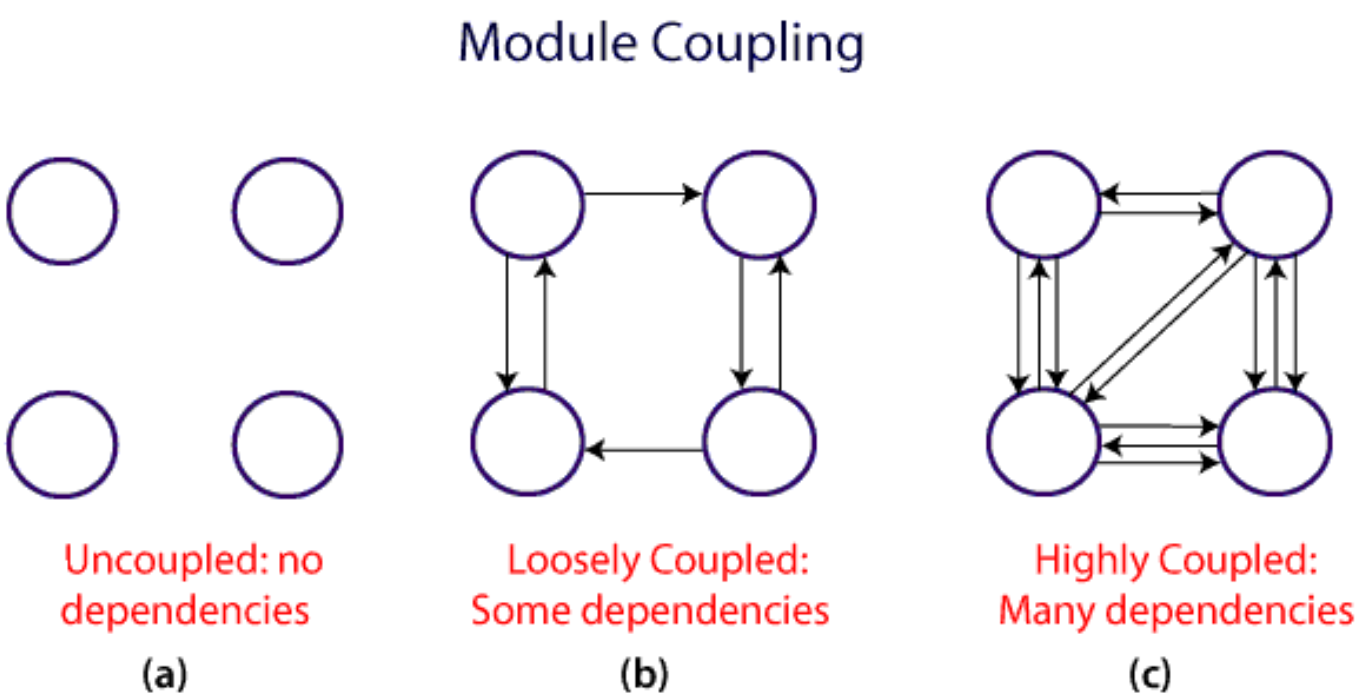



Java™


O
O
P
OBJECT
ORIENTED
PROGRAMMING

ACOPLAMIENTO EN DESARROLLO DE SOFTWARE

El **acoplamiento** en el desarrollo de software se refiere al grado de dependencia entre los módulos o componentes de un sistema. Un alto acoplamiento significa que los módulos están fuertemente interconectados, lo que dificulta su modificación, mantenimiento o reutilización. En contraste, un bajo acoplamiento indica que los módulos están más independientes, lo que mejora la flexibilidad y escalabilidad del sistema.



ACOPLAMIENTO EN DESARROLLO DE SOFTWARE

Alto acoplamiento

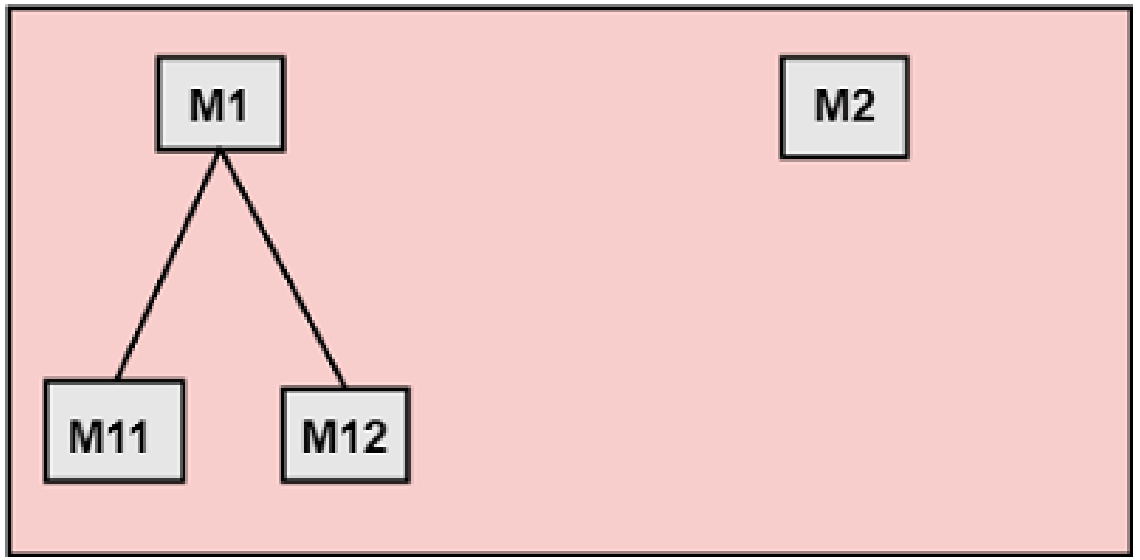
```
public class Pedido {
    private Factura factura;

    public Pedido() {
        factura = new Factura(); // Pedido depende directamente de Factura
    }
}
```

Bajo acoplamiento

```
public class Pedido {
    private Factura factura;

    public Pedido(Factura factura) { // Pedido depende de una interfaz o recibe la ins
        this.factura = factura;
    }
}
```

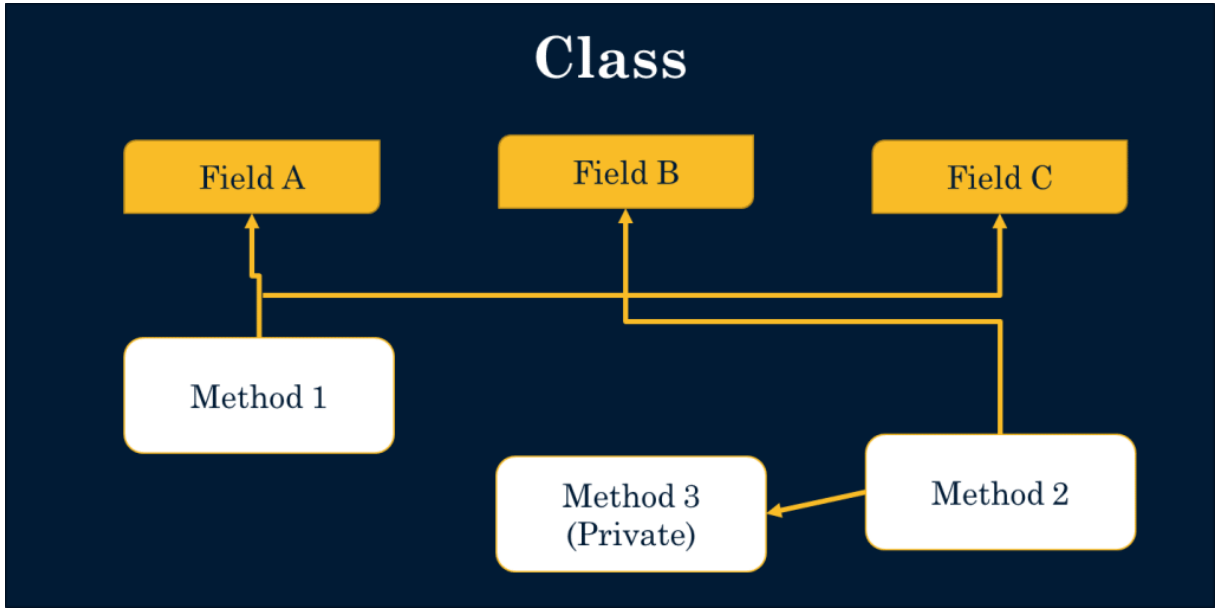
COHESIÓN EN DESARROLLO DE SOFTWARE

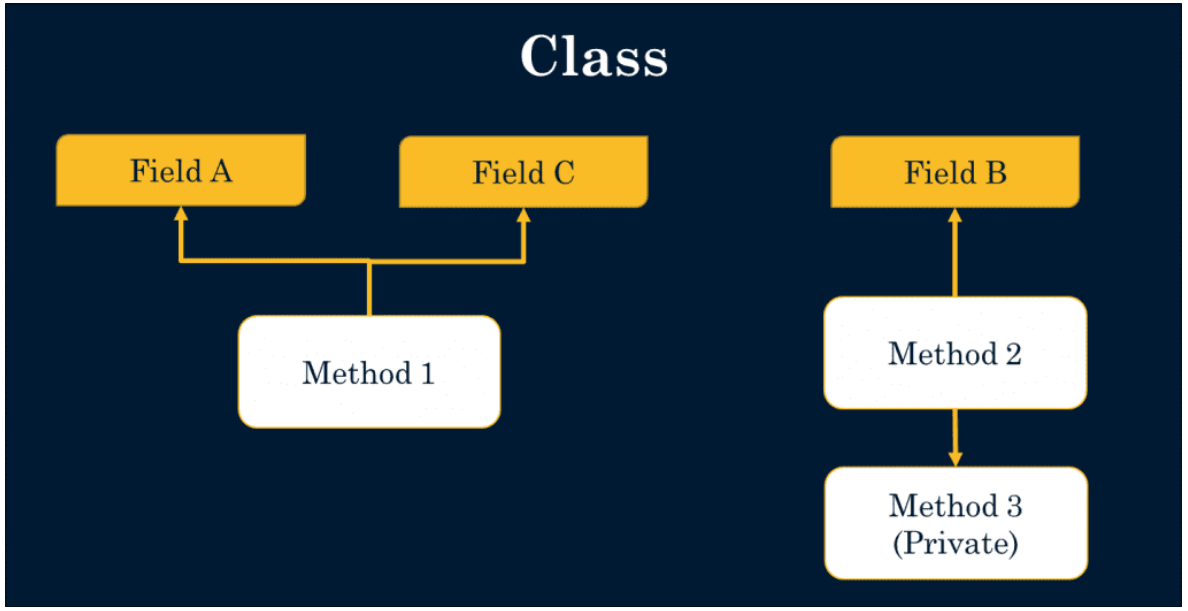
La **cohesión** en el desarrollo de software mide qué tan bien están relacionadas las responsabilidades o funcionalidades dentro de un módulo o componente. Una alta cohesión significa que un módulo se centra en una sola tarea o propósito, lo que facilita su comprensión, mantenimiento y reutilización. Una baja cohesión indica que un módulo realiza múltiples tareas no relacionadas, lo que puede llevar a un diseño confuso y difícil de mantener.

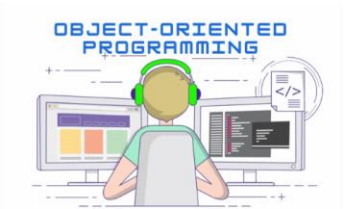


COHESIÓN EN DESARROLLO DE SOFTWARE

Baja Cohesión

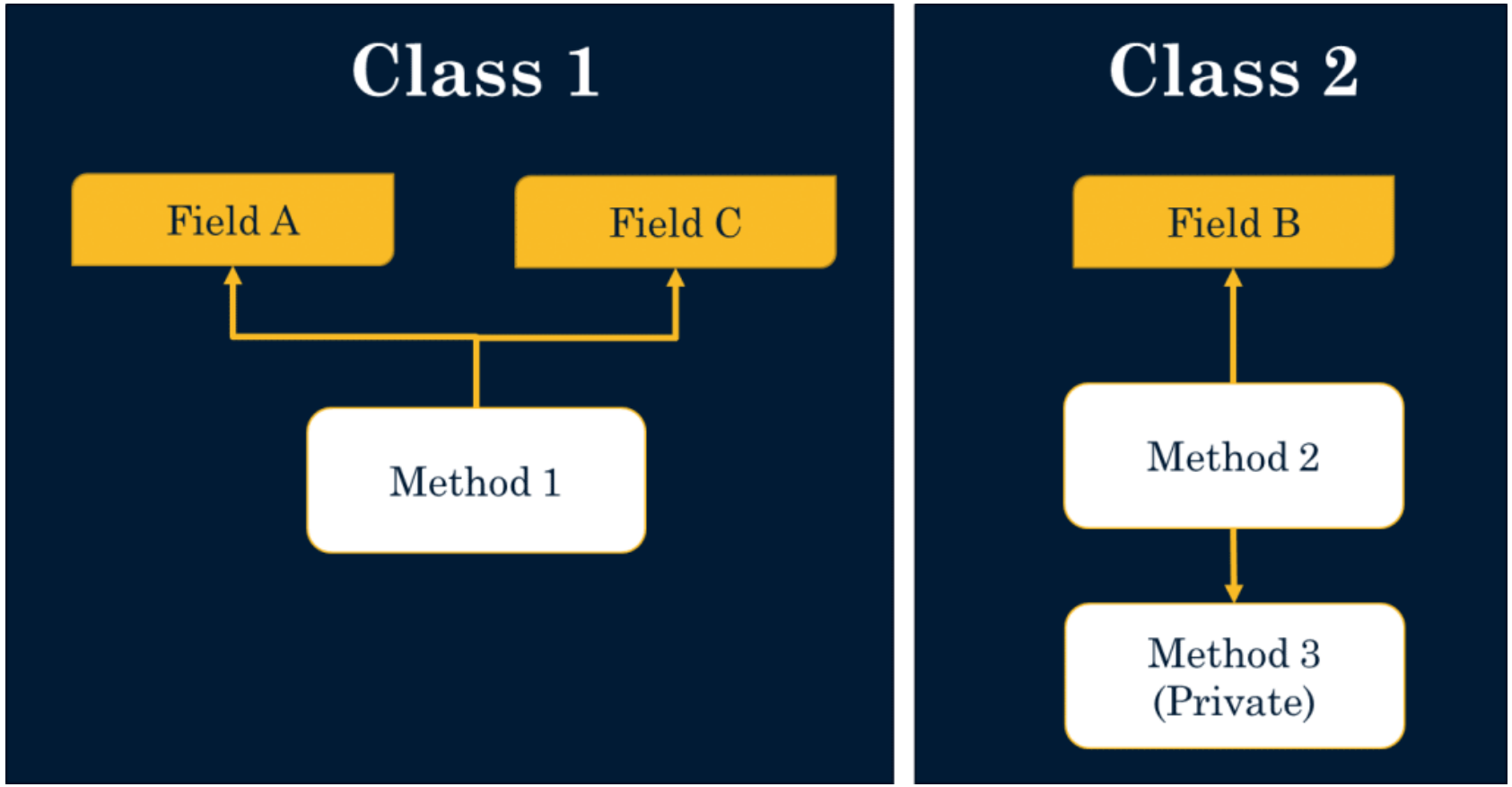






COHESIÓN EN DESARROLLO DE SOFTWARE

Alta Cohesión



COHESIÓN EN DESARROLLO DE SOFTWARE

Baja Cohesión

```
public class Utilidades {
    public void calcularDescuento() {
        // Cálculo del descuento
    }

    public void enviarCorreo() {
        // Envío de un correo
    }

    public void generarReporte() {
        // Generar un reporte
    }
}
```

Aquí, `Utilidades` realiza múltiples tareas no relacionadas, lo que dificulta su mantenimiento.

COHESIÓN EN DESARROLLO DE SOFTWARE

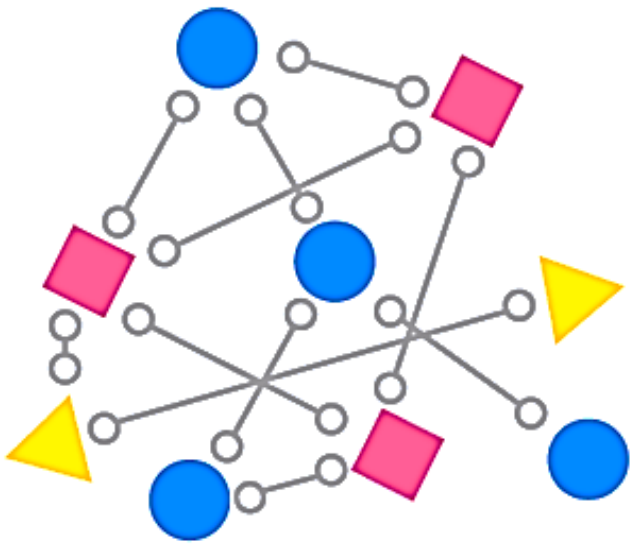
Alta Cohesión

```
public class Descuento {
    public void calcular() {
        // Cálculo del descuento
    }
}

public class NotificacionCorreo {
    public void enviarCorreo() {
        // Envío de un correo
    }
}

public class Reporte {
    public void generar() {
        // Generar un reporte
    }
}
```

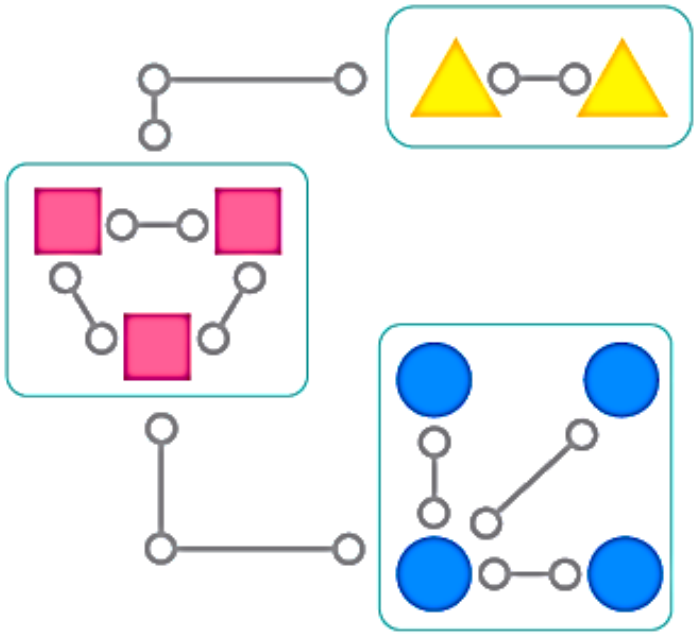
Cada clase tiene un propósito claro y está enfocada en una sola responsabilidad, aumentando la cohesión.

COHESIÓN EN DESARROLLO DE SOFTWARE

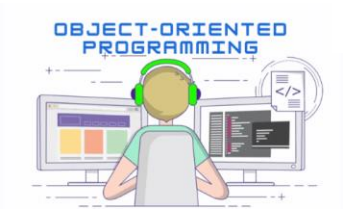
Cohesion + Coupling



Without

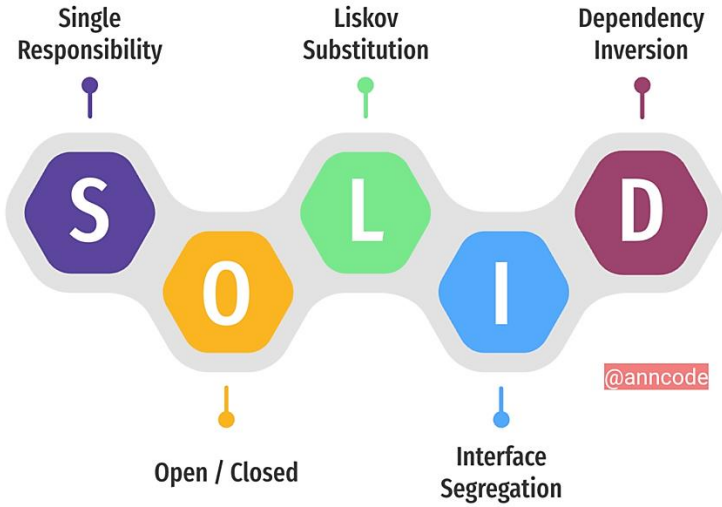


With



PRINCIPIOS SOLID

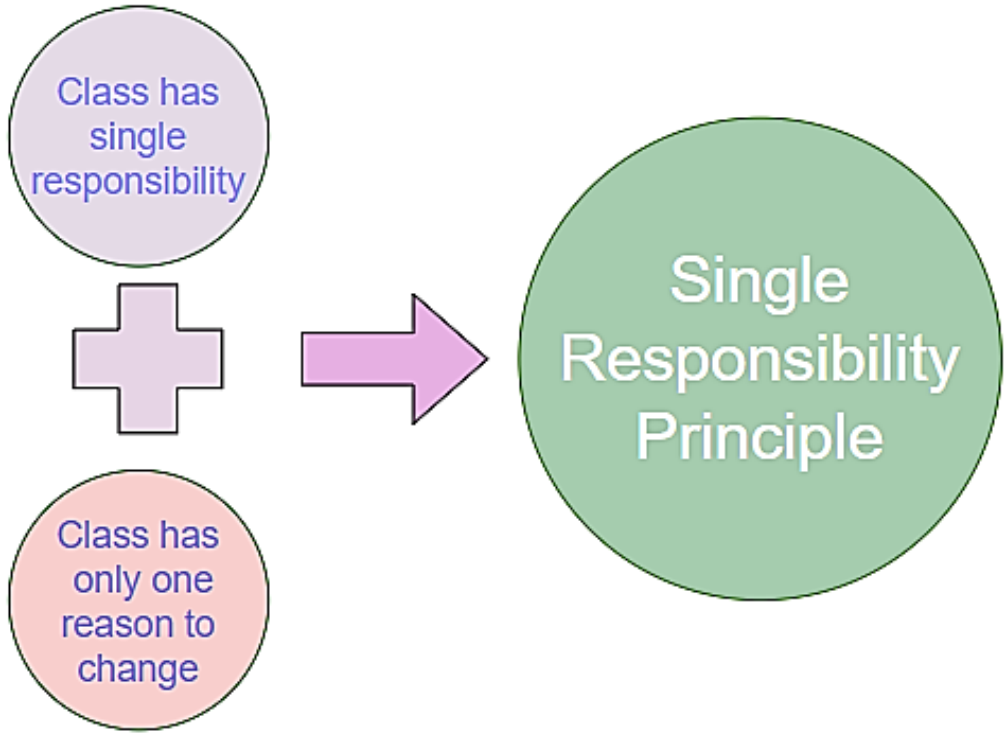
Si se habla de diseño y desarrollo de aplicaciones, es fundamental conocer los Principios SOLID, que constituyen la base de la arquitectura y desarrollo de software. SOLID es un acrónimo basado en los principios de la programación orientada a objetos recopilados por Robert C. Martin en 2000 en su paper *"Design Principles and Design Patterns"*, y fue acuñado por Michael Feathers. Posteriormente, Robert C. Martin continuó agregando consejos y buenas prácticas de desarrollo en su libro *"Clean Code"*, convirtiéndose en una referencia para desarrolladores.



Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

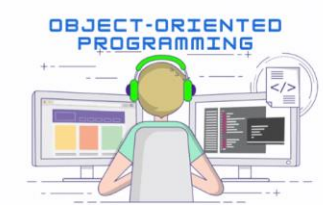
Un módulo o clase debe tener una, y solo una, razón para cambiar, lo que significa que debe tener **una sola** responsabilidad o propósito.

*Una clase debe **enfocarse** en una sola funcionalidad específica*.



Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

Supongamos que estamos desarrollando una aplicación para la gestión de empleados. En un mal diseño, podríamos tener una clase `Employee` que no solo gestione la información del empleado, sino que también maneje la lógica de persistencia (guardar, actualizar y eliminar empleados en una base de datos).



Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

Mal diseño

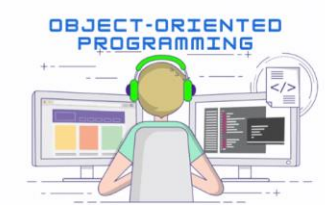
```java
public class Employee {
    private String id;
    private String name;
    private double salary;

    public Employee(String id, String name, double salary) {
        this.id = id; this.name = name; this.salary = salary;
    }

    public void save() {
    // Código para guardar el empleado en la base de datos
    }

    public void update() {
        // Código para actualizar el empleado en la base de datos
    }

    public void delete() {
    // Código para eliminar el empleado de la base de datos
    }
}
```

Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

Buen diseño

Para adherirse al SRP, debemos separar las responsabilidades en clases distintas:

1. **Clase Employee**: Solo maneja la información del empleado.
2. **Clase EmployeeRepository**: Maneja la lógica de persistencia del empleado.

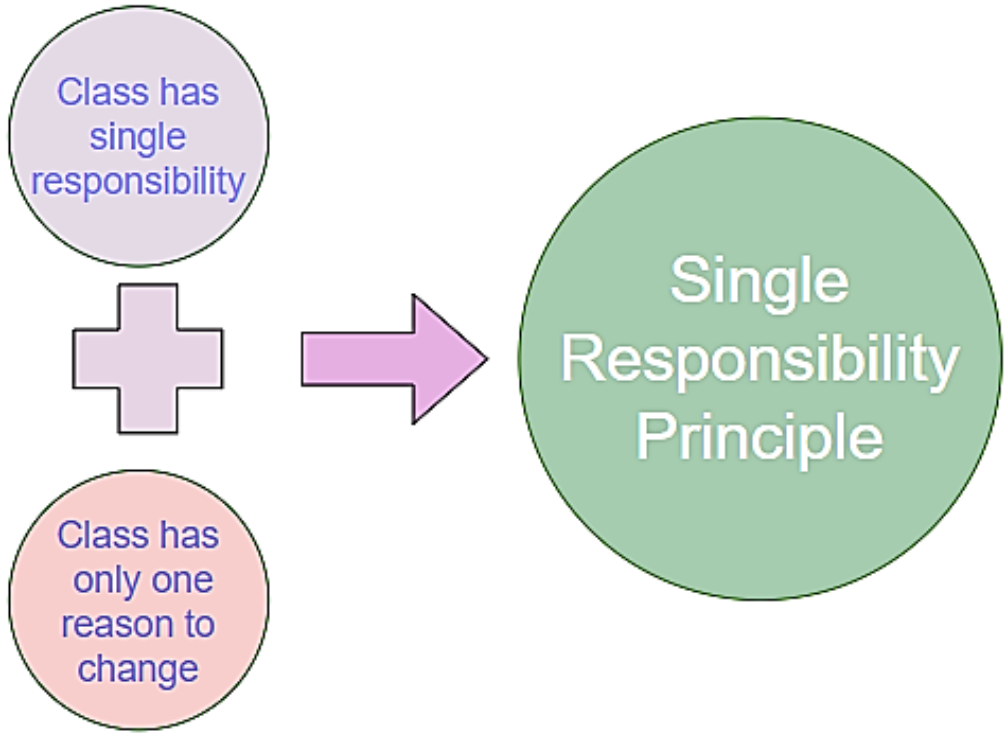


Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

Buen diseño

Para adherirse al SRP, debemos separar las responsabilidades en clases distintas:

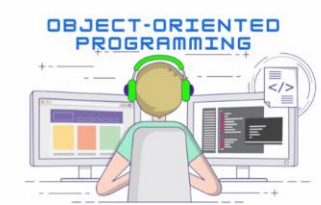
1. **Clase Employee**: Solo maneja la información del empleado.
2. **Clase EmployeeRepository**: Maneja la lógica de persistencia del empleado.



Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

Buen diseño

```java
// Clase Employee que maneja solo la información del empleado
public class Employee {
    private String id;
    private String name;
    private double salary;

    public Employee(String id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    // Getters y setters
}
```



Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

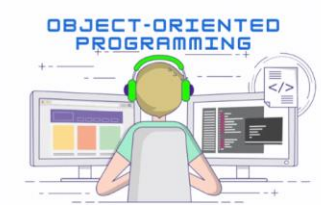
Buen diseño

```java
// Clase EmployeeRepository que maneja la persistencia de los empleados
public class EmployeeRepository {
    public void save(Employee employee) {
        // Código para guardar el empleado en la base de datos
    }

    public void update(Employee employee) {
        // Código para actualizar el empleado en la base de datos
    }

    public void delete(Employee employee) {
        // Código para eliminar el empleado de la base de datos
    }
}
```

Single Responsibility Principle (SRP)
Principio de Responsabilidad Única

Buen diseño

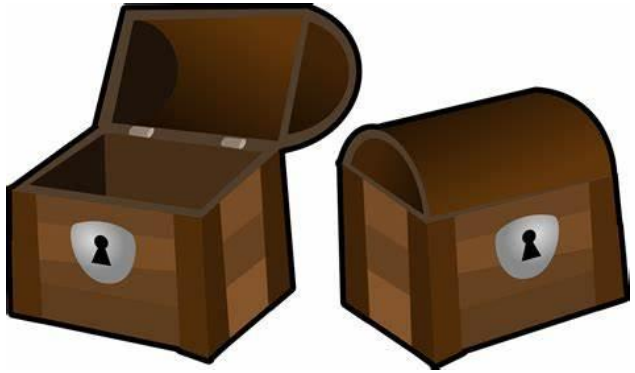
```java
public class Main {
    public static void main(String[] args) {
        Employee employee = new Employee("1", "John Doe", 50000);
        EmployeeRepository repository = new EmployeeRepository();

        repository.save(employee);
        // Otros métodos de repository como update() y delete()
    }
}
```



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

Este principio establece que las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para su extensión pero cerradas para su modificación. En otras palabras, debes poder añadir nuevas funcionalidades a una entidad de software existente sin modificar su código fuente.

Deberías poder agregar nuevas funcionalidades sin alterar el código existente.



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

Mal diseño

```java
public class Employee {
    private String id;
    private String name;
    private double baseSalary;
    private String type; // "FULL_TIME" or "CONTRACTOR"

    public Employee(String id, String name, double baseSalary, String
        type) {
        this.id = id; this.name = name;
        this.baseSalary = baseSalary; this.type = type;
    }
}

public class SalaryCalculator {
    public double calculateSalary(Employee employee) {
        if (employee.getType().equals("FULL_TIME"))
            return employee.getBaseSalary();
        else if (employee.getType().equals("CONTRACTOR"))
            return employee.getBaseSalary() * 0.8;
        return 0;
    }
}
```

Este diseño ***viola el OCP*** porque cada vez que se añade un nuevo tipo de empleado, ***se necesita modificar*** la clase `SalaryCalculator`.

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

Supongamos que estamos desarrollando un *sistema de cálculo de salarios* para diferentes tipos de empleados (por ejemplo, empleados a tiempo completo y empleados contratados). Queremos que el sistema sea extensible sin necesidad de modificar el código existente cuando se agregue un nuevo tipo de empleado.

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

Buen diseño

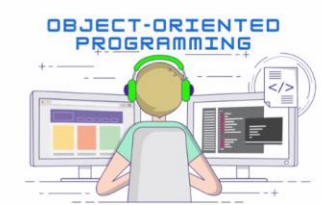
Para adherirse al OCP, se debe utilizar la herencia y el polimorfismo. Cada tipo de empleado debe tener su propia implementación del cálculo de salario.

- ✓ **Clase Employee**: Clase base abstracta para empleados.
- ✓ **Clases concretas**: FullTimeEmployee y ContractorEmployee que extienden Employee.
- ✓ **Interfaz SalaryCalculator**: Define el método de cálculo de salario.

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

Buen diseño

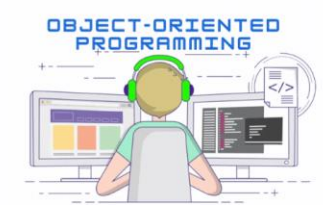
```java
// Clase base abstracta para empleados
public abstract class Employee {
    private String id;
    private String name;
    private double baseSalary;

    public Employee(String id, String name, double baseSalary) {
        this.id = id;
        this.name = name;
        this.baseSalary = baseSalary;
    }

    public double getBaseSalary() {
        return baseSalary;
    }

    // Método abstracto para calcular el salario
    public abstract double calculateSalary();
}
```



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

Buen diseño

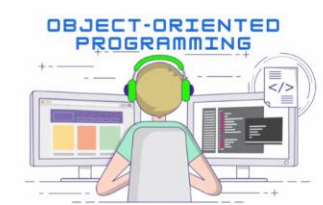
```java
// Empleado contratado
public class ContractorEmployee extends Employee {
    public ContractorEmployee(String id, String name, double baseSalary)
        {
        super(id, name, baseSalary);
    }

    @Override
    public double calculateSalary() {
        return getBaseSalary() * 0.8;
    }
}
```



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

Buen diseño

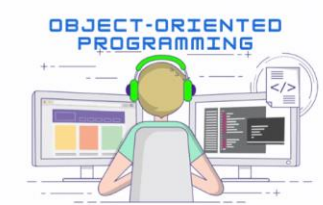
```java
// Clase para calcular el salario (interfaz opcional si es necesario)
public interface SalaryCalculator {
    double calculateSalary(Employee employee);
}

// Implementación de SalaryCalculator
public class DefaultSalaryCalculator implements SalaryCalculator {
    @Override
    public double calculateSalary(Employee employee) {
        return employee.calculateSalary();
    }
}
```



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

Buen diseño

```
public class Main {
    public static void main(String[] args) {
        Employee fullTimeEmployee =
                new FullTimeEmployee("1", "John Doe", 50000);
        Employee contractorEmployee =
                new ContractorEmployee("2", "Jane Smith", 40000);

        SalaryCalculator calculator =
                new DefaultSalaryCalculator();

        System.out.println("Full Time Employee Salary: "
                + calculator.calculateSalary(fullTimeEmployee));
        System.out.println("Contractor Employee Salary: "
                + calculator.calculateSalary(contractorEmployee));
    }
}
```

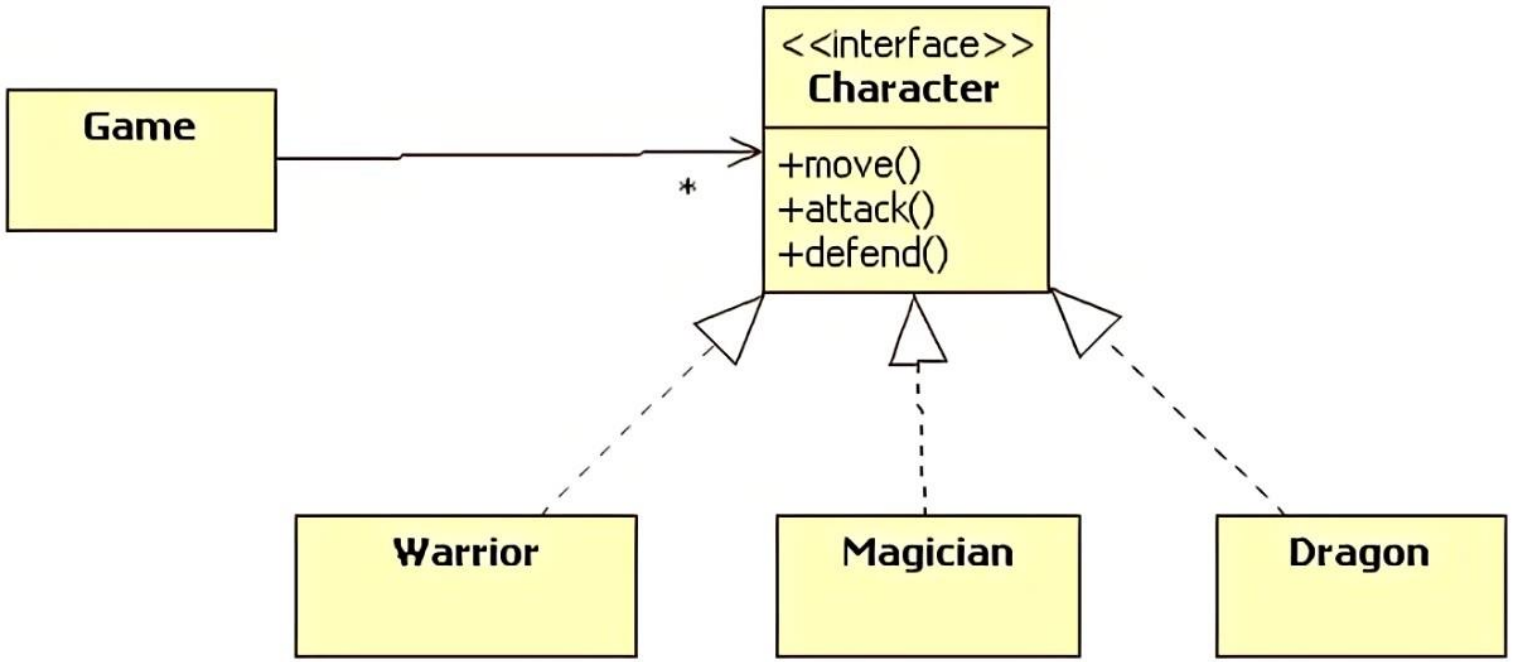


Open/Closed Principle (OCP)
Principio de Abierto/Cerrado

Beneficios del OCP

- **Extensibilidad**: Para agregar un nuevo tipo de empleado, solo necesitas crear una nueva clase que extienda `Employee` y sobrescribir el método `calculateSalary()`.
- **Mantenibilidad**: No es necesario modificar el código existente para soportar nuevos tipos de empleados, lo que reduce el riesgo de introducir errores.
- **Polimorfismo**: Aprovecha el polimorfismo para calcular los salarios de manera uniforme a través de diferentes tipos de empleados.

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

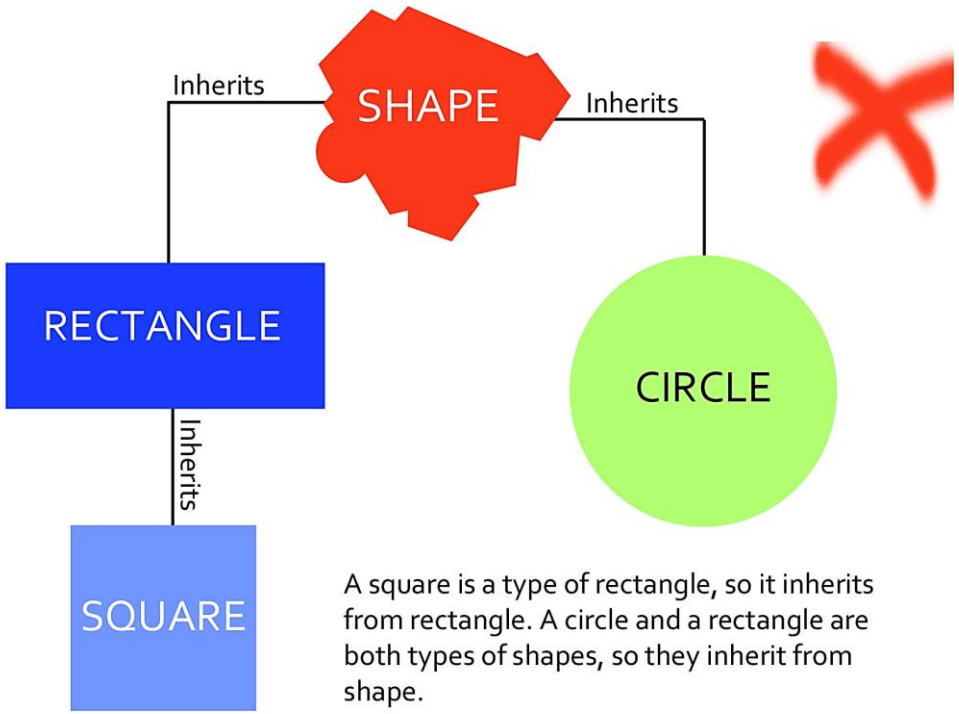


Liskov Substitution Principle (LSP)
Principio de Sustitución de Liskov

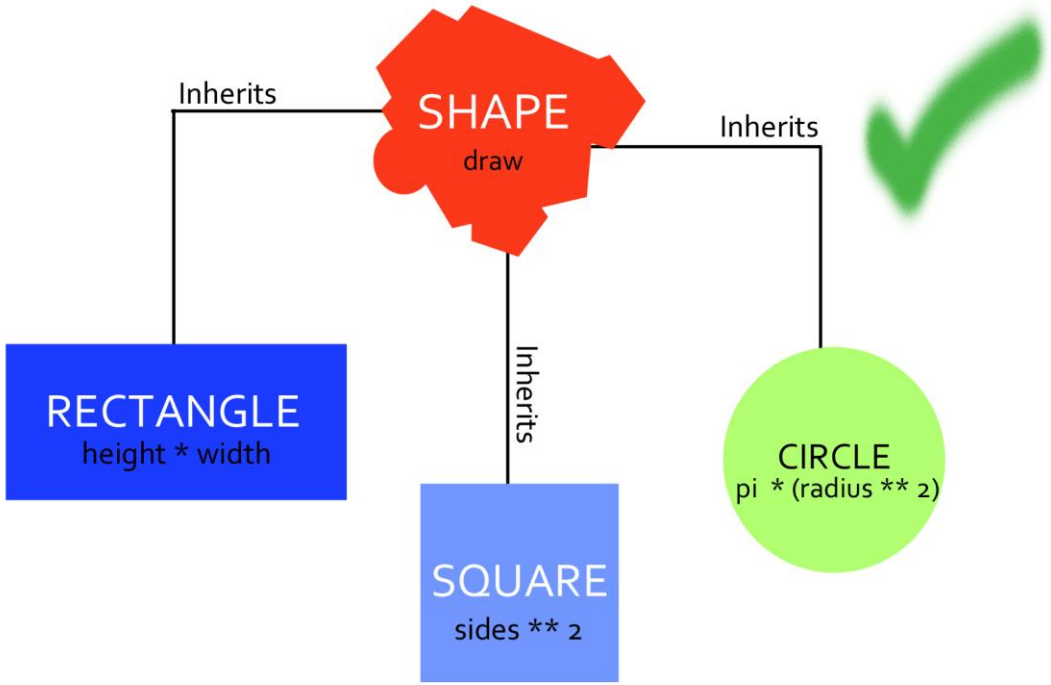
Este principio establece que los objetos de una clase derivada deben poder ser reemplazados por objetos de la clase base sin alterar el correcto funcionamiento del programa. Es decir, si S es una subclase de T, los objetos de tipo T en un programa deben poder ser reemplazados por objetos de tipo S sin afectar adversamente el comportamiento del programa.

Una subclase debe ser completamente intercambiable con su clase base.

Liskov Substitution Principle (LSP)
Principio de Sustitución de Liskov



A square is a type of rectangle, so it inherits from rectangle. A circle and a rectangle are both types of shapes, so they inherit from shape.



Liskov Substitution Principle (LSP)
Principio de Sustitución de Liskov

Supongamos que estamos desarrollando una aplicación para un zoológico y tenemos una clase `Bird` y sus subclases. Queremos asegurarnos de que cualquier subclase de `Bird` pueda ser usada donde sea que se use la clase `Bird` sin afectar la funcionalidad del programa.

Mal diseño

```java
public class Bird {
    public void fly() {
        System.out.println("Flying");
    }
}

public class Sparrow extends Bird {
    // Hereda el método fly() de Bird
}

public class Penguin extends Bird {
    @Override
    public void fly() {
        // Los pingüinos no pueden volar, esto viola el LSP
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}

public class Zoo {
    public void makeBirdFly(Bird bird) {
        bird.fly();
    }

    public static void main(String[] args) {
        Bird sparrow = new Sparrow();
        Bird penguin = new Penguin();

        Zoo zoo = new Zoo();
        zoo.makeBirdFly(sparrow); // Funciona bien
        zoo.makeBirdFly(penguin); // Lanza una excepción
    }
}
```

En este diseño, la clase Penguin viola el LSP porque su implementación del método fly() no es consistente con la clase base Bird. Esto puede llevar a comportamientos inesperados o errores en tiempo de ejecución.

Buen diseño

```java
// Clase base abstracta para todas las aves
public abstract class Bird {
    private String name;

    public Bird(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void move();
}
```

```java
// Clase para las aves que vuelan
public class FlyingBird extends Bird {
    public FlyingBird(String name) {
        super(name);
    }

    @Override
    public void move() {
        fly();
    }

    public void fly() {
        System.out.println(getName() + " is flying");
    }
}

// Clase para las aves que no vuelan
public class NonFlyingBird extends Bird {
    public NonFlyingBird(String name) {
        super(name);
    }

    @Override
    public void move() {
        walk();
    }

    public void walk() {
        System.out.println(getName() + " is walking");
    }
}
```

Buen diseño

```java
// Subclases específicas de aves
public class Sparrow extends FlyingBird {
    public Sparrow(String name) {
        super(name);
    }
}

public class Penguin extends NonFlyingBird {
    public Penguin(String name) {
        super(name);
    }
}
```

Para adherirse al LSP, podemos ***introducir una jerarquía de clases más adecuada*** *que refleje correctamente las capacidades de las diferentes aves.*

```java
// Clase Zoo que utiliza la clase Bird
public class Zoo {
    public void makeBirdMove(Bird bird) {
        bird.move();
    }

    public static void main(String[] args) {
        Bird sparrow = new Sparrow("Sparrow");
        Bird penguin = new Penguin("Penguin");

        Zoo zoo = new Zoo();
        zoo.makeBirdMove(sparrow); // Sparrow is flying
        zoo.makeBirdMove(penguin); // Penguin is walking
    }
}
```

Buen diseño

```java
// Subclases específicas de aves
public class Sparrow extends FlyingBird {
    public Sparrow(String name) {
        super(name);
    }
}

public class Penguin extends NonFlyingBird {
    public Penguin(String name) {
        super(name);
    }
}
```

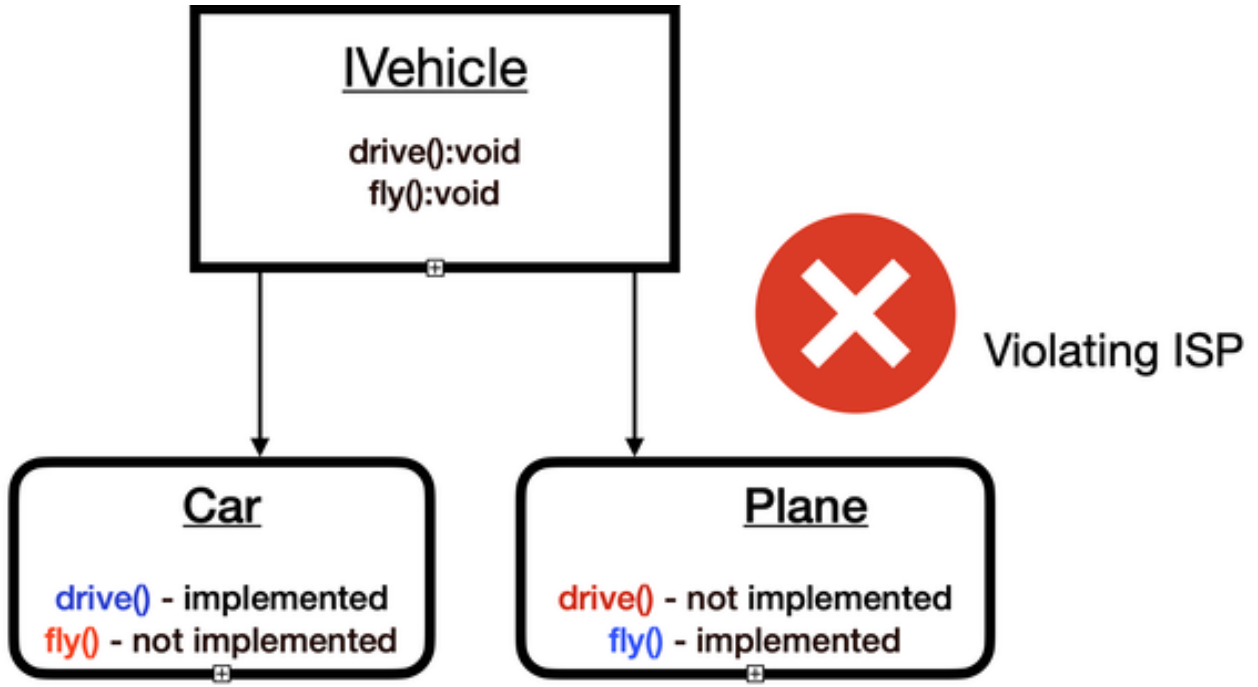
Para adherirse al LSP, podemos ***introducir una jerarquía de clases más adecuada*** *que refleje correctamente las capacidades de las diferentes aves.*

```java
// Clase Zoo que utiliza la clase Bird
public class Zoo {
    public void makeBirdMove(Bird bird) {
        bird.move();
    }

    public static void main(String[] args) {
        Bird sparrow = new Sparrow("Sparrow");
        Bird penguin = new Penguin("Penguin");

        Zoo zoo = new Zoo();
        zoo.makeBirdMove(sparrow); // Sparrow is flying
        zoo.makeBirdMove(penguin); // Penguin is walking
    }
}
```

Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

Este principio establece que los clientes no deberían verse obligados a depender de interfaces que no utilizan. Es mejor tener muchas interfaces específicas para clientes en lugar de una interfaz general.

Es mejor tener varias interfaces pequeñas y específicas que una general y extensa.



Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

Mal diseño (violación del ISP)

Supongamos que tenemos una interfaz `Worker` que define métodos para diferentes tipos de trabajadores en una empresa, incluidos los métodos para tareas que algunos trabajadores no realizan:

```
public interface Worker {
    void work();
    void eat();
    void sleep();
}
```

Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

Mal diseño

```java
public interface Worker {
    void work();
    void eat();
    void sleep();
}
```

```java
public class Developer implements Worker {
    @Override
    public void work() {
        System.out.println("Developing software");
    }

    @Override
    public void eat() {
        System.out.println("Eating lunch");
    }

    @Override
    public void sleep() {
        System.out.println("Sleeping at home");
    }
}
```

Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

Mal diseño

```java
public interface Worker {
    void work();
    void eat();
    void sleep();
}
```

*En este diseño, la interfaz `Worker` **fuerza** a que todas las implementaciones **incluyan métodos que no son aplicables** a ciertos tipos de trabajadores, como los robots.*

```java
public class Robot implements Worker {
    @Override
    public void work() {
        System.out.println("Assembling parts");
    }

    @Override
    public void eat() {
        // Los robots no comen, esto no tiene sentido
        throw new UnsupportedOperationException("Robots don't eat");
    }

    @Override
    public void sleep() {
        // Los robots no duermen, esto no tiene sentido
        throw new UnsupportedOperationException("Robots don't sleep");
    }
}
```

Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

Para adherirse al ISP, podemos dividir la interfaz `Worker` en varias interfaces más específicas que agrupen métodos relacionados.

Buen diseño

```
// Interfaz para trabajar
public interface Workable {
    void work();
}

// Interfaz para comer
public interface Eatable {
    void eat();
}

// Interfaz para dormir
public interface Sleepable {
    void sleep();
}
```

Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

```java
// Interfaz para trabajar
public interface Workable {
    void work();
}

// Interfaz para comer
public interface Eatable
    void eat();
}

// Interfaz para dormir
public interface Sleepabl
    void sleep();
}
```

```java
// Clase para desarrolladores que implementan todas las interfaces necesarias
public class Developer implements Workable, Eatable, Sleepable {
    @Override
    public void work() {
        System.out.println("Developing software");
    }

    @Override
    public void eat() {
        System.out.println("Eating lunch");
    }

    @Override
    public void sleep() {
        System.out.println("Sleeping at home");
    }
}
```

Buen diseño

Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

```java
// Interfaz para trabajar
public interface Workable {
    void work();
}

// Interfaz para comer
public interface Eatable {
    void eat();
}

// Interfaz para dormir
public interface Sleepabl
    void sleep();
}
```

```java
// Clase para robots que solo implementan la interfaz Workable
public class Robot implements Workable {
    @Override
    public void work() {
        System.out.println("Assembling parts");
    }
}
```

Buen diseño

Buen diseño

```java
// Interfaz para trabajar
public interface Workable {
    void work();
}

// Interfaz para comer
public interface Eatable {
    void eat();
}

// Interfaz para dormir
public interface Sleepable {
    void sleep();
}
```

```java
// Clase para la oficina que puede manejar
// diferentes tipos de trabajadores
public class Office {
    private Workable workable;
    private Eatable eatable;
    private Sleepable sleepable;

    public Office(Workable workable) {
        this.workable = workable;
    }

    public void setEatable(Eatable eatable) {
        this.eatable = eatable;
    }

    public void setSleepable(Sleepable sleepable) {
        this.sleepable = sleepable;
    }

    public void manageWork() {
        workable.work();
    }

    public void manageLunch() {
        if (eatable != null) eatable.eat();
    }

    public void manageSleep() {
        if (sleepable != null) sleepable.sleep();
    }
}
```

Buen diseño

```java
// Clase principal para ejecutar el código
public class Main {
    public static void main(String[] args) {
        Developer developer = new Developer();
        Robot robot = new Robot();

        Office office = new Office(developer);
        office.setEatable(developer);
        office.setSleepable(developer);
        office.manageWork();  // Developing software
        office.manageLunch(); // Eating lunch
        office.manageSleep(); // Sleeping at home

        office = new Office(robot);
        office.manageWork();  // Assembling parts
        office.manageLunch(); // No output, robots don't eat
        office.manageSleep(); // No output, robots don't sleep
    }
}
```

Interface Segregation Principle (ISP)
Principio de Segregación de Interfaces

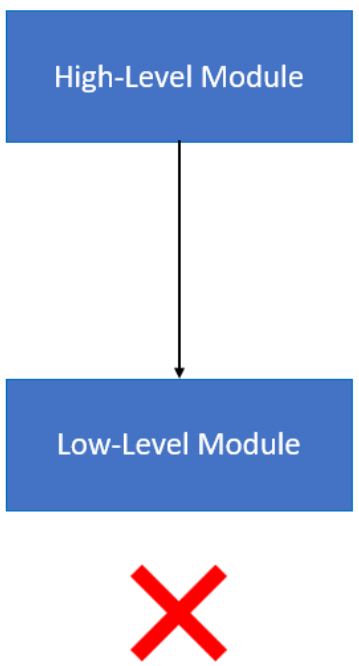
Beneficios del ISP

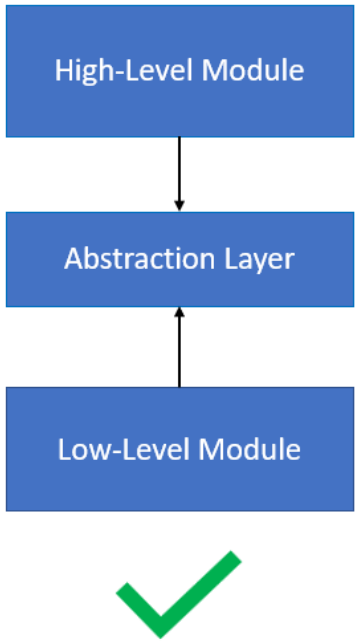
- **Claridad**: Las interfaces pequeñas y específicas son más fáciles de entender y mantener.
- **Flexibilidad**: Las clases solo implementan las interfaces que realmente necesitan, evitando la implementación de métodos no aplicables.
- **Reusabilidad**: Los componentes se pueden reutilizar de manera más efectiva, ya que solo dependen de las interfaces necesarias.

Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

Este principio establece que las clases de alto nivel no deberían depender de clases de bajo nivel. Ambas deberían depender de abstracciones. Además, las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Las dependencias deben dirigirse hacia interfaces o abstracciones, no hacia implementaciones concretas.





Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

Mal diseño (violación del DIP)

Supongamos que tenemos una clase `Database` que se conecta directamente a una base de datos específica (por ejemplo, MySQL). Además, tenemos una clase `UserService` que depende directamente de `Database`.

```
// Clase que se conecta a una base de datos MySQL
public class MySQLDatabase {
    public void connect() {
        System.out.println("Connecting to MySQL database...");
    }

    public void disconnect() {
        System.out.println("Disconnecting from MySQL database...");
    }

    public void getData() {
        System.out.println("Fetching data from MySQL database...");
    }
}
```

Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

```java
// Clase de servicio que depende directamente de MySQLDatabase
public class UserService {
    private MySQLDatabase database;

    public UserService() {
        this.database = new MySQLDatabase();
    }

    public void displayUserData() {
        database.connect();
        database.getData();
        database.disconnect();
    }
}
```

*En este diseño, `UserService` **depende directamente** de `MySQLDatabase`, lo que hace que **sea difícil cambiar** la implementación de la base de datos sin modificar `UserService`.*

```java
// Clase principal para ejecutar el código
public class Main {
    public static void main(String[] args) {
        UserService userService = new UserService();
        userService.displayUserData();
    }
}
```

Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

Buen diseño (siguiendo el DIP)

Para adherirse al DIP, debemos introducir una abstracción (una interfaz) y hacer que `UserService` dependa de esta abstracción en lugar de una implementación concreta.

```
// Interfaz para la base de datos
public interface Database {
    void connect();
    void disconnect();
    void getData();
}
```

Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

```java
// Interfaz para la base de datos
public interface Database {
    void connect();
    void disconnect();
    void getData();
}
```

```java
// Implementación de MySQLDatabase que
// cumple con la interfaz Database
public class MySQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to MySQL database...");
    }

    @Override
    public void disconnect() {
        System.out.println("Disconnecting from MySQL database...");
    }

    @Override
    public void getData() {
        System.out.println("Fetching data from MySQL database...");
    }
}
```

Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

```java
// Interfaz para la base de datos
public interface Database {
    void connect();
    void disconnect();
    void getData();
}
```

```java
// Implementación de OracleDatabase que cumple con la interfaz Database
public class OracleDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to Oracle database...");
    }

    @Override
    public void disconnect() {
        System.out.println("Disconnecting from Oracle database...");
    }

    @Override
    public void getData() {
        System.out.println("Fetching data from Oracle database...");
    }
}
```

Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

```java
// Interfaz para la base de datos
public interface Database {
    void connect();
    void disconnect();
    void getData();
}
```

```java
// Clase de servicio que depende
//de la interfaz Database
public class UserService {
    private Database database;

    // Constructor que acepta
    // cualquier implementación de Database
    public UserService(Database database) {
        this.database = database;
    }

    public void displayUserData() {
        database.connect();
        database.getData();
        database.disconnect();
    }
}
```

Dependency Inversion Principle (DIP)
Principio de Inversión de Dependencias

```java
// Clase de servicio que depende
//de la interfaz Database
public class UserService {
    private Database database;

    // Constructor que acepta
    // cualquier implementación de Database
    public UserService(Database database) {
        this.database = database;
    }

    public void displayUserData() {
        database.connect();
        database.getData();
        database.disconnect();
    }
}
```

```java
// Interfaz para la base de datos
public interface Database {
    void connect();
    void disconnect();
    void getData();
}
```

```java
// Clase principal para ejecutar el código
public class Main {
    public static void main(String[] args) {
        Database mySQLDatabase = new MySQLDatabase();
        UserService userService1 = new UserService(mySQLDatabase);
        userService1.displayUserData();

        Database oracleDatabase = new OracleDatabase();
        UserService userService2 = new UserService(oracleDatabase);
        userService2.displayUserData();
    }
}
```

S O L I D - Resumen

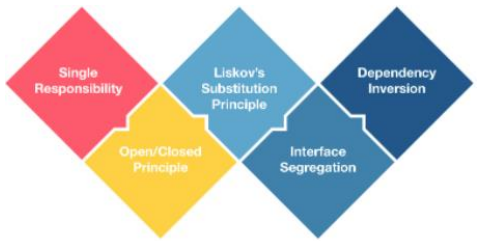
Principio de Responsabilidad Única (SRP): Cada clase debe tener una sola responsabilidad o motivo para cambiar.

Principio de Abierto/Cerrado (OCP): Las clases deben estar abiertas para extensión, pero cerradas para modificación. Es decir, deberías poder añadir nuevas funcionalidades sin cambiar el código existente.

Principio de Sustitución de Liskov (LSP): Las subclases deben ser reemplazables por sus clases base sin alterar el comportamiento del programa. Las subclases deben poder usarse en lugar de las clases base sin problemas.

Principio de Segregación de Interfaces (ISP): Los clientes no deben verse obligados a depender de interfaces que no usan. Es mejor tener interfaces específicas y pequeñas.

Principio de Inversión de Dependencias (DIP): Las clases de alto nivel no deben depender de clases de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles; los detalles deben depender de abstracciones.



Principios S.O.L.I.D.

Principio de responsabilidad única

Se debe crear una responsabilidad por clase o función y que está sea tolerante a cambios, y fácil de leer y de testear.

Principio abierto y cerrado

El código debe estar abierto a extenderse con nuevas funcionalidades, pero cerrado a modificaciones.

Principio de sustitución de Liskov

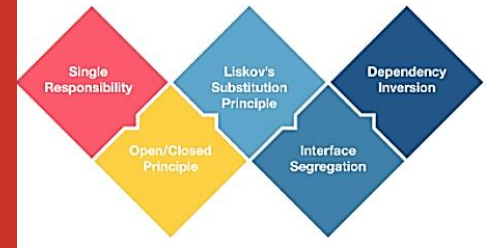
Las clases heredadas puede ser usada en cualquier parte sin que esto **afecte su comportamiento**, y sin tener que llamar a la clase padre cuando se espere un objeto.

Principio de segregación de interfaz

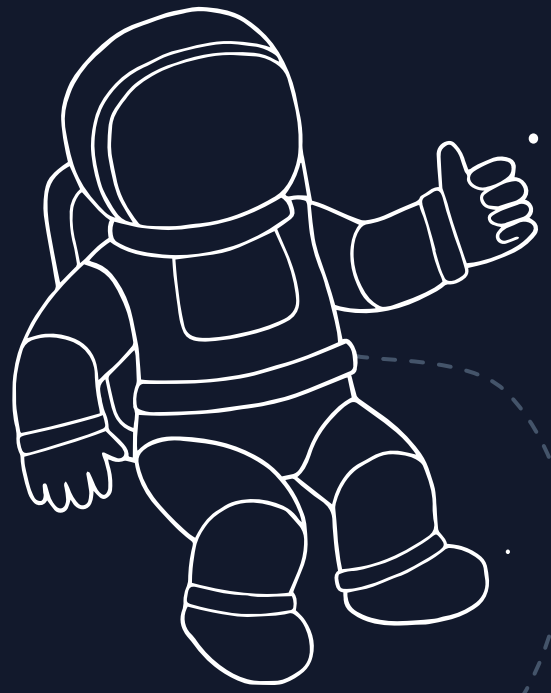
Las clases no deberían depender de métodos o propiedades que **no necesita**, sí es un lenguaje sin tipado se debe usar el **duck typing.**

Principio de inversión de dependencia

Los módulos de **alto nivel** no deben depender de un módulo de **bajo nivel**, por ejemplo: Un componente que manejan la lógica de negocio, no puede depender de un componente que maneja la UI.




OBJECT-ORIENTED
PROGRAMMING



Programa académico CAMPUS

Módulo JAVA