¿Qué son los principios SOLID?

- Los principios SOLID son un conjunto de principios de diseño de software que ayudan a los desarrolladores a crear sistemas más mantenibles, flexibles y escalables.

1. S - Principio de Responsabilidad Única (Single Responsibility Principle, SRP):

- Cada clase debe tener una única responsabilidad o motivo de cambio. En otras palabras, una clase debe centrarse en una única tarea o función.
- *Definición:* Cada clase debe tener una única responsabilidad, es decir, un solo motivo para cambiar.
- *Ejemplo:* Imagina que tienes una clase que maneja la lógica de un libro y también la lógica para imprimir el libro.

```java
public class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    // Método para imprimir los detalles del libro (mala práctica)
    public void printBook() {
        System.out.println("Title: " + title + ", Author: " + author);
    }
}
```

Corregido

- Separar la responsabilidad de impresión en una clase diferente.

```java
public class Book {
    private String title;
    private String author;
```

```
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }
}

public class BookPrinter {
    public void printBook(Book book) {
        System.out.println("Title: " + book.getTitle() + ", Author: " + book.getAuthor());
    }
}
```

2. O - Principio de Abierto/Cerrado (Open/Closed Principle, OCP):

- El Principio de Abierto/Cerrado establece que las clases deben estar abiertas para su extensión, pero cerradas para su modificación. Esto significa que debemos poder agregar nuevas funcionalidades extendiendo el código existente sin tener que modificar el código original.
- *Ejemplo:* Cálculo de Precios con Descuento
- *Implementación Inicial (Incorrecta):* Tenemos una clase PriceCalculator que calcula el precio de un producto. Al agregar un descuento, modificamos directamente la clase existente, lo que viola el principio OCP.

```
public class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public double getPrice() {
        return price;
    }
```

```
}

public class PriceCalculator {
    public double calculatePrice(Product product, boolean applyDiscount) {
        double price = product.getPrice();
        if (applyDiscount) {
            price = price * 0.9; // 10% de descuento
        }
        return price;
    }
}
```

En este diseño, cada vez que queramos agregar un nuevo tipo de descuento, tendríamos que modificar PriceCalculator.

- *Implementación Mejorada (Correcta):* Creamos una interfaz `PricedItem` y extendemos las clases según sea necesario, sin modificar PriceCalculator.

```
public interface PricedItem {
    double getPrice();
}

public class RegularProduct implements PricedItem {
    private String name;
    private double price;

    public RegularProduct(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public double getPrice() {
        return price;
    }
}

public class DiscountedProduct implements PricedItem {
    private PricedItem product;
    private double discountPercentage;

    public DiscountedProduct(PricedItem product, double discountPercentage) {
        this.product = product;
        this.discountPercentage = discountPercentage;
    }

    @Override
```

```
    public double getPrice() {
        return product.getPrice() * (1 - discountPercentage / 100);
    }
}

public class PriceCalculator {
    public double calculatePrice(PricedItem pricedItem) {
        return pricedItem.getPrice();
    }
}
```

- Uso en la Aplicación

```
public class Main {
    public static void main(String[] args) {
        PricedItem regularProduct = new RegularProduct("Laptop", 1000.0);
        PricedItem discountedProduct = new DiscountedProduct(regularProduct, 10); // 10% de

        PriceCalculator calculator = new PriceCalculator();

        System.out.println("Precio regular: " + calculator.calculatePrice(regularProduct));
        System.out.println("Precio con descuento: " + calculator.calculatePrice(discountedPr
    }
}
```

3. L - Principio de Sustitución de Liskov (Liskov Substitution Principle, LSP)

- El Principio de Sustitución de Liskov establece que las subclases deben ser sustituibles por sus clases base sin alterar el comportamiento del programa.
-
- *Ejemplo:* Vehículos y Capacidad de Vuelo

Implementación Inicial (Incorrecta):

Imaginemos que tenemos una clase `Vehicle` una subclase `Car` y `Plane`. Si un Plane no puede sustituir a Vehicle sin problemas, estamos violando el LSP.

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Engine started");
    }

    public void fly() {
        // Los vehículos en general no vuelan
        throw new UnsupportedOperationException("This vehicle can't fly");
    }
```

```
}

public class Car extends Vehicle {
    // No necesita cambiar nada
}

public class Plane extends Vehicle {
    @Override
    public void fly() {
        System.out.println("Plane is flying");
    }
}
```

Al intentar sustituir `Vehicle` por `Plane` en cualquier contexto que use `Vehicle`, encontraremos problemas si el método `fly` es llamado por un `Car`.

Implementación Mejorada (Correcta):

Debemos replantear el diseño para que Vehicle no tenga un método fly que no todos sus hijos puedan implementar correctamente. En su lugar, podemos usar interfaces para manejar la capacidad de vuelo.

```
public abstract class Vehicle {
    public void startEngine() {
        System.out.println("Engine started");
    }
}

public interface Flyable {
    void fly();
}

public class Car extends Vehicle {
    // No necesita implementar fly
}

public class Plane extends Vehicle implements Flyable {
    @Override
    public void fly() {
        System.out.println("Plane is flying");
    }
}
```

Uso en la Aplicación

```
public class Main {
    public static void main(String[] args) {
```

```
        Vehicle car = new Car();
        Vehicle plane = new Plane();

        car.startEngine();
        plane.startEngine();

        // Si el vehículo es volador, volará
        if (plane instanceof Flyable) {
            ((Flyable) plane).fly();
        }
    }
}
```

4. I - Principio de Segregación de Interfaces (Interface Segregation Principle, ISP):

- El Principio de Segregación de Interfaces establece que los clientes no deben estar forzados a depender de interfaces que no utilizan. Esto significa que es mejor tener varias interfaces específicas que una sola interfaz general y grande.
- *Ejemplo:* Dispositivos Multifuncionales

Implementación Inicial (Incorrecta)

Imaginemos que tenemos una interfaz `MultiFunctionDevice` que incluye métodos para imprimir, escanear y enviar fax. No todos los dispositivos implementarán todas estas funciones, lo que puede llevar a métodos no implementados en algunas clases.

```
public interface MultiFunctionDevice {
    void print();
    void scan();
    void fax();
}

public class Printer implements MultiFunctionDevice {
    @Override
    public void print() {
        System.out.println("Printing...");
    }

    @Override
    public void scan() {
        throw new UnsupportedOperationException("Scan not supported");
    }
```

```
    @Override
    public void fax() {
        throw new UnsupportedOperationException("Fax not supported");
    }
}

public class Scanner implements MultiFunctionDevice {
    @Override
    public void print() {
        throw new UnsupportedOperationException("Print not supported");
    }

    @Override
    public void scan() {
        System.out.println("Scanning...");
    }

    @Override
    public void fax() {
        throw new UnsupportedOperationException("Fax not supported");
    }
}
```

Implementación Mejorada (Correcta)

Dividimos la interfaz grande en varias interfaces más pequeñas y específicas.

```
public interface Printer {
    void print();
}

public interface Scanner {
    void scan();
}

public interface Fax {
    void fax();
}

public class SimplePrinter implements Printer {
    @Override
    public void print() {
        System.out.println("Printing...");
    }
}
```

```java
public class SimpleScanner implements Scanner {
    @Override
    public void scan() {
        System.out.println("Scanning...");
    }
}

public class MultiFunctionPrinter implements Printer, Scanner, Fax {
    @Override
    public void print() {
        System.out.println("Printing...");
    }

    @Override
    public void scan() {
        System.out.println("Scanning...");
    }

    @Override
    public void fax() {
        System.out.println("Faxing...");
    }
}
```

Uso en la aplicación:

```java
public class Main {
    public static void main(String[] args) {
        Printer printer = new SimplePrinter();
        printer.print();

        Scanner scanner = new SimpleScanner();
        scanner.scan();

        MultiFunctionPrinter mfp = new MultiFunctionPrinter();
        mfp.print();
        mfp.scan();
        mfp.fax();
    }
}
```

D - Principio de Inversión de Dependencias (Dependency Inversion Principle, DIP):

- El Principio de Inversión de Dependencias establece que los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben

depender de abstracciones. Además, las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones.

- *Ejemplo:* Servicio de Notificaciones

Implementación Inicial (Incorrecta)

Tenemos una clase `NotificationService` que depende directamente de una implementación concreta `EmailService`.

```
public class EmailService {
    public void sendEmail(String message) {
        System.out.println("Sending email: " + message);
    }
}

public class NotificationService {
    private EmailService emailService;

    public NotificationService() {
        this.emailService = new EmailService();
    }

    public void notify(String message) {
        emailService.sendEmail(message);
    }
}
```

En este diseño, `NotificationService` depende directamente de `EmailService`, lo que hace que sea difícil cambiar la implementación del servicio de notificaciones (por ejemplo, para usar SMS en lugar de email).

Implementación Mejorada (Correcta)

Introducimos una abstracción `MessageService` que `NotificationService` utilizará. Luego, implementamos `EmailService` como una de las posibles implementaciones de `MessageService`.

Uso en la aplicación

```
public interface MessageService {
    void sendMessage(String message);
}

public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sending email: " + message);
```

```
    }
}

public class SmsService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

public class NotificationService {
    private MessageService messageService;

    // Inyección de dependencia a través del constructor
    public NotificationService(MessageService messageService) {
        this.messageService = messageService;
    }

    public void notify(String message) {
        messageService.sendMessage(message);
    }
}
```

Ejercicios

1. Ordenes

```
public class Order {
    private List<String> items;
    private double totalAmount;

    public Order(List<String> items) {
        this.items = items;
        this.totalAmount = calculateTotal();
    }

    private double calculateTotal() {
        // Lógica para calcular el total de la orden
        return 100.0; // simplificado
    }

    public void printOrder() {
        // Lógica para imprimir la orden
    }
```

```
    public void saveToDatabase() {
        // Lógica para guardar la orden en la base de datos
    }
}
```

Refactoriza la clase Order para que cumpla con el Principio de Responsabilidad Única. Debes separar las responsabilidades de cálculo, impresión y almacenamiento en clases diferentes.

Pistas:

1. La clase Order solo debe gestionar los datos de la orden.
2. Crea una nueva clase para manejar la impresión de la orden.
3. Crea una nueva clase para manejar el almacenamiento de la orden en la base de datos.

2. Métodos de pago

Refactoriza el código para que cumpla con el Principio de Abierto/Cerrado. Debes permitir la extensión de nuevos métodos de pago sin modificar la clase `PaymentProcessor`.

```
public class PaymentProcessor {
    public void processPayment(String paymentType) {
        if (paymentType.equals("credit")) {
            // Lógica para procesar el pago con tarjeta de crédito
        } else if (paymentType.equals("paypal")) {
            // Lógica para procesar el pago con PayPal
        }
    }
}
```

Pistas 1. Crea una interfaz PaymentMethod con un método `processPayment`. 2. Implementa esta interfaz en clases concretas para cada tipo de pago (por ejemplo, `CreditCardPayment` y `PayPalPayment`). 3. Modifica PaymentProcessor para que use la interfaz `PaymentMethod`.

3. Worker

Refactoriza el código para que cumpla con el Principio de Segregación de Interfaces. Crea interfaces más específicas para evitar que las clases implementen métodos que no necesitan.

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

public class HumanWorker implements Worker {
```

```
    @Override
    public void work() {
        // Lógica para trabajar
    }

    @Override
    public void eat() {
        // Lógica para comer
    }
}

public class RobotWorker implements Worker {
    @Override
    public void work() {
        // Lógica para trabajar
    }

    @Override
    public void eat() {
        throw new UnsupportedOperationException("Robots don't eat");
    }
}
```

Pistas

1. Crea una interfaz Workable para la funcionalidad de trabajo.
2. Crea una interfaz Eatable para la funcionalidad de comer.
3. Implementa estas interfaces en las clases correspondientes.

4. Database

Refactoriza el código para que cumpla con el Principio de Inversión de Dependencias. Introduce una abstracción para la funcionalidad de almacenamiento de datos.

```
public class Database {
    public void save(String data) {
        // Lógica para guardar datos en la base de datos
    }
}

public class DataService {
    private Database database = new Database();

    public void saveData(String data) {
        database.save(data);
    }
```

```
}
```

Pistas

1. Crea una interfaz `DataStorage` con un método save.
2. Implementa `DataStorage` en la clase `Database`.
3. Modifica `DataService` para que dependa de `DataStorage` en lugar de `Database`.

5. Employees

Refactoriza la clase Employee para que cumpla con el Principio de Responsabilidad Única. Debes separar las responsabilidades de cálculo, generación de reportes y almacenamiento en clases diferentes.

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

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

    public void calculatePay() {
        // Lógica para calcular el salario
    }

    public void generateReport() {
        // Lógica para generar un reporte del empleado
    }

    public void saveToDatabase() {
        // Lógica para guardar el empleado en la base de datos
    }
}
```

Pistas

1. La clase `Employee` solo debe manejar los datos del empleado.
2. Crea una nueva clase `PayrollService` para manejar el cálculo del salario.
3. Crea una nueva clase `EmployeeReport` para manejar la generación del reporte.
4. Crea una nueva clase `EmployeeRepository` para manejar el almacenamiento en la base de datos.

6. TaxCalculator

Refactoriza el código para que cumpla con el Principio de Abierto/Cerrado. Debes permitir la extensión de nuevos cálculos de impuestos sin modificar la clase TaxCalculator.

```
public class TaxCalculator {
    public double calculateTax(String country) {
        if (country.equals("USA")) {
            // Lógica para calcular el impuesto en USA
            return 0.1;
        } else if (country.equals("UK")) {
            // Lógica para calcular el impuesto en UK
            return 0.2;
        }
        return 0;
    }
}
```

Pistas

1. Crea una interfaz `TaxStrategy` con un método `calculateTax`.
2. Implementa esta interfaz en clases concretas para cada país (por ejemplo, USATaxStrategy y UKTaxStrategy).
3. Modifica `TaxCalculator` para que use la interfaz `TaxStrategy`.