Principios SOLID

▼ Principio de responsabilidad unica (SRP)

Enunciado

Cada clase debe tener responsabilidad únicamente sobre una parte de la funcionalidad del programa, y dicha responsabilidad debe estar completamente encapsulada por la clase. Todos los métodos y atributos de la clase deben estar alineados con dicha funcionalidad.

“Una clase debe tener una única responsabilidad”.

Beneficios

- Promueve la cohesion y en cierta medida el bajo acoplamiento.
- Se evita la complejidad de clases grandes y dependientes.
- Facilita la extension y es mas robusto a las modificaciones.

Ejemplos

▼ No cumple con SRP

```
public class Order {
    public void calculateTotalPrice(List<Item> items) {
        // Calcula el precio total de los items en el pedido
        // ...
    }

    public void saveToDatabase() {
        // Guarda el pedido en la base de datos
        // ...
    }

    public void sendConfirmationEmail() {
        // Envía un correo electrónico de confirmación al cliente
        // ...
    }
}
```

En el ejemplo anterior la clase `Order` tiene multiples responsabilidades de distintas funcionalidades del programa:

- Calcular el precio
- Guardar en la Base de Datos
- Enviar mail de confirmacion al cliente

Posee mas de una responsabilidad, por lo que no cumple con **SRP**

▼ Cumple con SRP

```
public class Order {
    public void calculateTotalPrice(List<Item> items) {
        // Calcula el precio total de los items en el pedido
        // ...
    }
}
```

```
public class OrderRepository {
    public void saveToDatabase(Order order) {
        // Guarda el pedido en la base de datos
        // ...
    }
}

public class EmailService {
    public void sendConfirmationEmail(Order order) {
        // Envía un correo electrónico de confirmación al cliente
        // ...
    }
}
```

En el ejemplo anterior se tienen tres clases distintas, de las cuales cada una ejerce una responsabilidad sobre una funcionalidad del programa:

- `Order` calcula el precio total de la orden.
- `OrderRepository` almacena la información den la Base de Datos.
- `EmailService` envia un correo electronico de confirmacion al usuario.

▼ Principio de abierto cerrado (OCP)

Enunciado

Las clases deben estar **abiertas a la extension**, pero **cerradas a la modificación**.

- "*Abiertas a la extension*": Las responsabilidades de la clase pueden ser extendidas. Se pueden agregar nuevas responsabilidades mediante herencia, o combinación de herencia composición o agregación.
- "*Cerradas a la modificación*": No es posible ni necesario (si se cumple el principio) realizar cambios en el código de la clase.

Beneficios

- Permite mas flexibilidad y adaptabilidad a los cambios futuros sin introducir errores en el código existente.

Ejemplos

▼ No cumple con OCP

```
public class ShoppingCart {
    public double calculateTotalPrice(List<Item> items) {
        double totalPrice = 0;
        for (Item item : items) {
            totalPrice += item.getPrice();
        }
        return totalPrice;
    }
}
```

En el ejemplo anterior, `ShoppingCart` calcula el precio total de los items adjudicados en el carrito. Pero en el caso de que se quiera aplicar alguna otra funcionalidad, como un descuento, se tendra que cambiar el codigo de `ShoppingCart`, por lo que no se cumple con OCP.

▼ Cumple con OCP

```
public interface PriceCalculator {
    double calculateTotalPrice(List<Item> items);
}

public class ShoppingCart {
    private PriceCalculator priceCalculator;

    public ShoppingCart(PriceCalculator priceCalculator) {
        this.priceCalculator = priceCalculator;
    }

    public double calculateTotalPrice(List<Item> items) {
        return priceCalculator.calculateTotalPrice(items);
    }
}

public class RegularPriceCalculator implements PriceCalculator {
    public double calculateTotalPrice(List<Item> items) {
        double totalPrice = 0;
        for (Item item : items) {
            totalPrice += item.getPrice();
        }
        return totalPrice;
    }
}

public class DiscountedPriceCalculator implements PriceCalculator {
    public double calculateTotalPrice(List<Item> items) {
        double totalPrice = 0;
        for (Item item : items) {
            totalPrice += item.getPrice() * 0.9; // Aplica un descuento del 10%
        }
        return totalPrice;
    }
}
```

En este ejemplo, para poder calcular el monto total de dinero de los items de un carrito, se crea una interfaz y una clase encargada de realizar la lógica del calculo, y a la clase del carrito se le pasa una instancia del tipo de la interfaz. Con esto, si se quiere agregar otra funcionalidad, como un descuento en el monto, lo que se puede hacer crear otra clase que implemente la misma interfaz que la primera, pero que su logica de calculo sea distinta. Extendiendo de esta forma la funcionalidad de la clase `ShoppingCart`.

▼ Sustitución de Liskov (LSP)

Enunciado

El código que envía mensajes a un objeto del tipo T, debe funcionar igual cuando ese objeto es de un subtipo S del tipo T. Los objetos de una clase base deben poder ser reemplazados por objetos de una clase derivada sin alterar la corrección del programa.

“Si por cada objeto o1 de tipo S hay un objeto o2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o1 se sustituye por o2 entonces S es un subtipo de T”

Beneficios

- Facilidad de mantenimiento
- Mayor robustez y confiabilidad
- Reusabilidad y confiabilidad

Ejemplos

▼ Principio de segregación de interfaces (ISP)

Enunciado

Los clientes de una interfaz no deben ser forzados a depender de métodos que no utilizan. Se deben crear interfaces pequeñas y especificas para los diferentes casos de uso, en lugar de crear interfaces monolíticas.

“Los clientes no deben ser forzados a depender de tipos que no usan.”

Beneficios

- Evita el tener que crear métodos innecesarios.
- Reduce el acoplamiento entre componentes.
- Aumenta la cohesion.

Ejemplos

▼ No cumple con ISP

```
public interface Vehicle {
    void start();
    void accelerate();
    void brake();
    void fly();
    void land();
}


public class Airplane implements Vehicle {
    public void start() {
        // Iniciar los motores del avión
    }

    public void accelerate() {
        // Aumentar la velocidad del avión
    }

    public void brake() {
        // Disminuir la velocidad del avión
    }

    public void fly() {
        // Hacer volar el avión
    }

    public void land() {
        // Aterrizar el avión
    }
}

public class Car implements Vehicle {
    public void start() {
        // Iniciar el motor del automóvil
    }

    public void accelerate() {
        // Acelerar el automóvil
    }
```

```
    public void brake() {
        // Frenar el automóvil
    }

    public void fly() {
        // Este método no es relevante para un automóvil, pero se debe implementar debido a la interfaz
    }

    public void land() {
        // Este método no es relevante para un automóvil, pero se debe implementar debido a la interfaz
    }
}
```

En el ejemplo anterior, la clase `Car` que implementa la interfaz `Vehicle`, tiene que definir los metodos `fly()` y `land()` de dicha interfaz, aun cuando la clase no los usa.

▼ **Cumple con ISP**

```
public interface Drivable {
    void start();
    void accelerate();
    void brake();
}

public interface Flyable {
    void fly();
    void land();
}

public class Car implements Drivable {
    public void start() {
        // Iniciar el motor del automóvil
    }

    public void accelerate() {
        // Acelerar el automóvil
    }

    public void brake() {
        // Frenar el automóvil
    }
}

public class Airplane implements Drivable, Flyable {
    public void start() {
        // Iniciar los motores del avión
    }

    public void accelerate() {
        // Aumentar la velocidad del avión
    }

    public void brake() {
        // Disminuir la velocidad del avión
    }

    public void fly() {
        // Hacer volar el avión
    }

    public void land() {
        // Aterrizar el avión
    }
}
```

En este ejemplo, al contrario que en el anterior, se segmenta la clase `Vhicule` y se crean dos interfaces distintas, implementando cada una aspectos mas especificos. Ahora, al estar separados los metodos, la clase `Car` puede implementar solamente la interfaz `Drivable`, ya que es la que tiene todos los metodos necesarios, y ni uno mas.

▼ Principio de inversion de dependencia (DIP)

Enunciado

"*Las clases de alto nivel no deben depender de clases de bajo nivel, ambas deben depender de abstracciones.*

Las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones."

Se debe depender de abstracciones.

Beneficios

- Hay mas flexibilidad frente a los cambios
- Mayor robustez del código
- La reutilizacion es mas simple

Ejemplos

▼ No cumple con DIP

▼ Cumple com DIP