5.1. Principios y patrones de diseño. Arquitecturas

Acceso a Datos

Alejandro Roig Aguilar
alejandro.roig@iesalvarofalomir.org

IES Álvaro Falomir
Curso 2023-2024

Principios SOLID

Los principios SOLID, acuñados por Robert C. Martin, representan cincos principios básicos de diseño de software que promueven la mejora de la calidad del software a través de la modularidad, la reutilización y la mantenibilidad del código fuente.

Estos principios son:

- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface segregation
- Dependency Inversion

Principio de Responsabilidad Única

El Single Responsibility Principle, o Responsabilidad Única, nos dice que una clase debe tener una única responsabilidad, es decir, que debe tener una única razón para cambiar.

Por ejemplo, si tenemos una clase que se encarga de gestionar los usuarios de nuestra aplicación, esta clase debería tener las siguientes responsabilidades: CRUD de usuarios

Principio de Responsabilidad Única

```
public class Informe {
    public void generarInforme() {
        // lógica para generar el informe
    }
}

public class ImprimirInforme {
    public void imprimir(Informe informe) {
        // lógica para imprimir el informe
    }
}
```

En este ejemplo, la clase Informe solo tiene la responsabilidad de generar el informe, mientras que la clase ImprimirInforme tiene la responsabilidad de imprimir el informe.

Principio Abierto/Cerrado

El Open/Closed Principle, o abierto/cerrado, nos dice que las entidades de nuestro código (clases, módulos, funciones, etc) deben estar abiertas a la extensión pero cerradas a la modificación.

Es decir, si tenemos una clase que implementa una funcionalidad, no deberíamos modificar esa clase para añadir nuevas funcionalidades, sino que deberíamos crear una nueva clase que herede de la clase base y que implemente la nueva funcionalidad.

Principio Abierto/Cerrado

```
public abstract class Forma {
    abstract void dibujar();
}
public class Circulo extends Forma {
    void dibujar() { // lógica para dibujar un círculo }
}
public class Cuadrado extends Forma {
    void dibujar() { // lógica para dibujar un cuadrado }
}
```

En este ejemplo, la clase Forma está abierta para la extensión (se puede crear una nueva forma como Circulo o Cuadrado), pero cerrada para la modificación (no necesitamos cambiar la clase Forma para añadir una nueva forma).

Principio de Sustitución de Liskov

El Liskov Substitution Principle, o sustitución de Liskov, nos dice que las clases derivadas deben ser sustituibles por sus clases base.

Es decir, si tenemos una clase base que implementa una funcionalidad, las clases derivadas deben poder usar esa funcionalidad sin tener que modificar el código de la clase base.

Principio de Sustitución de Liskov

```
public class Pajaro{
    public void volar () { // lógica para volar }
}
public class Pinguino extends Pajaro {
    @Override
    public void volar() {
        throw new UnsupportedOperationException();
}
```

Este es un ejemplo de violación del principio de sustitución de Liskov, ya que Pinguino es una subclase de Pajaro, pero no puede volar. Una solución sería tener una clase separada para pájaros que pueden volar.

Principio de Segregación de Interfaces

El Interface Segregation Principle, o segregación de interfaces, nos dice que las interfaces deben ser lo más pequeñas posibles.

Es decir, si tenemos una interfaz que define una funcionalidad, no deberíamos añadir más funcionalidades a esa interfaz, sino que deberíamos crear una nueva interfaz que herede de la interfaz base y que defina la nueva funcionalidad.

Principio de Segregación de Interfaces

```
public interface Pajaro{
    void comer();
}
public interface PajaroVolador{
    void volar();
}
public class Pinguino implements Pajaro {
    public void comer () { // lógica para comer }
}
public class Paloma implements Pajaro, PajaroVolador {
    public void comer () { // lógica para comer }
    public void volar () { // lógica para volar }
}
```

En este ejemplo, Pinguino no está forzado a implementar un método volar() que no necesita, al segregar en dos interfaces diferentes.

Principio de Inversión de Dependencias

El Dependecy Inversion Principle, o inversión de dependencias, nos dice que las clases de alto nivel no deben depender de las clases de bajo nivel, sino que ambas deben depender de abstracciones.

Es decir, si tenemos una clase que implementa una funcionalidad, no deberíamos depender de la implementación de esa funcionalidad, sino que deberíamos depender de una interfaz que defina la funcionalidad.

Principio de Inversión de Dependencias

```
public interface BaseDeDatos{
    void guardar(String datos);
}
public class MySQLDB implements BaseDeDatos {
    public void guardar (String datos) { // lógica para guardar datos en MySQL}
}
public class Aplicacion {
    private BaseDeDatos db;
    public Aplicacion (BaseDeDatos db) { this.db = db; }
    public void guardarDatos(String datos) { db.guardar(datos); }
}
```

En este ejemplo, la clase Aplicacion no depende directamente de la clase MySqlDB, sino que ambos dependen de la abstracción BaseDeDatos. Así, si queremos cambiar la base de datos en el futuro, solo necesitamos crear una nueva implementación de BaseDeDatos, sin cambiar la clase Aplicacion.

Patrones de Diseño

Un patrón de diseño de software es una solución generalmente aplicable a un problema común en el diseño de software, con el objetivo de mejorar la eficiencia del proceso y la calidad del código.

Los patrones GoF (Gang of Four) hacen referencia a un conjunto de 23 patrones clasificados en tres categorías:

- Creacionales: Enfocados en la creación de objetos. Ej: Singleton, Factory Method, Abstract Factory y Builder.
- Estructurales: Abordan la composición de las clases y objetos. Algunos ejemplos son: Adapter, Decorator, Composite y Proxy.
- De comportamiento: Se ocupan de la comunicación e interacción entre objetos. Ej: Observer, Strategy y Command.

Patrones de Diseño

Podemos añadir también otras dos categorías:

- Arquitectónicos: Estos patrones abordan la estructura y organización de sistemas de software a gran escala. Algunos ejemplos son: Modelo-Vista-Controlador (MVC), Capas y Microservicios.
- De Concurrencia: Estos patrones se utilizan para gestionar la concurrencia y la comunicación entre hilos. Algunos ejemplos son: Mutex, Semaphore, Productor-Consumidor y Monitor.

Arquitecturas Software

Llamamos arquitectura de software a la estructura de un sistema de software, es decir, a la forma en la que se organizan los componentes del sistema.

Las arquitecturas de software se pueden clasificar en diferentes tipos, como arquitecturas monolíticas, arquitecturas de capas, arquitecturas basadas en componentes, arquitecturas basadas en microservicios, etc.

Arquitectura Monolítica

Es un enfoque tradicional en el que todos los componentes de una aplicación se agrupan en un solo bloque. La lógica de negocio, la interfaz de usuario y la capa de acceso a datos se encuentran dentro de la misma aplicación.

Es fácil de desarrollar y desplegar, pero puede volverse complejo y difícil de mantener a medida que la aplicación crece. Características:

- Todos los componentes se ejecutan en el mismo proceso y comparten recursos.
- La escalabilidad puede ser un desafío, ya que la aplicación se ejecuta en una sola instancia.
- Los cambios en una parte de la aplicación pueden afectar a otras partes.

Arquitectura de capas

La arquitectura de capas es una arquitectura de software en la que los componentes del sistema se organizan en capas, donde cada capa tiene una responsabilidad específica.

Las capas típicas incluyen la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos. Además, las capas se comunican entre sí mediante interfaces. Características:

- Mejora la modularidad y la reutilización del código.
- Permite cambios en una capa sin afectar a las demás.
- Facilita la escalabilidad y el mantenimiento del sistema.

Arquitectura basada en microservicios

La arquitectura basada en microservicios es una arquitectura de software en la que los componentes del sistema se organizan en microservicios. Cada microservicio se encarga de una parte del sistema, y los microservicios se comunican entre sí mediante interfaces.

- Cada microservicio se puede desarrollar, desplegar y escalar de forma independiente.
- Mejora la flexibilidad y la agilidad del desarrollo.
- Permite la adopción de diferentes tecnologías y enfoques dentro de cada microservicio.