Material descargable

Programación **orientada a objetos**

egg

Principios SOLID - Patrones de diseño

Los Principios SOLID y los Patrones de Diseño son dos conceptos fundamentales en la programación orientada a objetos que proporcionan técnicas y patrones probados para desarrollar software de alta calidad que sea fácil de mantener, comprender y ampliar.

Principios SOLID

Los Principios SOLID son un conjunto de cinco principios de diseño de software orientado a objetos que fueron introducidos por Robert C. Martin, a menudo conocido como "Uncle Bob". Estos principios son:

S - Principio de Responsabilidad Única (Single Responsibility Principle)

O - Principio de Abierto/Cerrado (Open-Closed Principle)

L - Principio de Sustitución de Liskov (Liskov Substitution Principle)

I - Principio de Segregación de Interfaz (Interface Segregation Principle)

D - Principio de Inversión de Dependencia (Dependency Inversion Principle)

Estos principios son importantes para la programación porque, cuando se aplican correctamente, mejoran la estructura y la organización del código, lo que facilita su mantenimiento y extensión.

💡 **Hoy veremos cómo aplicar el primer principio, los otros principios serán abordados a medida que veamos temas más avanzados.**

Patrones de Diseño

Los patrones de diseño son soluciones reutilizables a problemas comunes que ocurren en el diseño de software. Son plantillas que se pueden aplicar a diversos problemas en distintos contextos. Los patrones de diseño no son fragmentos de código que se pueden copiar y pegar en un programa, sino más bien conceptos y técnicas que ayudan a los desarrolladores a crear un diseño más eficiente, escalable y mantenible.

Hay muchísimos patrones de diseño orientados a distintas áreas de la programación. Los más famosos pertenecen al grupo "Gang of Four" (*grupo de los cuatro, por sus fundadores*) y suman 23 patrones en total. Además de estos, hay más de otros 20 que se pueden aplicar en el backend.

A medida que avancemos explicaremos algunos de ellos para aplicarlos en los nuevos temas. Por ejemplo, ahora que sabemos crear nuestros modelos y nuestro código se volvió más complejo, nos beneficiaremos de aplicar un patrón de diseño.

S - Principio de Responsabilidad Única (SRP)

El Principio de Responsabilidad Única establece que una clase debería tener solo una razón para cambiar. Esto significa que una clase debe tener solo una tarea o responsabilidad. Al adherirse a este principio, se minimiza la cantidad de cambios necesarios cuando se modifica una funcionalidad. Es más fácil entender, probar y mantener una clase que tiene una sola responsabilidad.

El SRP es importante porque ayuda a mantener las clases enfocadas y pequeñas. Cuando una clase tiene múltiples responsabilidades, se vuelve más compleja, difícil de leer, de mantener y puede llevar a efectos secundarios no deseados cuando se realizan cambios.

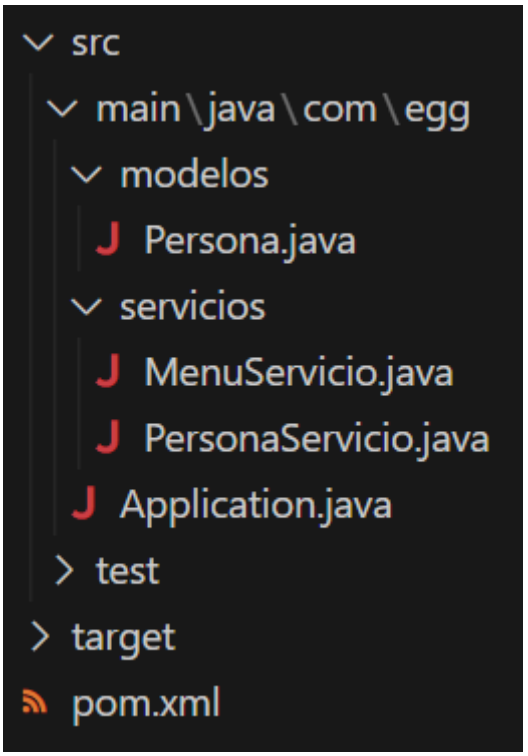
Si aplicamos este principio a los modelos, entonces deberíamos cambiar aquellos métodos que apliquen una lógica que no le "pertenezca". Para eso usaremos el patrón de diseño experto.

Patrón experto

El Patrón Experto es uno de los principios de GRASP (General Responsibility Assignment Software Patterns) que guían el diseño orientado a objetos. Este patrón sugiere que la responsabilidad de una determinada tarea debe asignarse al módulo o a la clase que tiene la mayor cantidad de información necesaria para cumplir esa tarea.

Por ejemplo, el método **imprimirPropiedades()** de una clase Persona debería devolver un String y no realizar un System.out.println(), la impresión de los datos debería ocurrir en otra clase, y el método debería cambiar su nombre a obtenerPropiedades().

Esta otra clase se le suele llamar **"ClaseServicio"**, en nuestro caso sería **"PersonaServicio"** y debería estar en un paquete "servicios" a la altura del directorio de "modelos".



Esta clase ***PersonaServicio*** deberá implementar toda la lógica de la aplicación que involucre la manipulación de los objetos Persona: la creación y validación de objetos, la solicitud de datos al usuario, la impresión de sus datos, etc.

También en la imagen puedes observar como existe un ***MenuServicio***, donde se implementa toda la lógica correspondiente a la impresión del menú en pantalla y el flujo de navegación en el mismo.

La clase **PersonaServicio** podría tener un atributo global para almacenar a todas las personas de la aplicación. A diferencia de las clases modelos, las clases servicios no deben implementar getters y setters.

```
public class PersonaServicio {

    private Scanner pepe = new Scanner(System.in);
    private Persona[] personas = new Persona[2];

    public Persona crearPersona() {
        System.out.println("ingrese un nombre");
        String nombre = pepe.nextLine();
        System.out.println("ingrese una edad");
        Integer edad = pepe.nextInt();
        Persona persona = new Persona(nombre, edad);
        almacenarEnPersonas(persona);
        return persona;
    }

    public void almacenarEnPersonas(Persona persona) {
        for (int i = 0; i < personas.length; i++) {
            if (personas[i] == null) {
                personas[i] = persona;
                break;
            }
        }
    }
}
```

La clase **MenuServicio** debe tener un atributo **PersonaServicio** que le permita invocar los métodos de la clase.

```java
public class MenuServicio {

    private PersonaServicio personaServicio;
    private Scanner pepe = new Scanner(System.in);

    public MenuServicio() {
        personaServicio = new PersonaServicio();
    }

    public void generarMenu() {
        System.out.println("Menu:");
        System.out.println("1 - crear persona");
        System.out.println("selecciona una opcion");
        Integer opcion = pepe.nextInt();
        switch (opcion) {
            case 1 -> opcion1();
        }
    }

    public void opcion1() {
        personaServicio.crearPersona();
    }
}
public class Application {

    public static void main(String[] args) {
        MenuServicio menuServicio = new MenuServicio();
        menuServicio.generarMenu();
    }

}
```

Y la clase Application solo debe usar la clase *MenuServicio* para iniciar el programa.

```java
public class Application {

    public static void main(String[] args) {
        MenuServicio menuServicio = new MenuServicio();
        menuServicio.generarMenu();
    }
}
```

☝️ Aplicar este patrón nos permitirá tener el código mucho más ordenado y legible.