DAM 1 DAW 1 / Entornos de desarrollo

Optimización y documentación

4.2 Principios SOLID

Autor: Lisa ERIKSEN
Fecha: 2025 / 2026

1. Introducción

Al finalizar la actividad, el estudiante será capaz de:

1. Comprender los principios SOLID
2. Identificar violaciones de cada principio
3. Proponer mejoras en el diseño del software

2. Principios SOLID — Descubriendo buenas prácticas

2.1 Single Responsibility Principle (SRP)

Actividad: Estás trabajando en una aplicación de gestión de usuarios (registro, autenticación y almacenamiento). Descubrimos y usamos los principios SOLID para mejorar el sistema de gestión.

❌ **Observa la mala practica**

```
class UsuarioManager {

    public void validarUsuario() {
        // lógica para validar el usuario
    }

    public void guardarEnBaseDeDatos() {
        // lógica para guardar el usuario en la base de datos
    }

    public void enviarEmail() {
        // lógica para enviar un correo electrónico
    }
}
```

Preguntas:

1. ¿Cuántas responsabilidades tiene la clase **UsuarioManager**?
2. ¿Qué problemas puede causar esta situación?
3. ¿Qué ocurre si cambia la lógica de envío de correos?

✅ **Observa la buena práctica:**

```
class UserValidator {
    public void validarUsuario() {
        // lógica para validar el usuario
    }
}

class UserRepository {
    public void guardarEnBaseDatos() {
        // lógica para guardar en la base de datos
    }
}

class EmailService {
    public void enviarEmail() {
        // lógica para enviar un correo electrónico
    }
}
```

4. Explica el cambio en el código.
5. ¿Qué gana el proyecto?

   ☐ Orden
   ☐ Facilidad de mantenimiento
   ☐ Posibilidad de ampliar
   ☐ Confusión

6. Descubres:

El primer de los principios SOLID dice: Cada clase debe tener **una sola responsabilidad**.

Este principio se llama:

Single Responsibility Principle (SRP)

2.2 Open / Closed Principle (OCP)

Situación: Antes de guardar un usuario en la base de datos, el sistema debe verificar que cumple ciertas reglas de validación. Por el momento, solo existe **una regla**: el usuario debe ser mayor de edad.

❌ **Observa la mala practica**

```
class UserValidator {
    boolean validate(User user) {
        if(user.age < 18) return false;
        return true;
    }
}
```

Tu jefe te informa que el sistema debe evolucionar y que ahora también se debe: verificar que el correo electrónico tenga un formato válido Te pide que **modifiques el código para añadir esta nueva validación**.

Preguntas:

1. ¿Qué ocurre si mañana se pide añadir otras validaciones (contraseña, numero de teléfono…)?
2. ¿Cuántas veces habría que modificar la clase UserValidator?
3. ¿Qué riesgos existen al modificar código que ya funciona?
4. ¿Como podriamos mejorar el código para no modificar la clase cada vez que una validacion se añade?

✅ **Observa la buena práctica:**

Creamos una clase común

```
interface ValidationRule {

    boolean validate(User user);

}
```

Y Implementaciones específicas

```java
class AgeValidation implements ValidationRule {

    public boolean validate(User user) {
        return user.age >= 18;
    }
}
```

```java
class EmailValidation implements ValidationRule {

    public boolean validate(User user) {
        return user.email.contains(s: "@");
    }

}
```

Preguntas:

1. ¿Por qué ya no es necesario modificar una clase existente para añadir una validación?
2. ¿Qué ventaja aporta tener una clase por cada regla?
3. ¿Cómo ayuda la interfaz **ValidationRule** a extender el comportamiento del sistema?
4. ¿Qué pasaría si mañana se añade **PasswordValidation**?
5. ¿Este diseño facilita el trabajo en equipo? ¿Por qué?

6. Descubre

El segundo de los principios SOLID dice: **Las clases deben estar abiertas a la extensión, pero cerradas a la modificación.**

Esto significa que:

- Podemos **añadir nuevas funcionalidades**
- **Sin modificar el código existente**

Este principio se llama:

Open / Closed Principle (OCP)

2.3 Liskov Substitution Principle (LSP)

Situación: Estás desarrollando una aplicación de dibujo que trabaja con **rectángulos**. El sistema calcula superficies en función del ancho y la altura, sin preocuparse del tipo exacto de figura.

```
class Rectangle {
    int width, height;

    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }

    int area() {
        return width * height;
    }
}
```

Para reutilizar código, decides crear un **cuadrado** como una subclase de Rectangle.

```
class Square extends Rectangle {
    void setWidth(int w) {
        width = height = w;
    }

    void setHeight(int h) {
        width = height = h;
    }
}
```

Preguntas:

1. ¿Puede un Square reemplazar siempre a un Rectangle sin errores?
2. ¿Qué suposición del código cliente deja de ser válida?
3. ¿Por qué este diseño puede generar errores difíciles de detectar?
4. ¿Qué alternativas de diseño podrías proponer?
    - ¿Usar una interfaz común?
    - ¿Evitar la herencia?

Uno de los principios SOLID dice:
Las subclases deben poder sustituir a sus clases base sin alterar el comportamiento esperado del programa.

👉 Este principio se llama:

Liskov Substitution Principle (LSP)

2.4 Interface Segregation Principle (ISP)

Situación: Estás diseñando un sistema de gestión de usuarios con diferentes perfiles:

- **UsuarioRegistrado** – puede iniciar sesión, registrarse y eliminar su cuenta
- **Administrador** – puede iniciar sesión, registrarse y eliminar cualquier cuenta
- **UsuarioInvitado** – solo puede registrarse

Cada tipo de usuario **tiene capacidades diferentes**.

❌ **Observa la mala practica**

```
interface UserActions {

    void login();
    void register();
    void deleteUser();

}
```

Preguntas:

1. ¿Por qué es un problema que UsuarioInvitado tenga que implementar métodos que no puede usar?
2. ¿Qué consecuencias tiene tener **métodos vacíos o con excepciones** en una clase?
3. ¿Cómo afectaría esto al mantenimiento del código y a la incorporación de nuevos tipos de usuarios?
4. ¿Qué podría pasar si otro desarrollador utiliza UsuarioInvitado esperando que pueda llamar a login() o deleteUser()?

✅ **Observa la buena práctica:**

```
interface UserAuthenticator {
    void login();
}

interface UserRegistrar {
    void register();
}

interface UserDeleter {
    void deleteUser();
}
```

5. Completa las clases **implementando solo los métodos que corresponden** a cada tipo de usuario:

```
class UsuarioInvitado implements UserRegistrar {
   public void ..... () { }
}
class UsuarioRegistrado implements UserAuthenticator, UserRegistrar {
   public void ..... () { }
   public void ..... () { }
}
class Administrador implements UserAuthenticator, UserRegistrar, UserDeleter {
   public void ..... () { }
   public void ..... () { }
   public void ..... () { }
}
```

Preguntas:

1. ¿Por qué una interfaz demasiado grande es un problema?
2. ¿Qué ocurre cuando una clase implementa métodos inútiles?
3. ¿Cómo mejora la legibilidad del código este diseño?
4. ¿Qué tipo de usuarios podrían implementar ambas interfaces?
5. Da un ejemplo del mundo real donde no todos los objetos tengan las mismas acciones.

6. **Descubres**

Uno de los principios SOLID dice: **Es mejor tener varias interfaces específicas que una única interfaz general.**

Este principio se llama:

Interface Segregation Principle (ISP)

2.5 Dependency Inversion Principle (DIP)

Situación: en la aplicación, existe un **controlador** que se encarga de manejar la lógica de usuario:

❌ **Observa la mala practica**

```
class UserController {

    MySQLUserRepository repo = new MySQLUserRepository();

}
```

Preguntas:

1. ¿Qué pasa si mañana quieres cambiar de base de datos MySQL por MongoDB con el diseño original?

✅ **Observa la buena práctica**

Separamos la abstracción de la implementación:

```
// Interfaz que define cómo acceder a los datos de usuario
interface UserRepository {
    void addUser(User user);
    User getUser(String id);
}
```

```java
// Implementación concreta de MySQL
class MySQLUserRepository implements UserRepository {

    public void addUser(User user) {
        /* implementación MySQL */
    }

    public User getUser(String id) {
        /* implementación MySQL */
        return null;
    }
}
```

```java
// Controlador depende de la abstracción, no de la implementación
class UserController {

    UserRepository repo;

    // Podemos inyectar la implementación que queramos
    UserController(UserRepository repo) {
        this.repo = repo;
    }
}
```

Preguntas

2. ¿Por qué es mejor que UserController dependa de una **interfaz (UserRepository)** en lugar de una clase concreta?
3. ¿Cómo facilita DIP la **realización de pruebas unitarias** del controlador?
4. ¿Qué otros beneficios tienen este enfoque en proyectos grandes y mantenibles?
5. Escribe un ejemplo de otra clase de tu aplicación que podría beneficiarse de aplicar DIP.

Descubres

Uno de los principios SOLID dice: Las clases deben depender de **abstracciones**, no de implementaciones concretas.

Este principio se llama:

Dependency Inversion Principle (DIP)