DAM 1 DAW 1 / Entornos de desarrollo

Optimización y documentación

4.1 Patrones de diseño

Autor: Lisa ERIKSEN
Fecha: 2025 / 2026

1. Introducción

Descubrir, comprender y aplicar los patrones de diseño y los principios SOLID para diseñar software flexible, mantenible y reutilizable en situaciones reales de desarrollo.

2. Introducción a los Patrones de Diseño

Actividad: Una empresa tiene una aplicación grande. Cada nuevo desarrollador implementa las soluciones **a su manera**.
Con el tiempo:

- El código es muy diferente entre módulos
- Nadie comprende completamente el sistema
- Cambiar algo provoca errores en otras partes

Preguntas:

1. ¿Cuáles de estos problemas aparecen en la situación?

   ☐ Confusión en el código
   ☐ Falta de reutilización
   ☐ Dificultad para hacer cambios
   ☐ Mejor rendimiento

2. ¿Qué crees que podría ayudar a mejorar este tipo de proyectos?
3. Lee: En informática existen **soluciones típicas que se reutilizan** una y otra vez para resolver problemas comunes de diseño.
   Esas soluciones tienen un nombre: **Patrones de diseño**.

4. Completa: Los patrones de diseño permiten _____ soluciones que ya funcionan bien.

3. Familias de Patrones de Diseño

3.1 Descubrimos los patrones

Actividad: Observa estos 3 casos reales:

Caso A

Una aplicación necesita crear distintos objetos (Usuario, Proveedor, Admin) según datos que llegan del servidor.

Caso B

Un proyecto usa una API antigua incompatible con el nuevo sistema.

Caso C

En una app de mensajería:
cuando un usuario envía un mensaje → todos reciben una notificación.

Preguntas:

1. Relaciona cada situación con lo que se intenta resolver:

Caso	Problema principal
Caso A	☐ Creación de objetos
Caso B	☐ Compatibilidad entre sistemas
Caso C	☐ Comunicación entre objetos

2. Lee: Existen **3 grandes familias de patrones**:
1. Creacionales
2. Estructurales
3. Comportamiento

3. Busca en internet y une cada familias al patrón siguiente: *Creación de objetos ; Adaptación de clases ; Comunicación múltiple.*

3.2 Patrones Creacionales: Factory Method

Actividad: Estás desarrollando el sistema de gestión de usuarios de una aplicación web. Existen **tres tipos de usuarios**: Admin ; Cliente ; Invitado. Cada uno muestra un mensaje distinto al iniciar sesión.

Ejemplo n°1: Clase sin patrón

- En VS Code, crea un archivo **Main.java** y escribe:

```java
public class Main {

    public static void main(String[] args) {

        String tipo = "admin";
        Usuario user;

        if (tipo.equals("admin")) {
            user = new Admin();
        }
        else if (tipo.equals("cliente")) {
            user = new Cliente();
        }
        else {
            user = new Invitado();
        }

        user.mostrarRol();

    }

}
```

1. ¿Qué problema aparece si se añade un nuevo tipo de usuario?
2. ¿Qué ocurre si este código se repite en 10 archivos distintos?

Ejemplo n°2: Factory Method

Vas a utilizar el patrón creacional **Factory Method** para centralizar la creación de usuarios y evitar múltiples "if" repartidos por el proyecto.

- Crea las siguientes clases Java en los siguientes archivos:

Usuario.java

```java
public abstract class Usuario {
    public abstract void mostrarRol();
}
```

Admin.java

```java
public class Admin extends Usuario {
    @Override
    public void mostrarRol() {
        System.out.println("Soy administrador");
    }
}
```

Client.java

```java
public class Cliente extends Usuario {
    @Override
    public void mostrarRol() {
        System.out.println("Soy cliente");
    }
}
```

Invitado.java

```
Completa el código de la clase
```

- Creación de la **Factory** la clase **UsuarioFactory** en el archivo UsuarioFactory.java

UsuarioFactory.java

```
public class UsuarioFactory {

    public static Usuario crear(String tipo) {

        if (tipo.equals("admin")) {
            return new Admin();
        }
        else if (tipo.equals("cliente")) {
            return new Cliente();
        }
        else {
            return new Invitado();
        }

    }

}
```

- Uso del patrón en la clase principal: modifica Main.java para usar la Factory:

```
public class Main {

    public static void main(String[] args) {

        // Cambia el tipo para probar distintos usuarios:
        String tipo = "admin";    // admin | cliente | invitado

        // Creación del usuario usando la Factory
        Usuario user = UsuarioFactory.crear(tipo);

        // Mostrar rol por consola
        user.mostrarRol();

    }

}
```

- Ejecuta el programa y prueba cambiando el valor de tipo.

Preguntas:

1. ¿Cuál de los dos ejemplos (con o sin Factory) es más fácil de mantener, por que?

3. Los **patrones creacionales** ayudan a:

   ☐ Crear objetos sin repetir código
   ☐ Centralizar la creación
   ☐ Facilitar ampliaciones

3. ¿Dónde se decide ahora qué objeto se crea?

4. Si se añade una nueva clase Premium, ¿en cuántos archivos tendrás que modificar el código?

5. Marca las ventajas obtenidas gracias al patrón:

   ☐ Código duplicado
   ☐ Centralización de la lógica
   ☐ Fácil ampliación del sistema
   ☐ Mayor mantenimiento

6. Completa:

El patrón **Factory Method** permite crear objetos de distintas clases sin conocer la implementación concreta en el código cliente, lo que facilita la _____ y el _____ del proyecto.

3.2 Patrones estructurales: Adapter

Situación: Una aplicación de música quiere reproducir archivos MP3. Pero una vieja biblioteca solo sabe reproducir CD.

La biblioteca funciona con **CDPlayer**

```
MusicPlayer player = new Mp3Adapter(new CDPlayer());
player.play("musica.mp3");
```

La app trabaja con **MP3Player**

```
CDPlayer reproductor = new CDPlayer();
reproductor.reproducirCD("musica.mp3");
```

💥 **Problema:**
El formato no es compatible

✅ **Solución:**
Creamos un **adaptador** que convierte el MP3 en algo que el CDPlayer entiende:

Con el patrón Adapter

```
MusicPlayer player = new Mp3Adapter(new CDPlayer());
player.play("musica.mp3");
```

El **Mp3Adapter** se encarga de traducir la llamada para que funcione con CD.

Preguntas:

1. ¿Cuál es la función del **adapter**?
☐ Traduce una interfaz a otra
☐ Cambia la biblioteca original
☐ Hace compatibles dos sistemas

2. Los patrones estructurales permiten:
☐ Conectar sistemas diferentes sin modificarlos
☐ Hacer que componentes incompatibles puedan trabajar juntos
☐ Obligar a cambiar todo el código

3.3 Patrones de comportamiento: Observer

Actividad: En una aplicación escolar: Cuando el profesor publica una nota, todos los alumnos inscritos reciben un aviso.

- Observa el código siguiente:

Alumno.java

```java
public class Alumno {

    private String nombre;

    public Alumno(String nombre) {
        this.nombre = nombre;
    }

    public void recibirAviso() {
        System.out.println(nombre + " ha recibido la notificación.");
    }
}
```

Profesor.java

```java
import java.util.ArrayList;

public class Profesor {

    private ArrayList<Alumno> alumnos = new ArrayList<>();

    public void agregarAlumno(Alumno a) {
        alumnos.add(a);
    }

    public void publicarNota() {
        for (Alumno a : alumnos) {
            a.recibirAviso();
        }
    }
}
```

Main.java

```java
public class Main {

    public static void main(String[] args) {

        Profesor profe = new Profesor();

        Alumno ana = new Alumno("Ana");
        Alumno david = new Alumno("David");

        profe.agregarAlumno(ana);
        profe.agregarAlumno(david);

        // Evento
        profe.publicarNota();

    }

}
```

Preguntas:

1. Completa
   - El objeto Profesor guarda una lista de…
   - Cuando ocurre un evento (**publicar nota**) se envía una notificación a …

2. Marca la respuesta correcta:
   ☐ Un alumno notifica al profesor
   ☐ Un profesor notifica a muchos alumnos
   ☐ Los alumnos se notifican entre ellos

3. El patrón utilizado para enviar un mismo aviso a muchos objetos es:

   ☐ Proxy
   ☐ Strategy
   ☐ Observer

4. Crea estas clases en VS Code y copia el codigo de arriba.

- Alumno.java
- Profesor.java
- Main.java

5. Modifica el programa para añade un nuevo alumno llamado **Carlos**

6. Regístralo usando **agregarAlumno()** y ejecuta el programa.

7. ¿Has tenido que cambiar el código de Profesor o Alumno para que Carlos reciba la notificación?

8. Completa la idea:

El patrón **Observer** permite que un objeto (el profesor) avise automáticamente a _____ cuando ocurre un evento.

9. Marca lo correcto. El patrón **Observer**:
   ☐ Facilita la comunicación entre objetos
   ☐ Evita código repetido
   ☐ Centraliza la notificación
   ☐ Obliga a conocer todas las clases