Patrones de Diseño

Los **patrones de diseño son** una serie de técnicas para resolver problemas concretos en el desarrollo de software. Normalmente se aplican cuando se identifica un problema y se considera que es posible adaptar un patrón para solucionarlo. Los patrones de diseño son bastante concretos en cuanto a la solución que ofrecen, por lo que no siempre son la respuesta adecuada. En ocasiones, pueden llegar incluso a descartarse porque no aportan beneficio alguno al programa, o su complicación es excesiva. Por otra parte, la correcta aplicación de un patrón de diseño tiene beneficios que a veces no son evidentes sino a muy largo plazo, como por ejemplo permitir en un futuro ampliar las funcionalidades del programa de forma sencilla.

Un patrón NUNCA de forzarse. La decisión de utilizar uno u otro patrón suele tomarse durante la fase de diseño de un proyecto, antes de que entren los programadores, y lo hacen los **analistas**.

Existen multitud de patrones, algunos genéricos incluso, pero todos ellos tienen un objetivo. Por ejemplo, el patrón **Modelo Vista Controlador (MVC)** es un patrón de arquitectura de software que separa el código en tres capas: la capa modelo, la capa controladora y la capa vista.

Patrón Singleton

Problema: Tenemos una clase importante en el proyecto, pero sólo debe de haber una única instancia (objeto) de esa clase en memoria.

Solución (en Java): La forma de conseguir hacer esto es mediante un uso adecuado de las variables tipo **static**. Hay varias formas de implementarlo, aquí te muestro solamente una de ellas.

```
public class Singleton {

    private static Singleton instance = null;

    public static Singleton getInstance() {
        if (null == instance)
            instance = new Singleton();
        return instance;
    }

    private Singleton () {
        // Some code...
    }

    // Some code...
}
```

Se define una variable estática **instance** que inicialmente es nula. Dado que el constructor es privado, no puede hacerse un **new Singleton ()**. En cambio, si que podemos llamar a **getInstance ()** dado que es un método estático.

El código de **getInstance ()** genera un objeto tipo Singleton que guarda en instance, pero sólo lo hace la primera vez que se llama a la función. Las demás veces, simplemente lo devuelve. De esta forma, podemos hacer desde cualquier sitio **getInstance ()** y siempre nos devolverá la única instancia de Singleton.

```
Singleton singleton1 = new Singleton();

Singleton singleton2 = Singleton.getInstance();
```

Patrón Factory

Problema: Tenemos una superclase (abstracta) con varias subclases. Lo que queremos hacer es generar una de esas subclases bajo demanda.

Solución (en Java): Vamos a dejar la responsabilidad de crear esas subclases a una clase específica llamada Factoría. Cada vez que queramos pedir un objeto de la subclase, se lo pedimos a la Factoría.

Por ejemplo, tenemos estas tres clases. Animal es la superclase, y Perro y Gato las subclases que queremos generar.

```
public abstract class Animal {

}
```

```
public class Gato extends Animal{

}
```

```
public class Perro extends Animal{

}
```

Vamos a crear ahora la factoría, capaz de crear Perros y Gatos:

```
public class AnimalFactory {

    public static String PERRO = "perro";
    private static String GATO = "gato";

    public Animal getAnimal(String type) {
        if (type == null) {
            return null;
        }
        if (type.equalsIgnoreCase(PERRO)) {
            return new Perro();
        }
        if (type.equalsIgnoreCase(GATO)) {
            return new Gato();
        }
        return null;
    }
}
```

Ahora, basta con pedir a AnimalFactory que nos genere una subclase:

```
AnimalFactory animalFactory = new AnimalFactory ();
Gato gato = (Gato) animalFactory.getAnimal (AnimalFactory.GATO);
```

Este patrón permite que nuestro código sea más robusto, menos acoplado y más fácil de extender. Si te das cuenta, sería muy fácil modificar el ejemplo para añadir cualquier tipo de animal sin que el “usuario” de la fábrica tenga que saber exactamente cómo funciona.