Anexo I.- Patrones de diseño

1. Introducción

Por muy específico que sea un problema, hay una probabilidad elevada que alguien se haya enfrentado a un problema muy similar en el pasado y se pueda modelar del mismo modo.

Con modelado me estoy refiriendo al hecho que la estructura de las clases que conforma la solución del problema puede estar ya inventada, porque estamos resolviendo un problema común que otra gente ya ha solucionado antes. Si la manera de solucionar este problema se puede extraer, explicar y reutilizar en múltiples ámbitos, entonces nos encontramos ante un **patrón de diseño de software** .

> Un patrón de diseño es una forma reutilizable de resolver un problema común.

El concepto de patrón de diseño lleva existiendo desde finales de los 70, pero su popularización surgió en los 90 con el lanzamiento del libro de Design *Pattern* de la Banda de los Cuatro (Gang of Four), nombre con el cual se conoce a los creadores de este libro: Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides. En él explican 23 patrones de diseño que son todo un referente.

2. ¿Por qué son útiles los patrones de diseño?

Puede parecer una tontería, pero si no le encontramos utilidad a las cosas, acabarmos por no usarlas. Los patrones de diseño son muy útiles por los siguientes motivos:

a) Ahorran tiempo

A una buena parte de los programadores nos encanta encontrar soluciones ingeniosas en un problema cuando estamos modelando software. El desarrollo es un reto mental que revierte en una gran satisfacción personal una vez que conseguimos un buen resultado.

Pero buscar siempre una nueva solución en los mismos problemas reduce nuestra eficacia como desarrolladores, porque perdemos mucho tiempo en el proceso. Hace falta no olvidar que el desarrollo de software también es una ingeniería, y que por lo tanto en muchas ocasiones habrá reglas comunes para solucionar problemas comunes.

Los patrones de diseño nos proporcionarán un conjunto de "comodines" (herramientas muy probadas y que funcionan bien), que nos van a permitir solucionar una buena parte de los problemas de forma directa.

b) Nos ayudan a estar seguros de la validez de nuestro código

Cuando creamos código nuevo, nos surge la duda de si es una solución correcta, o si realmente habrá una respuesta mejor. Se trata de una duda muy razonable y que en muchos casos la respuesta será la que no deseamos: sí que hay una solución más válida, y hemos perdido un tiempo valioso implementando algo que, aunque funciona, podría haberse modelado mejor.

Los patrones de diseño son estructuras probadas por millones de desarrolladores a lo largo de muchos años, por lo cual si elegimos el patrón adecuado para modelar el problema adecuado, podemos estar seguros que será una de las soluciones más válidas (si no la que más) que podamos encontrar.

c) Establecen un lenguaje común

Modelar el código mediante patrones nos ayudará a explicar a otros desarrolladores, conozcan nuestro código o no, como hemos abordado el problema.

También nos servirá para establecer una puesta en escena común de forma que todo el equipo pueda discurrir sobre como solucionar algo, y ponerse de acuerdo más rápido, es decir, explicar de forma más sencilla cuáles son nuestras ideas y que el resto las comprenda sin ningún problema. En definitiva, los patrones de diseño nos ayudarán a avanzar mucho más rápido, con un código más fácil de entender para todos y mucho más robusto.

3. ¿Cómo identificar qué patrón encaja con nuestro problema?

Este es el punto más complicado, y la respuesta es que se aprende practicando. La experiencia es la única forma válida de ser más hábil detectando donde nos pueden ayudar los patrones de diseño.

Por supuesto, hay situaciones conocidas en las cuales un patrón u otro nos puede ayudar, pero la experiencia y el estudio de los patrones (muy recomendable el libro de Head First Design Patterns, explican muy bien como usarlos en la vida real) son los que nos permitirán aplicar correctamente los patrones.

4. Listado de patrones de diseño

Los patrones, según el tipo de problema que resuelven, se dividen en diferentes grupos:

a) Patrones creacionales

Son los que facilitan la tarea de creación de nuevos objetos, de tal forma que el proceso de creación pueda ser desacoplado de la implementación del resto del sistema.

Los patrones creacionales están basados en dos conceptos:

1. Encapsular el conocimiento sobre los tipos concretos que nuestro sistema utiliza. Estos patrones normalmente trabajarán con interfaces, por lo cual la implementación concreta que utilizamos queda aislada.
2. Ocultar como estas implementaciones concretas necesitan ser creadas y como se combinan entre sí.

Los patrones creacionales facilitan la tarea de creación de nuevos objetos encapsulando el proceso.

Los patrones creacionales más conocidos son:

- **Abstract Factory**: El problema que intenta solucionar este patrón es el de crear diferentes familias de objetos. Nos provee una interfaz que delega la creación de un conjunto de objetos relacionados sin necesidad de especificar en ningún momento cuáles son las implementaciones concretas.
- **Factory Method**: Expone un método de creación, delegando en las subclases la implementación de este método.
- **Builder**: Separa la creación de un objeto complejo de su estructura, de tal forma que el mismo proceso de construcción nos puede servir para crear representaciones diferentes.
- **Singleton**: Limita a uno el número de instancias posibles de una clase en nuestro programa, y proporciona un acceso global a este.
- **Prototype**: Permite la creación de objetos basados en "plantillas". Un nuevo objeto se crea a partir de la clonación de otro objeto.

b) Patrones estructurales

Son patrones que nos facilitan la modelización de nuestro software especificando la forma en la cual unas clases se relacionan con otras.

Los patrones estructurales definidos por Gang of Four son:

- **Adapter**: Permite a dos clases con diferentes interfaces trabajar entre ellas, a través de un objeto intermedio con el cual se comunican e interactúan.
- **Bridge**: Desacopla una abstracción de su implementación, porque las dos puedan evolucionar de forma independiente.
- **Composite**: Facilita la creación de estructuras de objetos en árbol, donde todos los elementos emplean una misma interfaz. Cada uno de ellos puede, a su vez, contener un listado de estos objetos, o ser el último de esta rama.
- **Decorator**: Permite añadir funcionalidad extra a un objeto (de forma dinámica o estática) sin modificar el comportamiento del resto de objetos del mismo tipo.

- **Facade**: Una facade (o fachada) es un objeto que crea una interfaz simplificada para tratar con otra parte del código más compleja, de tal forma que simplifica y aísla su uso. Un ejemplo podría ser crear una fachada para tratar con una clase de una librería externa.
- **Flyweight**: Una gran cantidad de objetos comparte un mismo objeto con propiedades comunes con el fin de ahorrar memoria.
- **Proxy**: Es una clase que funciona como interfaz hacia cualquier otra cosa: una conexión a Internet, un archivo en disco o cualquier otro recurso que sea costoso o imposible de duplicar.

c) Patrones de comportamiento

En este último grupo se encuentran los patrones que se usan para gestionar algoritmos, relaciones y responsabilidades entre objetos.

- **Command**: Son objetos que encapsulan una acción y los parámetros que necesitan para ejecutarse.
- **Chain of responsibility**: Se evita acoplar al emisor y receptor de una petición dando la posibilidad a varios receptores de consumirlo. Cada receptor tiene la opción de consumir esta petición o pasarlo al siguiente dentro de la cadena.
- **Interpreter**: Define una representación para una gramática así como el mecanismo para evaluarla. El árbol de sintaxis del lenguaje se suele modelar mediante el patrón Composite.
- **Iterator**: Se utiliza para poder movernos por los elementos de un conjunto de forma secuencial sin necesidad de exponer su implementación específica.
- **Mediator**: Objeto que encapsula como otro conjunto de objetos interactúan y se comunican entre sí.
- **Memento**: Este patrón otorga la capacidad de restaurar un objeto en un estado anterior.
- **Observer**: Los objetos son capaces de subscribirse a una serie de acontecimientos que otro objetivo emitirá, y serán avisados cuando esto ocurra.
- **State**: Permite modificar la forma en que un objeto se comporta en tiempo de ejecución, basándose en su estado interno.
- **Strategy**: Permite la selección del algoritmo que ejecuta cierta acción en tiempo de ejecución.
- **Template Method**: Especifica el esqueleto de un algoritmo, permitiendo a las subclases definir como implementan el comportamiento real.
- **Visitor**: Permite separar el algoritmo de la estructura de datos que se utilizará para ejecutarlo. De esta forma se pueden añadir nuevas operaciones a estas estructuras sin necesidad de modificarlas.

5. Conclusiones

Este documento pretende ser solo un punto de partida que abra el camino hacia el entendimiento de los patrones de diseño. Es lógico que con la pequeña definición que hemos dado, no acabemos de entender para que sirven algunos.

Se puede considerar que el MVC (Modelo-Vista-Controlador) es un patrón de diseño, pero se trata de un

patrón arquitectónico que está en una capa de abstracción superior, es decir, su ámbito de aplicación es más amplio.

Existen otros muchos patrones de diseño además de los 23 que figuran en el libro de Design Patterns de la Gang of Four, pero los 23 que hemos definido son la base principal sobre la cual se han fundamentado otros muchos patrones.