# PATRONES DE DISEÑO

Por Antonio Leiva

## ¿Qué son los patrones de diseño?

Hay una cosa que está clara: por muy específico que sea un problema al que te estés enfrentando durante el desarrollo de tu software, hay un 99% de posibilidades (cifra totalmente inventada, pero seguro que muy real) de que **alguien se haya enfrentado a un problema tan similar en el pasado, que se pueda modelar de la misma manera**.

Con modelado me estoy refiriendo a que la estructura de las clases que conforma la solución de tu problema puede estar ya inventada, porque estás resolviendo un problema común que otra gente ya ha solucionado antes. Si **la forma de solucionar ese 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 verdadera popularización surgió en los 90 con el lanzamiento del libro de **Design Pattern** de la **Banda de los Cuatro** (Gang of Four), que aunque parezca que estamos hablando de los Trotamúsicos, es el nombre con el que se conoce a los creadores de este libro: Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. En él explican 23 patrones de diseño, que desde entonces han sido un referente.

## ¿Por qué son útiles los patrones de diseño?

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

1. Te ahorran tiempo

   Sé que te encantará encontrar una solución ingeniosa a un problema cuando estás modelando tu software, y es normal, a mí también me pasa. Como he comentado alguna vez, el desarrollo es un proceso casi artístico, y ese reto mental que supone revierte en una satisfacción personal enorme una vez que consigues un buen resultado.

   Pero hay que ser sinceros: **buscar siempre una nueva solución a los mismos problemas reduce tu eficacia como desarrollador**, porque estás perdiendo mucho tiempo en el proceso. No hay que olvidar que el desarrollo de software también es una ingeniería, y que por tanto en muchas ocasiones habrá reglas comunes para solucionar problemas comunes.

   Los patrones de diseño atajan ese punto. Una vez los conozcas, contarás con un conjunto de «trucos», de reglas, de herramientas muy probadas, que te permitirán **solucionar la mayor parte de tus problemas de forma directa**, sin tener que pensar en cómo de válidas son, o si puede haber una alternativa mejor.

2. Te ayudan a estar seguro de la validez de tu código
   Un poco relacionado con lo anterior, siempre que creamos algo nuevo nos surge la duda de si realmente estamos dando con la solución correcta, o si realmente habrá una respuesta mejor. Y el tema es que es una duda muy razonable y que en muchos casos la respuesta sea la que no deseas: sí que hay una solución más válida, y has perdido tu valioso tiempo en implementar 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 que, si eliges el patrón adecuado para modelar el problema adecuado, puedes estar seguro de que va a ser una de las soluciones más válidas (si no la que más) que puedas encontrar.

3. Establecen un lenguaje común
   Todas las demás razones palidecen ante esta. Modelar tu código mediante patrones te ayudará a explicar a otras personas, conozcan tu código o no, a entender cómo has atajado un problema. Además, ayudan a otros desarrolladores a comprender lo que has implementado, cómo y por qué, y además a descubrir rápidamente si esa era la mejor solución o no.

   Pero también te servirá para sentarte con tus compañeros a pensar sobre cómo solucionar algo, y poneros de acuerdo mucho más rápido, explicar de forma más sencilla cuáles son vuestras ideas y que el resto lo comprenda sin ningún problema. Los patrones de diseño os ayudarán a ti y a tu equipo, en definitiva, a avanzar mucho más rápido, con un código más fácil de entender para todos y mucho más robusto.

## ¿Cómo identificar qué patrón encaja con tu problema?

Desafortunadamente, tengo malas noticias… Este es el punto más complicado, y la respuesta más evidente, que es también la que menos nos gusta, es que se aprende practicando. La experiencia es la única forma válida de ser más hábil detectando dónde te pueden ayudar los patrones de diseño.

Por supuesto, hay situaciones conocidas en las que un patrón u otro nos puede ayudar, y las iré comentando a lo largo de los artículos. Además te recomiendo que te leas el libro de Head First Design Patterns, en el que además de explicarte los patrones de forma muy amena, explican muy bien cómo usarlos en la vida real.

Pero a partir de ese punto estás solo. Necesitarás conocer qué tipo de problemas soluciona cada uno y descubrir cómo aplicarlo a casos concretos. Como comentaba en el artículo de los miedos, en este caso lo mejor que te puede pasar es que encuentres a un compañero que los domine y que te haga de mentor. Pégate a él y exprímelo hasta que tengas todo su conocimiento. En caso contrario, practica, practica y práctica.

## Listado de patrones de diseño

Este es el listado de los patrones de diseño más conocidos. Poco a poco iré escribiendo artículos sobre cada uno de ellos y los iré enlazando aquí. Es un proceso largo porque son muchos y quiero encontrar la mejor forma de explicarlos, pero llegará el día en que tengamos aquí un listado completo que te sirva de referencia.

Los patrones se dividen en distintos grupos según el tipo de problema que resuelven, a saber:

## 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 acerca de los tipos concretos que nuestro sistema utiliza. Estos patrones normalmente trabajarán con interfaces, por lo que la implementación concreta que utilicemos queda aislada.
2. Ocultar cómo estas implementaciones concretas necesitan ser creadas y cómo se combinan entre sí.

## Los patrones creacionales más conocidos son:

- Abstract Factory: 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 al mismo.
- Prototype: Permite la creación de objetos basados en «plantillas». Un nuevo objeto se crea a partir de la clonación de otro objeto.

## Patrones estructurales

Son patrones que nos facilitan la modelización de nuestros softwares especificando la forma en la que unas clases se relacionan con otras.

## Estos son los patrones estructurales que definió la Gang of Four:

- Adapter: Permite a dos clases con diferentes interfaces trabajar entre ellas, a través de un objeto intermedio con el que se comunican e interactúan.
- Bridge: Desacopla una abstracción de su implementación, para que 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 esos objetos, o ser el último de esa 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.

## Patrones de comportamiento

En este último grupo se encuentran la mayoría de los patrones, y se usan para **gestionar algoritmos, relaciones y responsabilidades entre objetos**.

## Los patrones de comportamiento son:

- **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 esa petición o pasárselo 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 cómo otros conjuntos de objetos interactúan y se comunican entre sí.
- **Memento**: Este patrón otorga la capacidad de restaurar un objeto a un estado anterior
- **Observer**: Los objetos son capaces de suscribirse a una serie de eventos que otro objetivo va a emitir, 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 cómo 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.