Acceso a Datos, 2ºDAM
Samuel Amat Enguidanos

Principales patrones del software

ÍNDICE

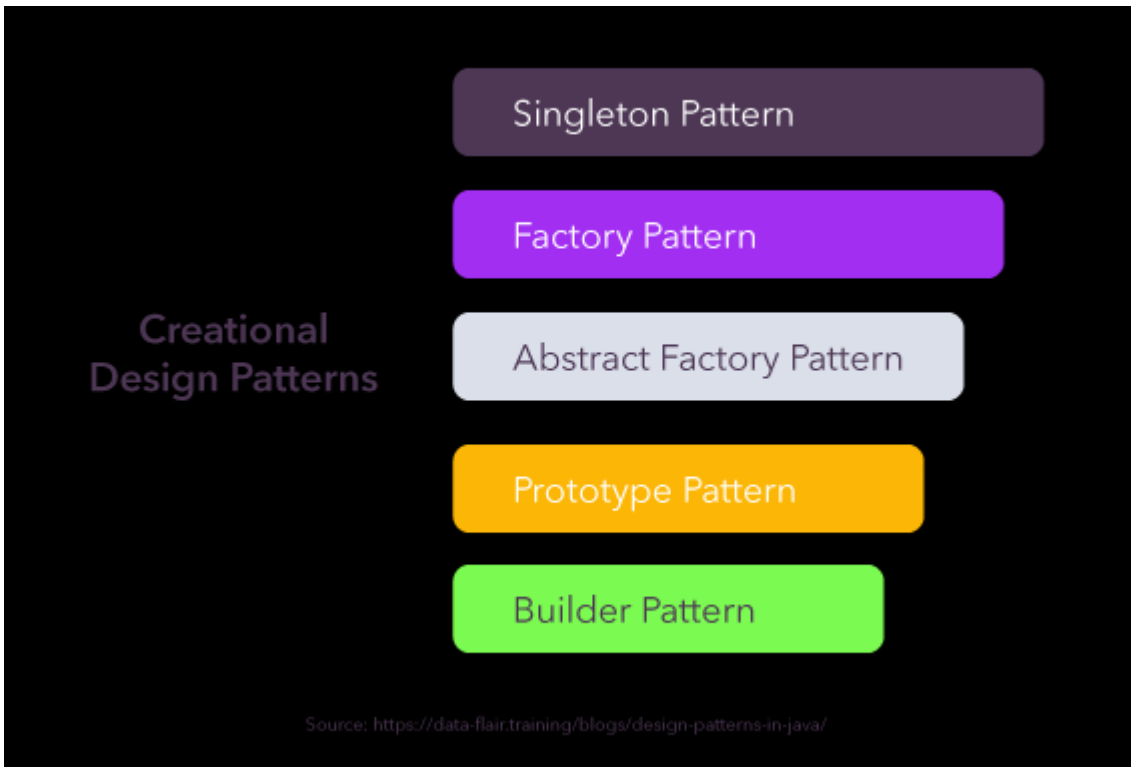
---

1. Introducción. ¿Qué son los patrones de diseño de software?

En informática, un patrón de diseño es una solución general para un problema que se presenta comúnmente en el diseño de software. Un patrón de diseño no es un diseño acabado que pueda transformarse directamente en código, es una descripción o plantilla de cómo resolver un problema que puede utilizarse en muchas situaciones diferentes. Están clasificados en tres principales categorías: Patrones de diseño **creacionales**, **estructurales** y **de comportamiento**. En los siguientes apartados veremos algunos de los principales patrones de dichas categorías y sus principales ventajas.

Los patrones de diseño pueden acelerar el proceso de desarrollo al proporcionar paradigmas de desarrollo probados y comprobados. Un diseño de software eficaz requiere tener en cuenta cuestiones que pueden no ser visibles hasta más tarde en la implementación. La reutilización de patrones de diseño ayuda a evitar problemas sutiles que pueden causar problemas importantes y mejora la legibilidad del código para los codificadores y arquitectos familiarizados con los patrones.

Además, los patrones permiten a los desarrolladores comunicarse utilizando nombres conocidos y bien entendidos para las interacciones del software. Los patrones de diseño comunes pueden mejorarse con el tiempo, lo que los hace más robustos.



2. Patrones creacionales

Descripción

Un patrón de diseño de creación se ocupa de la creación e inicialización de objetos, proporcionando orientación sobre los objetos que se crean para una situación determinada. Estos patrones de diseño se utilizan para aumentar la flexibilidad y reutilizar el código existente.

Principales patrones creacionales y ventajas

- **Factory Method**: Crea objetos con una interfaz común y permite que una clase difiera la instanciación de las subclases.
    - **Ventajas**: este patrón de diseño permite que el código interactúe directamente con la interfaz de modo que funcionará con cualquier clase que implemente dicha interfaz o herede de la clase abstracta.

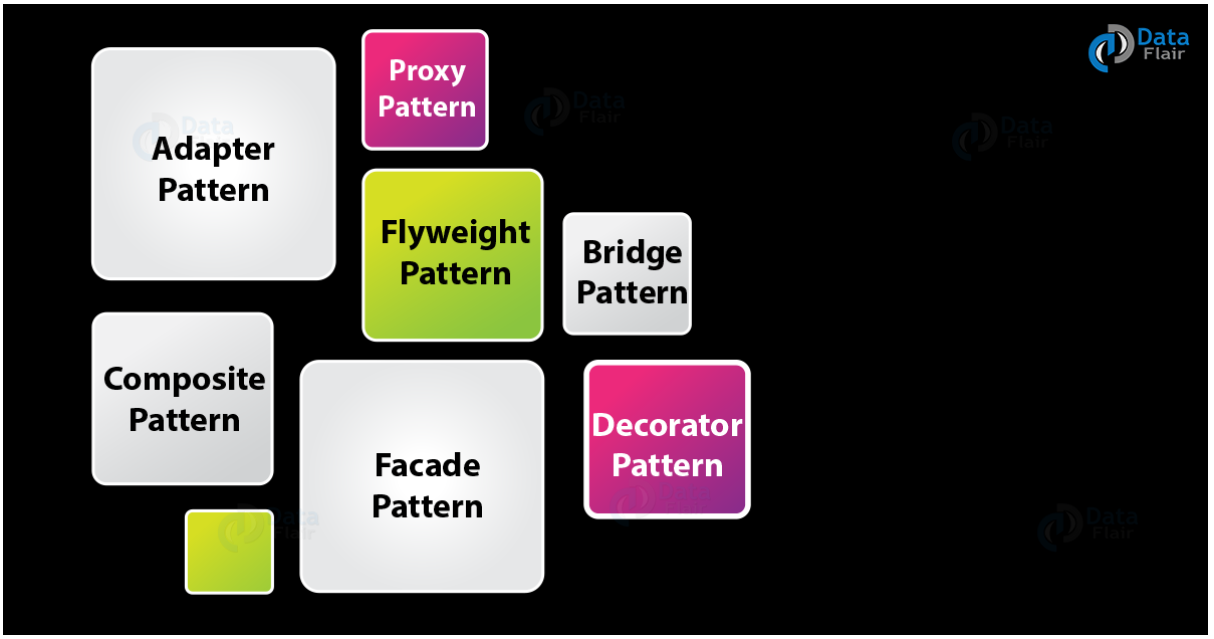
- **Abstract Factory**: Crea una familia de objetos relacionados.
    - **Ventajas**: El patrón Abstract Factory ayuda a controlar las clases de objetos que crea una aplicación. Dado que una fábrica encapsula la responsabilidad y el proceso de creación de objetos del producto, aísla a los clientes de las

clases de implementación. Los clientes manipulan las instancias a través de sus interfaces abstractas.

- **Builder**: Un patrón para crear objetos complejos, separando la construcción y la representación.
    - **Ventajas**: Proporciona una clara separación entre la construcción y la representación de un objeto. Proporciona un mejor control sobre el proceso de construcción. Permite cambiar la representación interna de los objetos.
- **Prototype**: Admite la copia de objetos existentes sin que el código dependa de las clases.
    - **Ventajas**: este patrón de diseño permite que el código interactúe directamente con la interfaz de modo que funcionará con cualquier clase que implemente dicha interfaz o herede de la clase abstracta.
- **Singleton**: Restringe la creación de objetos para una clase a una sola instancia.
    - **Ventajas**: Permite añadir o eliminar objetos en tiempo de ejecución. Se pueden conseguir objetos sin necesidad de conocer el tipo de objeto de antemano. Reduce la necesidad de hacer subclases.

---

3. Patrones estructurales



Descripción

Estos patrones tienen que ver con la composición de clases y objetos. Los patrones estructurales de creación de clases utilizan la herencia para componer interfaces. Los patrones estructurales de objetos definen formas de componer objetos para obtener nuevas funcionalidades.

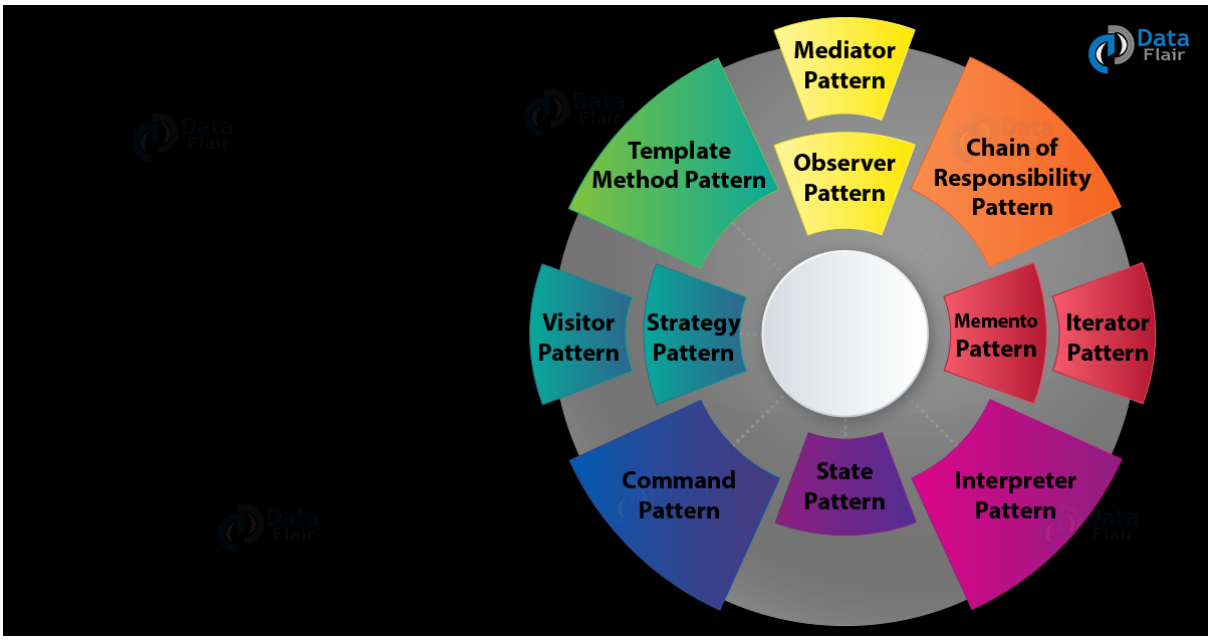
Principales patrones estructurales y ventajas

- **Adapter**: Cómo cambiar o adaptar una interfaz a la de otra clase existente para permitir que interfaces incompatibles funcionen juntas.
    - **Ventajas**: Nos permite utilizar código de terceros de forma relativamente fluida. No es necesario cambiar las interfaces existentes para utilizar el nuevo código
- **Bridge**: Un método para desacoplar una interfaz de su implementación.
    - **Ventajas**: Menos refactorización en el futuro ya que el número de clases no aumentará tanto. Tanto la abstracción como la implementación pueden desarrollarse de forma independiente.
- **Composite:** Una estructura de árbol de objetos simples y compuestos
    - **Ventajas**: Define jerarquías de clases que contienen objetos primitivos y complejos. Facilita la adición de nuevos tipos de componentes. Proporciona flexibilidad de estructura con clase o interfaz manejable.
- **Decorator:** Extiende dinámicamente (añade o sobrecarga) la funcionalidad.
    - **Ventajas**: Ampliación de la función de las clases sin herencia. Simplifica el código al permitir desarrollar una serie de funcionalidades a partir de clases dirigidas en lugar de codificar todo el comportamiento en el objeto.
- **Façade:** Describe una interfaz de alto nivel que facilita el uso del subsistema.
    - **Ventajas**: Protege a los usuarios de las complejidades de los componentes del subsistema. Promueve el acoplamiento flexible entre los subsistemas y sus clientes.
- **Flyweight:** Minimizar el uso de la memoria compartiendo datos con objetos similares.
    - **Ventajas**: Reduce el número de objetos y reduce la cantidad de memoria y dispositivos de almacenamiento necesarios si los objetos son persistentes.

- **Proxy:** Cómo representar un objeto con otro objeto para permitir el control de acceso, reducir el coste y reducir la complejidad.
    - **Ventajas**: Proporciona la protección al objeto original de agentes externos.

---

4. Patrones de comportamiento



Descripción

Estos patrones de diseño tienen que ver con la comunicación de los objetos de la clase. Los patrones de comportamiento son aquellos patrones que se refieren más específicamente a la comunicación entre objetos.

Principales patrones de comportamiento y ventajas

- **Chain of Responsibility:** En la cadena de responsabilidad, el emisor envía una solicitud a una cadena de objetos. La solicitud puede ser gestionada por cualquier objeto de la cadena.
    - **Ventajas**: Añade flexibilidad a la hora de asignar las responsabilidades a los objetos. Permite que un conjunto de clases actúe como una sola; los eventos producidos en una clase pueden ser enviados a otras clases con la ayuda de la composición.
- **Command:** Encapsula una solicitud de comando en un objeto.

    - **Ventajas**: Separa el objeto que invoca la operación del objeto del que realmente la realiza. Facilita la adición de nuevos comandos, ya que las clases existentes no se modifican.

- **Interpreter:** Básicamente, el patrón del intérprete tiene un área limitada en la que se puede aplicar. Este patrón puede aplicarse para analizar las expresiones definidas en sintaxis simples y, a veces, en motores de reglas simples.

    - **Ventajas**: Es más fácil cambiar y ampliar la sintaxis.

- **Iterator:** Accede secuencialmente a los elementos de una colección, el ejemplo más conocido es el de la interfaz java.util.Iterator ya que utiliza este patrón de diseño.

    - **Ventajas**: Admite variaciones en el recorrido de una Collection. Simplifica la interfaz de la Collection..

- **Mediator:** Define la comunicación simplificada entre clases. Ejemplo: Cuando comenzamos con el desarrollo, tenemos unas pocas clases y estas clases interactúan entre sí. Ahora, consideremos que poco a poco, la lógica se vuelve más compleja cuando la funcionalidad aumenta. Entonces, ¿qué sucede? Añadimos más clases y siguen interactuando entre sí, pero ahora se hace difícil mantener el código. Por lo tanto, el patrón mediador se encarga de este problema. El patrón mediador se utiliza para reducir la complejidad de la comunicación entre múltiples objetos o clases.

    - **Ventajas**: Los componentes individuales se simplifican y son mucho más fáciles de manejar porque no necesitan pasarse mensajes unos a otros.

- **Memento:** Un proceso para guardar y restaurar el estado interno/original de un objeto. Ejemplo: Deshacer o ctrl+z es una de las operaciones más utilizadas en un editor, este patrón de diseño se utiliza para implementar la operación de deshacer. Esto se hace guardando el estado actual del objeto a medida que cambia de estado. También se usa en transacciones de Bases de Datos.

    - **Ventajas**: Conserva los límites de la encapsulación.

- **Observer:** Define cómo notificar a los objetos los cambios en otros objetos

    - **Ventajas**: Proporciona soporte para la comunicación de tipo broadcast.

- **State:** Alterar el comportamiento de un objeto cuando su estado cambia
    - **Ventajas**: Hace explícitas las transiciones de estado.

- **Strategy:** Encapsula un algoritmo dentro de una clase.
    - **Ventajas**: Define cada comportamiento dentro de su propia clase, eliminando la necesidad de declaraciones condicionales. Facilita la ampliación e incorporación de nuevos comportamientos sin cambiar la aplicación.

- **Template Method:** Define el esqueleto de una operación al tiempo que permite a las subclases refinar ciertos pasos. Se utiliza cuando el comportamiento común entre subclases debe trasladarse a una única clase común evitando la duplicidad.
    - **Ventajas**: Es una técnica muy común para la reutilización del código.

---

5. Conclusión

Esto ha sido a grandes rasgos la explicación de los diferentes tipos de patrones de diseño junto a sus tres clasificaciones principales. Hay algunos muy conocidos como el patrón de diseño “Memento” ya que es uno que estamos muy acostumbrados a usar en el día a día, o por ejemplo el patrón Iterator el cual hemos visto en más de una ocasión en clase para usarlo en varios ejercicios de Java.

Los patrones de diseño aportan muchas ventajas y facilidades a la hora de programar a cambio de un mayor nivel de abstracción. A continuación enumero algunas de las ventajas que nos aportan:

1. Son reutilizables en múltiples proyectos.

2. Proporcionan soluciones que ayudan a definir la arquitectura del sistema.

3. Proporcionan transparencia al diseño de una aplicación.

4. Son soluciones bien probadas y atestiguadas, ya que se han construido sobre el conocimiento y la experiencia de desarrolladores de software expertos.

5. Los patrones de diseño no garantizan una solución absoluta a un problema.

6. Proporcionan claridad a la arquitectura del sistema y la posibilidad de construir un sistema mejor.

---

6. Webgrafía

https://sourcemaking.com/design_patterns

https://www.netsolutions.com/insights/software-design-pattern

https://www.javatpoint.com/design-patterns-in-java