PATRONES DE DISEÑO

Los patrones de diseño en Java son soluciones reutilizables y probadas a problemas comunes que se encuentran en el desarrollo de software. Estos patrones son una forma de capturar prácticas de diseño exitosas para que puedan ser aplicadas en diferentes contextos. Los patrones de diseño ayudan a escribir código más flexible, reutilizable y mantenible.

Los patrones de diseño en Java son, en definitiva, soluciones a problemas recurrentes y que se ha documentado que funcionan y los resuelven.

Clasificación de los Patrones de Diseño:

- **Patrones Creacionales:**

Los patrones de creación se utilizan para crear objetos para una clase adecuada que sirva como solución a un problema.

Algunos Patrones Creacionales son:

- **Abstract Factory**

Propósito: Proveer una interfaz para la creación de familias u objetos dependientes relacionados, sin especificar sus clases concretas.

Es una jerarquía que encapsula muchas familias posibles y la creación de un conjunto de productos. El objeto "fábrica" tiene la responsabilidad de proporcionar servicios de creación para toda una familia de productos. Los "clientes" nunca crean directamente los objetos de la familia, piden la fábrica que los cree por ellos.

Aplicación: Usamos el patrón Abstract Factory...

- Cuando tenemos una o múltiples familias de productos.
- Cuando tenemos muchos objetos que pueden ser cambiados o agregados durante el tiempo de ejecución.
- Cuando queremos obtener un objeto compuesto de otros objetos, los cuales desconocemos a que clase pertenecen.
- Para encapsular la creación de muchos objetos.

- **Singleton**

Propósito: Asegurar que una clase tenga una única instancia y proporcionar un punto de acceso global a la misma. El cliente llama a la función de acceso cuando se requiere una referencia a la instancia única.

Aplicación: Usamos el patrón Singleton...

- La aplicación necesita una, y sólo una, instancia de una clase, además está instancia requiere ser accesible desde cualquier punto de la aplicación.
- Típicamente para:
    - Manejar conexiones con la base de datos.
    - La clase para hacer Login.

- **Prototype**

Propósito: Especificar varios tipos de objetos que pueden ser creados en un prototipo para crear nuevos objetos copiando ese prototipo. Reduce la necesidad de crear subclases.

Aplicación: Usamos el patrón Prototype...

- Queremos crear nuevos objetos mediante la clonación o copia de otros.
- Cuando tenemos muchas clases potenciales que queremos usar sólo si son requeridas durante el tiempo de ejecución.

- **Patrones Estructurales:**

Los patrones estructurales forman estructuras más grandes a partir de elementos únicos, generalmente de diferentes clases.

Algunos Patrones Estructurales son:

- **Adapter**

Como cualquier adaptador en el mundo real este patrón se utiliza para ser una interfaz, un puente, entre dos objetos. En el mundo real existen adaptadores para fuentes de alimentación, tarjetas de memoria de una cámara, entre otros. En el desarrollo de software, es lo mismo.

Propósito: Convertir la interfaz (adaptee) de una clase en otra interfaz (target) que el cliente espera. Permitir a dos interfaces incompatibles trabajar en conjunto. Este patrón nos permite ver a nuevos y distintos elementos como si fueran igual a la interfaz conocida por nuestra aplicación.

Aplicación: Usamos el patrón Adapter...

- Cuando el cliente espera usar la interfaz de destino (target).
- Deseamos usar una clase existente pero la interfaz que ofrece no concuerda con la que necesitamos.

- **Composite**

Propósito: Componer objetos en estructuras de árbol que representan jerarquías de un todo y sus partes. El Composite provee a los clientes un mismo trato para todos los objetos que forman la jerarquía.

Pensemos en nuestro sistema de archivos, este contiene directorios con archivos y a su vez estos archivos pueden ser otros directorios que contenga más archivos, y así sucesivamente. Lo anterior puede ser representado fácilmente con el patrón Composite.

Aplicación: Usamos el patrón Composite...

- Cuando queremos representar jerarquías de objetos compuestas por un todo y sus partes.
- Se quiere que los clientes ignoren la diferencia entre la composición de objetos y su uso individual.

- **Decorator**

Extender la funcionalidad de los objetos se puede hacer de forma estática en nuestro código (tiempo de compilación) mediante el uso de la herencia, sin embargo, podría ser necesario extender la funcionalidad de un objeto de manera dinámica.

Propósito: Adjuntar responsabilidades adicionales a un objeto de forma dinámica. Los decoradores proporcionan una alternativa flexible para ampliar la funcionalidad.

Aplicación: Usamos el patrón Decorator...

- Cuando necesitamos añadir o eliminar dinámicamente las responsabilidades a un objeto, sin afectar a otros objetos.
- Cuando queremos tener las ventajas de la Herencia pero necesitemos añadir funcionalidad durante el tiempo de ejecución. Es más flexible que la Herencia,
- Simplificar el código agregando funcionalidades usando muchas clases diferentes.
- Evitar sobrescribir código viejo agregando, envés, código nuevo.

- **Patrones de Comportamiento:**

Los patrones de comportamiento describen interacciones entre objetos y se centran en cómo los objetos se comunican entre sí. Pueden reducir los diagramas de flujo complejos a simples interconexiones entre objetos de varias clases

Algunos patrones de comportamiento son:

- **Observer**

Propósito: Definir una dependencia de uno a muchos entre los objetos de manera que cuando un objeto cambia de estado, todos los que dependen de él son notificados y se actualizan automáticamente.

Los Observers se registran con el Subject a medida que se crean. Siempre que el Subject cambie, difundirá a todos los Observers registrados que ha cambiado, y cada Observer consulta al Subject que supervisa para obtener el cambio de estado que se haya generado.

En Java tenemos acceso a la clase Observer mediante java.util.Observer

Aplicación: Usamos el patrón Observer cuando...

- Un cambio en un objeto requiere cambiar los demás, pero no sabemos cuántos objetos hay que cambiar.
- Configurar de manera dinámica un componente de la Vista, envés de estáticamente durante el tiempo de compilación.
- Un objeto debe ser capaz de notificar a otros objetos sin que estos objetos estén fuertemente acoplados.

- **Command**

El patrón Command encapsula comandos (llamados a métodos) en objetos, permitiéndonos realizar peticiones sin conocer exactamente la petición que se realiza o el objeto al cuál se le hace la petición. Este patrón nos provee las opciones para hacer listas de comandos, hacer/deshacer acciones y otras manipulaciones.

Este patrón desacopla al objeto que invoca la operación del objeto que sabe cómo llevar a cabo la misma. Un objeto llamado Invoker transfiere el comando a otro objeto llamado Receiver el cual ejecuta el código correcto para el comando recibido.

Propósito: Encapsular una petición en forma de objeto, permitiendo de ese modo que parametrizar clientes con diferentes peticiones, "colas" o registros de solicitudes, y apoyar las operaciones de deshacer.

Aplicación: Usamos el patrón Command...

Cuando queremos realizar peticiones en diferentes tiempos. Se puede hacer a través de la especificación de una "cola".

Para implementar la función de deshacer (undo), ya que se puede almacenar el estado de la ejecución del comando para revertir sus efectos.

Cuando necesitemos mantener un registro (log) de los cambios y acciones.

Usos típicos:

- Mantener un historial de peticiones. (requests)
- Implementar la funcionalidad de callbacks.
- Implementar la funcionalidad de undo.

• Iterator

Propósito: Define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.

Algunos de los métodos que podemos definir en la interfaz Iterador son:

Primero(), Siguiente(), HayMas() y ElementoActual().

Aplicación: Usamos el patrón Iterator cuando ...

- Se desea acceder a los elementos de un contenedor de objetos (por ejemplo, una lista) sin exponer su representación interna.