PATRONES INTRODUCCIÓN A OBJETOS

QUE ES UN PATRÓN

Básicamente son soluciones recurrentes a problemas de diseño que se presentan comúnmente. Maneras convenientes de reutilizar código orientado a objetos entre proyectos y programadores. Se trata de anotar y catalogar aquellas interacciones comunes entre objetos que se encuentran frecuentemente y son útiles. Los patrones tratan problemas de diseño recurrente que aparecen en situaciones específicas del diseño y presentan soluciones al mismo.

Los buenos patrones deben solucionar un problema reiterativo, ser un concepto probado, describir participantes y sus relaciones, y no ser una solución obvia. Los patrones indican repetición, si algo no se repite, no es posible que sea un patrón. Un patrón se adapta para poder usarlo y es útil.

• Repetición: Característica cuantitativa pura. Es posible probarla demostrando la cantidad de veces que fue usada.
• Adaptabilidad: Característica cualitativa. (Cómo es exitoso el patrón)
• Utilidad: Característica cualitativa. (El patrón es exitoso y beneficioso)

Dependiendo de su finalidad pueden ser:

- **Patrones de creación**: utilizados para crear y configurar de clases y objetos.
- **Patrones estructurales**: su objetivo es desacoplar las interfaces e implementar clases y objetos. Crean grupos de objetos.
- **Patrones de comportamiento**: se centran en la interacción entre asociaciones de clases y objetos definiendo cómo se comunican entre sí.

VENTAJAS Y DESVENTAJAS

Ventajas de los patrones de Diseño

- Son soluciones concretas:
Un catálogo de patrones es un conjunto de recetas de diseño. Cada patrón es independiente del resto.
- Son soluciones técnicas:
Dada una situación, los patrones indican cómo resolverla.
- Se aplican en situaciones muy comunes:
Proceden de la experiencia y han demostrado su utilidad.
- Son soluciones simples:

Indican cómo resolver un problema utilizando un pequeño número de clases relacionadas de forma
determinada. No indican cómo diseñar un sistema completo, sino sólo aspectos puntuales del mismo.
- Facilitan la reutilización del código y del diseño:
Los patrones favorecen la reutilización de clases ya existentes y la programación de clases reutilizables.
La propia estructura del patrón se reutiliza cada vez que se aplica.

Desventajas de los patrones de Diseño

- Su uso no se refleja claramente en el código:
A partir de la implementación es difícil determinar qué patrón de diseño se ha utilizado. No es posible
hacer ingeniería inversa.
- Los equipos pueden padecer una sobrecarga de patrones
- Es difícil reutilizar la implementación del patrón:
El patrón describe roles genéricos, pero en la implementación aparecen clases y métodos concretos.

TIPOS DE PATRÓN (Composite, Observer, Adapter, State, Singleton**)**

COMPOSITE

Este útil patrón permite crear y manejar estructuras de objetos en forma de árbol, en las que un objeto puede contener a otros. Sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.

Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera. Dependiendo de la implementación, pueden aplicarse procedimientos al total o una de las partes de la estructura compuesta como si de un nodo final se tratara, aunque dicha parte esté compuesta a su vez de muchas otras.

VENTAJAS

- Define jerarquías de clases que consisten en objetos simples y en composiciones de esos objetos: Los objetos simples pueden ser compuestos en objetos más complejos que a su vez pueden ser compuestos por otros objetos compuestos y así recursivamente. En

cualquier lugar del código del cliente donde se necesite un objeto simple, también se podrá usar un objeto compuesto.

- Hace al cliente más simple: Los clientes pueden tratar los objetos simples y compuestos uniformemente. Los clientes normalmente no saben (y no les debería importar) si están tratando con una hoja (Leaf) o con un objeto Composite. Esto simplifica el código del cliente.
- El cliente puede esperar a un objeto simple o aceptar una composición, en cualquier lugar que se encuentre.
- Se trabaja con un diseño más general.
- Simplifica el código del cliente, porque se eliminan líneas de código repetitivas, para distinguir en qué caso se encuentra.
- Hace más fácil añadir nuevos tipos de componentes: Si se define una nueva clase Leaf o Composite, ésta funcionará automáticamente con la estructura que ya estaba definida y el cliente no tendrá que cambiar.

DESVENTAJAS

- Puede hacer nuestro diseño demasiado general: La desventaja de hacer más fácil el añadir nuevos componentes es que también hace más difícil restringir los componentes de una composición. A veces queremos que una composición tenga solo un determinado tipo de componentes. Con este patrón tendríamos que hacer las comprobaciones en tiempo de ejecución.

OBSERVER

Es un patrón de diseño de software que define una dependencia del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes. Se trata de un patrón de comportamiento (existen de tres tipos: creación, estructurales y de comportamiento), por lo que está relacionado con algoritmos de funcionamiento y asignación de responsabilidades a clases y objetos.

Los patrones de comportamiento describen no solamente estructuras de relación entre objetos o clases sino también esquemas de comunicación entre ellos y se pueden clasificar en función de que trabajen con clases (método plantilla) u objetos (cadena de responsabilidad, comando, iterador, recuerdo, observador, estado, estrategia, visitante).

La variación de la encapsulación es la base de muchos patrones de comportamiento, por lo que cuando un aspecto de un programa cambia frecuentemente, estos patrones definen un objeto que encapsula dicho aspecto. Los patrones definen una clase abstracta que describe la encapsulación del objeto.

Este patrón también se conoce como el patrón de publicación-inscripción o modelo-patrón. Estos nombres sugieren las ideas básicas del patrón, que son: el objeto de datos, que se le puede llamar Sujeto a partir de ahora, contiene atributos mediante los cuales cualquier objeto observador o vista se puede suscribir a él pasándole una referencia a sí mismo. El Sujeto mantiene así una lista de las referencias a sus observadores. Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el Sujeto es capaz de notificar a sus observadores suscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el Sujeto, ejecutado, por ejemplo, por alguno de los observadores, el objeto de datos puede recorrer la lista de observadores avisando a cada uno. Este patrón suele utilizarse en los entornos de trabajo de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir listeners a los objetos que pueden disparar eventos.

VENTAJAS

- La principal ventaja de este patrón es que todo se logra sin recurrir a un acoplamiento estrecho. El sujeto solo sabe de una lista de observadores y en una sola llamada los notifica. Al sujeto no le interesa los efectos o desenlaces de los observadores, él simplemente emite. El resultado es código flexible y reusable.
- Permite agregar un número ilimitado de Observadores que desean enterarse del cambio.
- El uso de este patrón te permite desacoplar el subjet (Observable) de todos aquellos observadores (Observer).

DESVENTAJAS

- La única desventaja apreciable, aparece cuando un observador es demasiado grande. Eso puede traer consecuencias en el uso de memoria.
- Otra posible desventaja aparece cuando se inscribe un observador que a su vez sea un sujeto para otros objetos. Si la implementación no es limpia, será muy difícil corregir o encontrar errores cuando estos sucedan.
- Hay que tener cuidado con la implementación de los Observer porque podría complicar la performance si son muchos o si está mal implementada.

ADAPTER

Este patrón permite que trabajen juntas clases con interfaces incompatibles.

Para ello, un objeto adaptador reenvía al otro objeto los datos que recibe (a través de los métodos que implementa, definidos en una clase abstracta o interface) tras manipularlos en caso necesario.

Convierte la interfaz de una clase en otra interfaz que el cliente espera. El adaptador permite a las clases trabajar juntas, lo que de otra manera no podría hacerse debido a sus interfaces incompatibles.

VENTAJAS

- Hace que dos interfaces incompatibles, sean compatibles. Puede servir para encapsular clases que no controlamos, y que pueden cambiar.
- Facilidad para redefinir el comportamiento de la clase adaptada.
- Simplicidad (un sólo objeto, no hay punteros ni indirecciones adicionales).
- Flexibilidad para que un sólo adaptador trabaje con muchas clases a adaptar (en concreto, puede hacerlo con toda una jerarquía de clases).
- Extensibilidad, puesto que se pueden añadir funcionalidades a todas las clases adaptadas a la vez.

DESVENTAJAS

- Inflexibilidad, puesto que un sólo adaptador no puede trabajar con una clase y sus hijos a la vez.
- Dificultad para redefinir el comportamiento de la clase adaptada.
- Como muchos patrones, añade complejidad al diseño. Hay quién dice que este patrón es un parche, utilizado en malos diseños.

STATE

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Busca que un objeto pueda reaccionar según su estado interno. Si bien muchas veces esto se puede solucionar con un boolean o utilizando constantes, esto suele terminar con una gran cantidad de if-else, código ilegible y dificultad en el mantenimiento. La intención del State es desacoplar el estado de la clase en cuestión.

En determinadas ocasiones se requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra. Esto resulta complicado de manejar, sobre todo cuando se debe tener en cuenta el cambio de comportamientos y estados de dicho objeto, todos dentro del mismo bloque de código. El patrón State propone una solución a esta complicación, creando un objeto por cada estado posible.

Este patrón debe ser utilizado cuando:

- El comportamiento de un objeto depende de un estado, y debe cambiar en tiempo de ejecución según el comportamiento del estado.
- Cuando las operaciones tienen largas sentencias con múltiples ramas que depende del estado del objeto.

VENTAJAS

- Se localizan fácilmente las responsabilidades de los estados específicos, dado que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior. Esta facilidad la brinda el hecho que los diferentes estados están representados por un único atributo (state) y no envueltos en diferentes variables y grandes condicionales.
- Aumenta las posibilidades de extensión.
- Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables, mientras que aquí al estar representado cada estado.
- Aumenta la cohesión.
- Los objetos State pueden ser compartidos si no contienen variables de instancia, esto se puede lograr si el estado que representan esta enteramente codificado en su tipo. Cuando se hace esto estos estados son Flyweights sin estado intrínseco.
- Facilita la ampliación de estados
- Permite a un objeto cambiar de clase en tiempo de ejecución dado que al cambiar sus responsabilidades por las de otro objeto de otra clase la herencia y responsabilidades del primero han cambiado por las del segundo.
- Eliminar código if/else.
- Reduce duplicación al eliminar código innecesario.
- Facilita las pruebas de la aplicación.

DESVENTAJAS

- Se incrementa el número de subclases.
- Demasiado código
- Difícil mantenimiento
- Posibles incongruencias
- Encapsula un comportamiento dependiente del estado (State) en que nos encontremos se va a comportar de una manera o de otra.

SINGLETON

El patrón de diseño Singleton (soltero) recibe su nombre debido a que sólo se puede tener una única instancia para toda la aplicación de una determinada clase, esto se logra restringiendo la libre creación de instancias de esta clase mediante el operador new e imponiendo un constructor privado y un método estático para poder obtener la instancia.

La intención de este patrón es garantizar que solamente pueda existir una única instancia de una determinada clase y que exista una referencia global en toda la aplicación.

VENTAJAS

- Acceso controlado a la única instancia: Encapsula su única instancia, puede tener un control estricto sobre cómo y cuándo acceden a ella los clientes.
- No se necesita pasar la referencia a todos los objetos que la necesiten, simplificando el desarrollo y haciendo la aplicación más mantenible.
- Espacio de nombres reducido: Es una mejora sobre las variables globales. Evita contaminar el espacio de nombres con variables globales que almacenan las instancias.
- Permite el refinamiento de operaciones y la representación: Se puede crear una subclase de la clase Singleton, y es fácil configurar una aplicación con una instancia de esta clase extendida, incluso en tiempo de ejecución.
- Permite un número variable de instancias: Hace que sea fácil permitir más de una instancia de la clase. Solo se necesitaría cambiar la operación que otorga acceso a la instancia del Singleton.
- La clase singleton es la única que puede crear objetos de la clase, asegurando la unicidad.

DESVENTAJAS

- Puede tener problemas en aplicaciones con muchos hilos de ejecución y con una única instancia.
- Si en el sistema evoluciona y necesitan más instancias de la clase, habría que cambiar todos los accesos a la clase singleton.