Patrones de Diseño de Software

Belizario Mamani, Loja Mamani, Lizárraga Paolo, Llanque Miguel

July 21, 2020

Abstract

There is one thing that is clear:as specific as it is a problem that you are facing during the development of your software, there is a 99 % chance that someone has faced such a similar problem in the past, that it can be model in the same way.
With modeling we mean that the class structure that makes up the solution to your problem may already have been invented, because you are solving a common problem that other people have already solved before. If the way to solve this problem can be extracted, explained and reused in multiple areas, then we are faced with a software design pattern.

1. Introduccion

En este artículo veremos los patrones de diseño de software que si eres programador seguro que has oído hablar de ellos. Es posible incluso, que ya los estés utilizando en tus aplicaciones. Los patrones de diseño nos ayudan a cumplir muchos principios o reglas de diseño.

2. RESUMEN

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 de que alguien se haya enfrentado a un problema tan similar en el pasado, que se pueda modelar de la misma manera.

Con modelado nos referimos 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

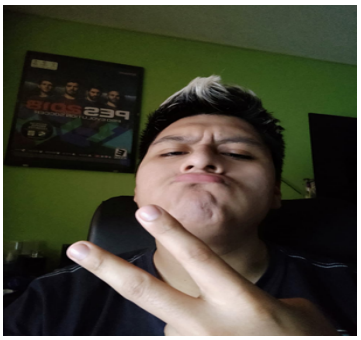
3. TITULO

Patrones de diseño de software

4. AUTORES

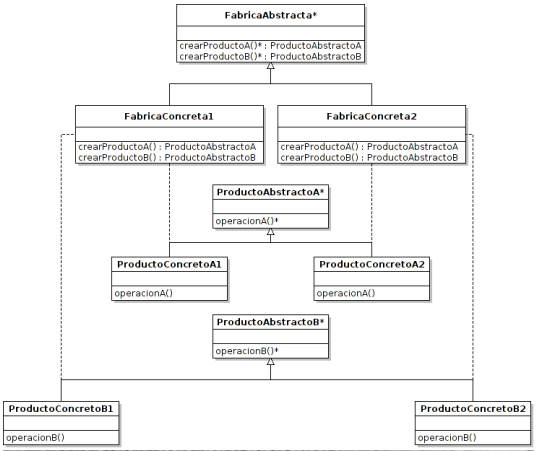
Los autores de este trabajo final de unidad son:

- Paolo Lizárraga



- Anthony Belizario

- Daniel Loja



- Miguel LLanque

V.DESARROLLO

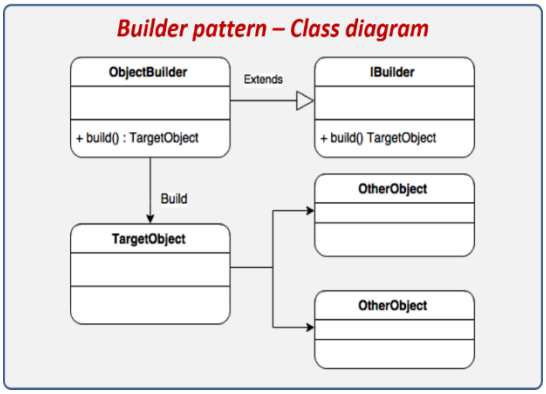
1) Patrones Creacionales

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.

Estos nos proveen soluciones para la creación de objetos, permitiéndonos hacer un sistema independiente de cómo sus objetos son creados.

Los Patrones creacionales más conocidos:

- **Patrón Abstract Factory**



Definición: Es un patrón de diseño para el desarrollo de software. Provee una interfaz para crear familias de objetos relacionados o dependientes entre ellos sin especificar una clase en concreto.

Este patrón se puede aplicar cuando:

- Un sistema debe ser independiente de cómo sus objetos son creados.
- Un sistema debe ser 'configurado' con una cierta familia de productos.
- Se necesita reforzar la noción de dependencia mutua entre ciertos objetos.

Participantes:

Fabrica Abstracta∗ : Define un conjunto de métodos (interfaz) para la creación de productos abstractos.

FabricaConcreta1/2: Implementa la interfaz de la Fabrica Abstracta para la creación de los distintos productos concretos.

ProductoAbstractoA∗ /B∗ : Define la interfaz de los objetos de tipo ProductoA/B.

ProductoConcretoA1/A2/B1/B2: Implementan su respectiva interfaz representando un producto concreto.

- **Patrón Builder**

Definición: Este es un patrón bastante simple pero muy útil, el cual nos permite crear objetos complejos a través de uno más simple. Es muy común encontrarnos con situaciones en las cuales tenemos que crear objetos compuestos de forma manual y repetidas veces, lo que nos lleva a tener que establecer cada propiedad del objeto y si ésta además tiene objetos compuestos dentro, tenemos que crearlos primero para después ser asignados al objeto que estamos creando. Esto desde luego que se hace una tarea tediosa y cansada, sobre todo cuando tenemos que crear de manera frecuente los objetos.

Participantes:

ObjectBuilder: Esta es la clase que utilizaremos para crear los TarjetObjet, esta clase debe de heredar de IBuilder e implementar el método build, el cual será utilizado para crear al TarjetObject. Como regla general todos los métodos de esta clase retornan a si mismo con la finalidad de agilizar la creación, esta clase por lo general es creada como una clase interna del TargetObject.

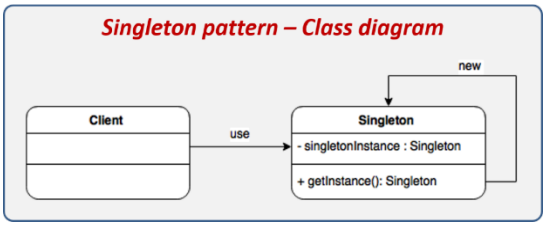
TarjetObjet: Representa el objeto que deseamos crear mediante el ObjectBuilder, ésta puede ser una clase simple o puede ser una clase muy compleja que tenga dentro más objetos.

OtherObjets: Representa los posibles objetos que deberán ser creados cuando el TarjetObject sea construido por el ObjectBuilder.

- **Patrón Singleton**

Definición: 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.



Participantes:

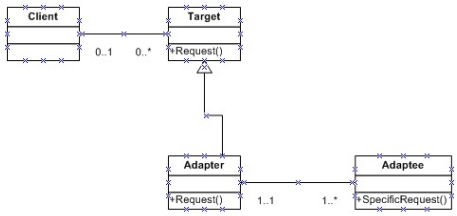
- **Client**: Componente que desea obtener una instancia de la clase Singleton.
- **Singleton**: Clase que implementa el patrón Singleton, de la cual únicamente se podrá tener una instancia durante toda la vida de la aplicación.

5. Patrones Estructurales

Los patrones estructurales se enfocan en como las clases y objetos se componen para formar estructuras mayores, los patrones estructurales describen como las estructuras compuestas por clases crecen para crear nuevas funcionalidades de manera de agregar a la estructura flexibilidad y que la misma pueda cambiar en tiempo de ejecución lo cual es imposible con una composición de clases estáticas.

Entre los más conocidos elegimos 3, los cuales son:

Patrón Adapter

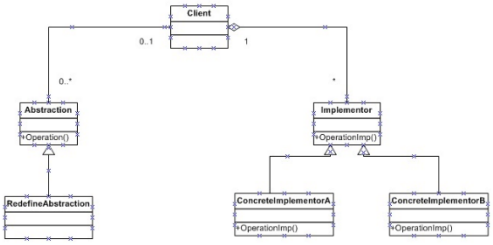


Definición: Convierte la interfaz de una clase en otra que espera un cliente.
Participantes:

Target: Define la clase que es esperada por el cliente.
Adapter: Adapta la clase a otra esperada por el cliente.
Adaptee: Define la clase que necesita a ser adaptada.
Client: Consume el servicio target.

Este patrón se usa comúnmente en los casos en los que queremos utilizar una clase cuya interfaz no coincide con nuestras necesidades.

Patrón Brigde
Definición: Desacopla una clase abstracta de su implementación final de manera de que ambas puedan ser independientes.



Participantes:

Abstraction: Define una interfaz abstracta y mantiene una referencia a un objeto del tipo Implementor.

RefinedAbstraction: Extiende de la interfaz definida por la clase Abstraction.

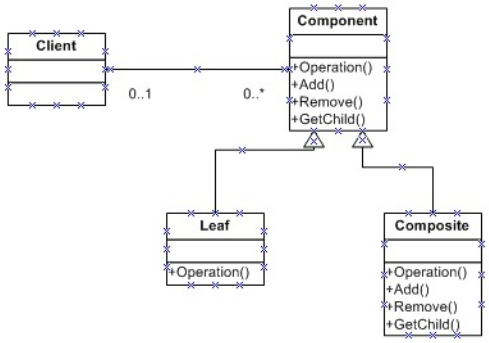
Implementor: Define la interfaz para la clase que realiza la implementación, la misma no tiene una correspondencia exacta con la interfaz Abstraction, de hecho, ambas interfaces pueden ser completamente diferentes.

ConcreteImplementor: Implementa la interfaz Implementor y define una implementación en concreto.

- **Patrón Composite**

Definición: Este patrón construye objetos dentro de una estructura de árbol, de esta manera los objetos que componen el árbol pueden tratarse de manera individual o como una composición de objetos.

Como podemos ver en este patrón los objetos pueden estar compuestos por objetos más complejos que a su vez pueden ser también objetos compuestos y así sucesivamente, sin embargo, un cliente podría manejar una estructura compuesta o una simple de manera uniforme y sin necesidad de saber si están tratando con objetos compuestos o no.



Participantes:

Component: Declara una interfaz que implementa los objetos en el árbol, define el comportamiento por defecto de los objetos en el árbol y la manera en la que se acceden y se administran los hijos de la composición.

Leaf: Representa una rama de objetos dentro del árbol o composición, una Leaf no tiene hijos y define el comportamiento de objetos primitivos en la composición.

Composite: Define el comportamiento para los hijos que posee y almacena los mismos.

Client: Manipula los objetos de la composición a través de la interfaz de los componentes. Este patrón permite que un cliente pueda interactuar con los objetos que componen toda una estructura.

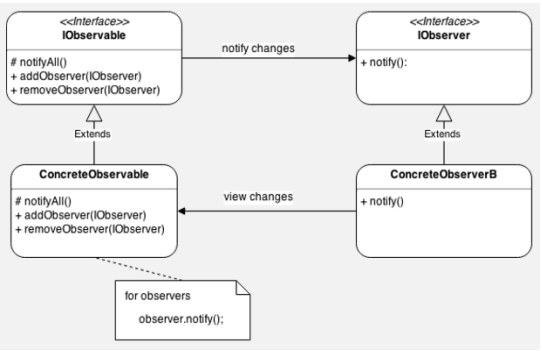
Como podemos ver en este patrón los objetos pueden estar compuestos por objetos más complejos que a su vez pueden ser también objetos compuestos y así sucesivamente, sin embargo, un cliente podría manejar una estructura compuesta o una simple de manera uniforme y sin necesidad de saber si están tratando con objetos compuestos o no.

6. Patrones de comportamientos

Se centran en las formas de como interactúan y como se reparten las responsabilidades las distintas clases y objetos.

- **Patrón Observer**

Definicion: El patrón de diseño Observer permite observar los cambios producidos por un objeto, de esta forma, cada cambio que afecte el estado del objeto observado lanzará una notificación a los observadores; a esto se le conoce como Publicador-Suscriptor. Observer es uno de los principales patrones de diseño utilizados en interfaces gráficas de usuario (GUI), ya que permite desacoplar al componente gráfico de la acción a realizar.



Participantes:

IObservable: Interface que deben de implementar todos los objetos que quieren ser observados, en ella se definen los métodos mínimos que se deben implementar.

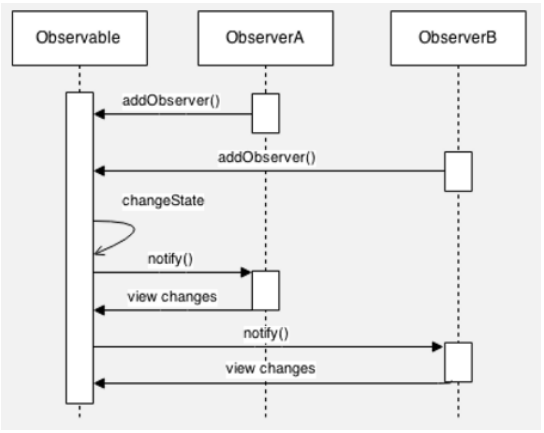
ConcreteObservable: Clase que desea ser observada, ésta implementa IObservable y debe implementar sus métodos.

IObserver: Interfaces que deben implementar todos los objetos que desean observar los cambios de IObservable.

ConcreteObserver: Clase concreta que está atenta de los cambios de IObserver, esta clase hereda de IObserver y debe de implementar sus métodos.

El patrón de diseño Observer es parecido al patrón Mediator, si bien en él una clase central encapsula y dirige la comunicación generada entre los demás objetos.

Funcionamiento:



1. El ObserverA se registra con el objeto Observable para ser notificado de algún cambio.

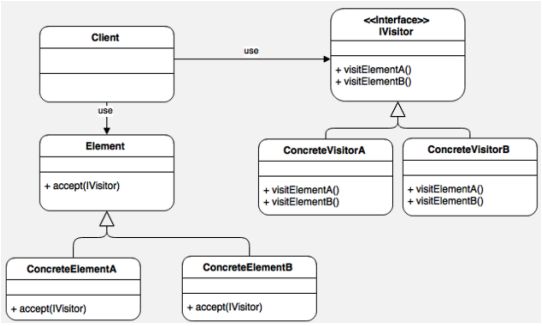
1. El ObserverB se registra con el objeto Observable para ser notificado de algún cambio.

1. Ocurre algún cambio en el estado del Observable.

1. Todos los Observers son notificados con el cambio ocurrido.

- **Patrón Vistor**

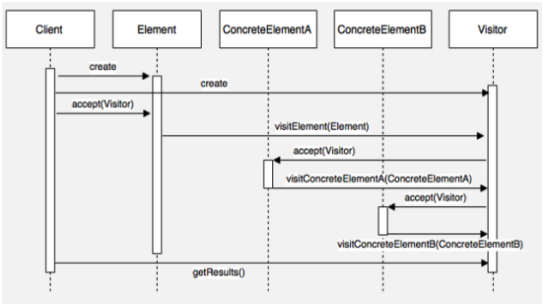
Definicion: El patrón de diseño Visitor se utiliza para separar la lógica u operaciones que se pueden realizar sobre una estructura compleja. En ocasiones nos podemos encontrar con estructuras de datos que requieren realizar operaciones sobre ella, pero estas operaciones pueden ser muy variadas e incluso se pueden desarrollar nuevas a medida que la aplicación crece.



Participantes:
Cliente: Componente que interactúa con la estructura (element) y con el Visitante, éste es responsable de crear los visitantes y enviarlos al elemento para su procesamiento.
Element: Representa la raíz de la estructura, en forma de árbol, sobre la que utilizaremos el Visitante. Este objeto por lo general es una interface que define el método accept y deberán implementar todos los objetos de la estructura.

ConcreteElement: Representa un hijo de la estructura compuesta, la estructura completa puede estar compuesta de un gran número de estos objetos y cada uno deberá implementar el método accept.
IVisitor: Interface que define la estructura del visitante, la interface deberá tener un método por cada objeto que se requiera analizar de la estructura (element).
ConcreteVisitor: Representa una implementación del visitante, esta implementación puede realizar una operación sobre el element. Es posible tener todos los ConcreteVisitor necesarios para realizar las operaciones que necesitemos.

Funcionamiento:



1. El *cliente* crea la estructura (Element).

1. El cliente crea la instancia del Visitante a utilizar sobre la estructura.

1. El cliente ejecuta el método accept de la estructura y la envía al Visitante.

1. El Element le dice al Visitante con que método lo debe procesar. El Visitante deberá tener un método para cada tipo de clase de la estructura.

1. El Visitante analiza al Element mediante su método visitElement y repite el proceso de ejecutar el método accept sobre los hijos del Element. Nuevamente el Visitante deberá tener un método para procesar cada clase hija de la estructura.

1. El ConcreteElementA le indica al Visitante con qué método debe procesarlo, el cual es visitElementA.

1. La visitante continúa con los demás hijos de Element y esta vez ejecuta el método accept sobre el ConcreteElementB.

1. El ConcrteElementB le indica al Visitante con qué método debe procesarlo, el cual es visitElementB.

1. Finalmente, el Visitante termina la operación sobre la estructura cuando ha recorrido todos los objetos, obteniendo un resultado que es solicitado por el cliente mediante el método getResults (el resultado es opcional ya que existen operaciones que no arrojan resultados).

.

VI. CONCLUCIONES

- Los patrones de diseño ayudan a estandarizar el código.

- Hacen que el diseño sea más comprensible para otros programadores.

- Estamos mal si pensamos usarlos porque los demás lo hacen.

- Los Patrones de diseño son una solución general para problemas conocidos, pero se adaptan de acuerdo a contextos específicos.

- La mayor complejidad, que representan, es saber cuándo utilizarlos.

7. RECOMENDACIONES

8. Si no usas patrones, deberías hacerlo. Los patrones ayudan a estandarizar el código, haciendo que el diseño sea más comprensible para otros programadores. Son muy buenas herramientas, y como programadores, siempre deberíamos usar las mejores herramientas a nuestro alcance.

9. De nada vale aplicar patrones sin una buena razón.

VIII. BIBLIOGRAFIA

Los Patrones de Diseño Hoy: Patrones de Comportamiento | Koalite. (2018). Recuperado 21 Julio 2020, de: https://blog.koalite.com/2016/12/los-patrones-de-diseno-hoy-patrones-de-comportamiento/

Observer. (2019). Recuperado 21 Julio 2020, de: https://reactiveprogramming.io/blog/es/patrones-de-diseno/observer

Patrón de diseño Command (comportamiento). (2019). Recuperado 21 Julio 2020, de: https://informaticapc.com/patrones-de-diseno/command.php

Patrón de diseño Command (comportamiento). (2019). Recuperado 21 Julio 2020, de: https://informaticapc.com/patrones-de-diseno/command.php

Patrones de diseño creacionales. (2016). Recuperado 21 Julio 2020, de: https://ed.team/blog/patrones-de-diseno-creacionales

Patrones Estructurales. (2019). Recuperado 21 Julio 2020, de: https://highscalability.wordpress.com/2010/04/12/patrones % C2 % A0estructurales/

Programacion en Castellano, S. (2018). Patrones de Diseño (XIX): Patrones de Comportamiento - Mediator. Recuperado 21 Julio 2020, de:

https://programacion.net/articulo/patrones_de_diseno_xix_patrones_de_comportam

2020). Recuperado 21 Julio 2020, de: http://siul02.si.ehu.es/~alfredo/iso/06Patrones.pdf