Desarrollo de contendido

Unidad 2

Programación Orientada a Objetos II

IU Digital de Antioquia
INSTITUCIÓN UNIVERSITARIA DIGITAL DE ANTIOQUIA

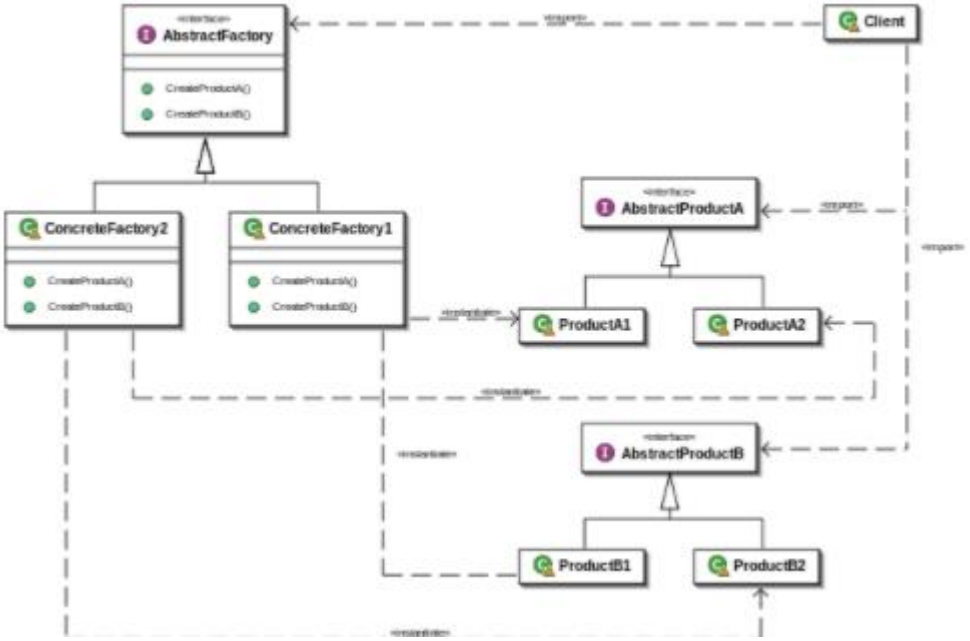
ORD No 74 de 2017 · VIGILADA MinEducación

www.iudigital.edu.co

Unidad 2. Patrones de diseño

El término patrón fue empleado en el campo de la arquitectura, por Christopher Alexander a finales de los años 70. Este conocimiento fue transferido al ámbito del desarrollo de software orientado a objeto.

Los últimos avances en el área de desarrollo de software orientado a objeto, fueron los patrones, con los que se buscaba resolver problemas de ingeniería de software haciendo uso del concepto de reutilización. Con los patrones, se busca crear un lenguaje común que permita comunicar todas las experiencias sobre los problemas y las soluciones.



Con los patrones, se busca que ciertas características se repitan; si no se repiten, entonces es posible que no estemos hablando de patrón, pero algo que se debe tener en cuenta es, que la repetición no es la única característica, es importante también demostrar que este patrón se pueda adaptar al momento de ser utilizado y que sea útil.

Cuando hablamos de repetición, hacemos referencia a una característica cuantitativa, y cuando decimos que debe ser adaptable y útil, nos referimos a una característica

cualitativa. La repetición es aplicada si usamos la regla de tres (mínimo existen 3 sistemas), la adaptabilidad se muestra cuando decimos cómo nuestro patrón es un éxito, y la utilidad se hace evidente cuando explicamos porqué este patrón es un éxito y es beneficioso.

Los tres tipos de patrones a desarrollar en esta Unidad, son:

- Patrones creacionales: estos patrones nos ayudan a independizar los objetos creados y/o representados del sistema en mención.
- Patrones estructurales: estos patrones nos permiten organizar las clases y objetos de tal manera, que podamos crear estructuras y sistemas más complejos.
- Patrones de comportamiento: estos patrones se centralizan en el algoritmo, y en cómo son asignadas las responsabilidades entre los objetos.

Ahora, no todas las soluciones que tengan las características de un patrón, son un patrón; por eso se debe validar que la solución planteada que se da en un problema que se repite, para que sea considerado como un patrón, se debe someter a un test de patrón y, mientras esto sucede, se considera entonces un proto-patrón.

¿Sabías qué?

Existen diferentes definiciones de lo que es un patrón de software. A continuación, te enumeramos algunas de ellas:

Según Dirk Riehle y Heinz Zullinghoven, un patrón es la abstracción de una forma concreta que puede repetirse en contextos específicos.

Un patrón es una información que captura la estructura esencial y la perspicacia de una familia de soluciones, probadas con éxito para un problema repetitivo que surge en un cierto contexto y sistema.

Un patrón es una unidad de información nombrada, instructiva e intuitiva, que captura la esencia de una familia exitosa de soluciones probadas a un problema recurrente dentro de un cierto contexto.

Los patrones deben tener ciertas características, según lo planteó James Coplien (escritor, conferenciante e investigador en el campo de la informática). Haz clic en cada numeral para conocerlas.

Solución a un problema: con los patrones se capturan soluciones, no solo principios o estrategias abstractas.

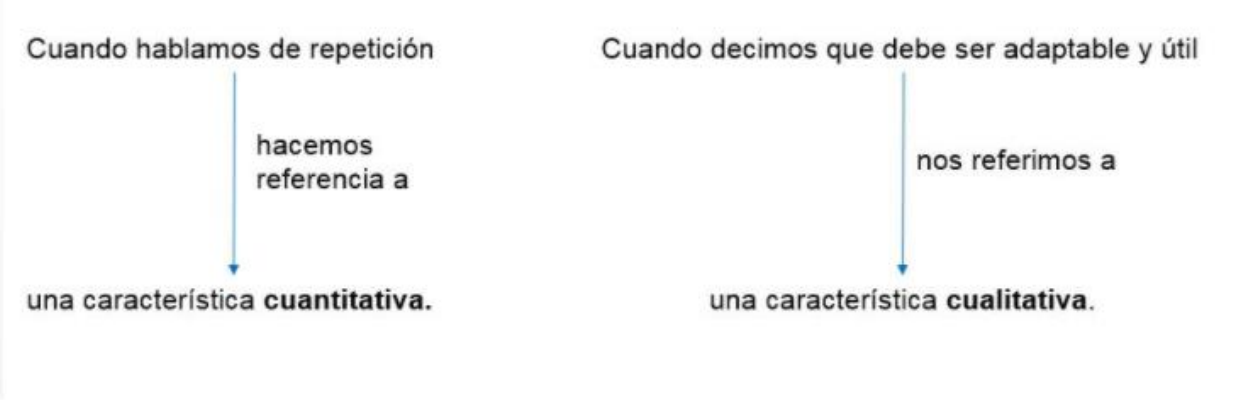
El concepto debe estar probado: con los patrones se deben tener soluciones demostradas, no teóricas y/o especulaciones.

La solución no es obvia: algunas de las técnicas de solución de los problemas, buscan encontrar soluciones a través de los principios básicos. Los mejores patrones obtienen una solución a un problema, de forma indirecta.

Descripción de los participantes y relación entre ellos: con los patrones se describen las estructuras del sistema y los mecanismos más complejos.

Componente humano significativo en los patrones: varios tipos de software brindan al ser humano confort y/o calidad de vida (estética y utilidad).

Lo que buscamos con los patrones es que ciertas características se repitan; si no se repiten, entonces muy posiblemente no estamos hablando de patrón; pero algo que se debe tener en cuenta es, que la repetición no es la única característica; es importante también demostrar que este patrón se pueda adaptar al momento de ser usado, y que sea útil



La repetición es aplicada si usamos la regla de tres (mínimo 3 existen sistemas); la adaptabilidad se muestra cuando decimos cómo nuestro patrón es un éxito, y la utilidad se hace evidente, cuando explicamos porqué este patrón es un éxito y es beneficioso.

Los patrones están compuestos por cuatro elementos fundamentales. Haz clic en cada uno para conocerlos.

Nombre del patrón: con el nombre se describe un problema de diseño, su solución y su consecuencia y se debe expresar en una o dos palabras, lo que permite que se tenga un diseño de un alto nivel de abstracción

Problema del patrón: aquí se debe describir cuándo debe ser aplicado un patrón. Se debe explicar el problema y todo su contexto. En algunas ocasiones, el problema incluirá una lista de condiciones que se deben cumplir para poder aplicar el patrón.

Solución del patrón*:* aquí deben describirse los elementos que hacen parte del diseño, la relación que existe entre ellos, la responsabilidad de los mismos y la colaboración existente. Con la solución no se describe una solución o una implementación, porque los patrones son como una plantilla que se aplica a una solución en particular; los patrones sí nos proporcionan una descripción abstracta del problema de diseño, y como puede ser solucionado.

Consecuencias del patrón: aquí obtenemos los resultados que se deben aplicar al patrón. Estas consecuencias son importantes para evaluar el diseño alternativo y para la comprensión de los costos y beneficios de la aplicación del patrón.

La ingeniería de software cuenta con diferentes ámbitos donde se pueden aplicar los patrones.

Patrones de diseño: Nos entregan un esquema que permite clarificar cada uno de los subsistemas o componentes que tienen los sistemas de software o las relaciones existentes entre ellos.

Patrones organizacionales: Permiten describir las estructuras y prácticas de las organizaciones humanas, especialmente aquellas en las que se produce, utiliza o administra software.

Patrones de análisis: Nos permiten describir el conjunto de prácticas que nos facultan para asegurar la obtención de un modelo del problema y su solución.

Patrones de programación: Describen cómo se pueden implementar los aspectos o relaciones entre ellos, utilizando las facilidades de un lenguaje dado.

Patrones de arquitectura: Permiten una organización o esquema estructural para los sistemas de software, proporcionando un conjunto de subsistemas predefinidos; se especifican responsabilidades, e incluye una organización de las relaciones entre ellos.

NOTA: Estos patrones se diferencian en su nivel de abstracción, detalle, y contexto en particular, en el cual están siendo aplicados durante el desarrollo del proceso.

En esta Unidad nos enfocaremos en los patrones de diseño; estos patrones nos permiten solucionar problemas específicos y nos ayudan para que los diseños de programación orientados a objetos, sean más flexibles y reutilizables, permitiendo así poder reutilizar los diseños para dar solución a otros desarrollos. Cada uno de los patrones de diseño permite mostrar, explicar y evaluar, la importancia del diseño en los sistemas orientados a objetos, de tal manera que se puedan agrupar todas estas experiencias, para luego ser usados con una mayor efectividad. Estos patrones ayudan a la elección de diseños alternativos que permiten que el sistema sea reutilizable, e incluso se pueden mejorar en la documentación y mantenimiento de los sistemas que ya existen.

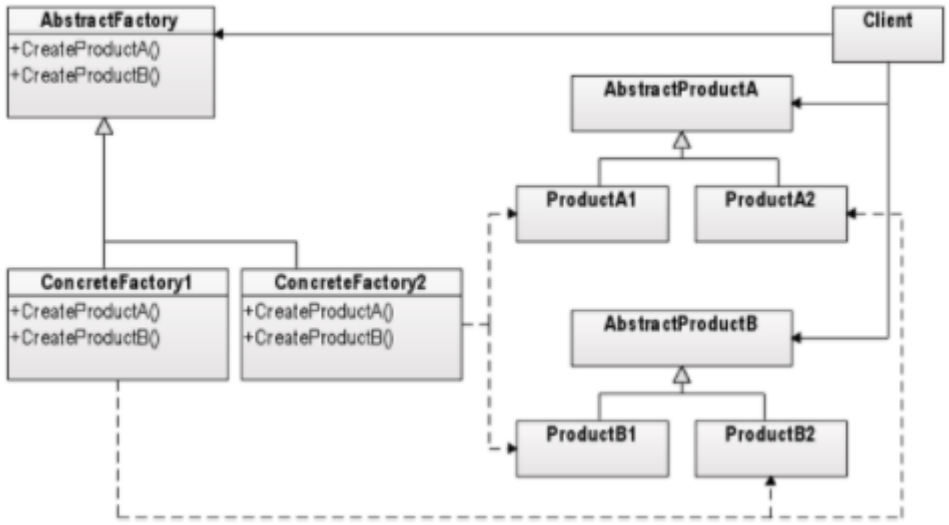
Los patrones de diseño contienen un nivel de abstracción; estos no son diseños como si se creara una lista, o una tabla que puede ser codificada en una clase y luego ser reutilizada. Los patrones de diseño describen la comunicación de objetos y clases que se personalizan para dar solución a un problema en un contexto particular. Cada patrón se divide en secciones que están de acuerdo con plantillas; estas plantillas nos muestran la

estructura para poder organizar la información, y para que estos patrones sean fáciles de entender, aprender y sobre todo, de adaptar y utilizar.

Fábrica abstracta

Cuando se tienen diferentes clases abstractas relacionadas, el patrón Abstract Factory, permite instanciar estas clases desde el conjunto de subclases concretas. Este patrón contiene una interfaz que permite crear familia de objetos que están relacionados o son dependientes, sin que se sepa a qué clase pertenecen.

Este patrón es útil cuando se necesita trabajar con un sinnúmero de clases o entidades externas, y se requiere construir objetos individuales para un único propósito, sin que se repitan en la construcción e identifiquen la clase a instanciar.



Este patrón debe usarse cuando:

- Unas familias de productos relacionados están hechas para utilizarse juntos.
- Un sistema debe configurarse con una o más familias de productos.
- Se necesita reforzar la noción de dependencia mutua entre ciertos objetos.

- Un sistema debe ser independiente de cómo son sus objetos.

Abstract factory: Se tiene la declaración de una interfaz que permite la creación de objetos de productos abstractos.

Concrete factory: Se implementan operaciones que permiten la creación de objetos de productos concretos.

Abstract product: Se declara una interfaz para los objetos de un tipo de producto.

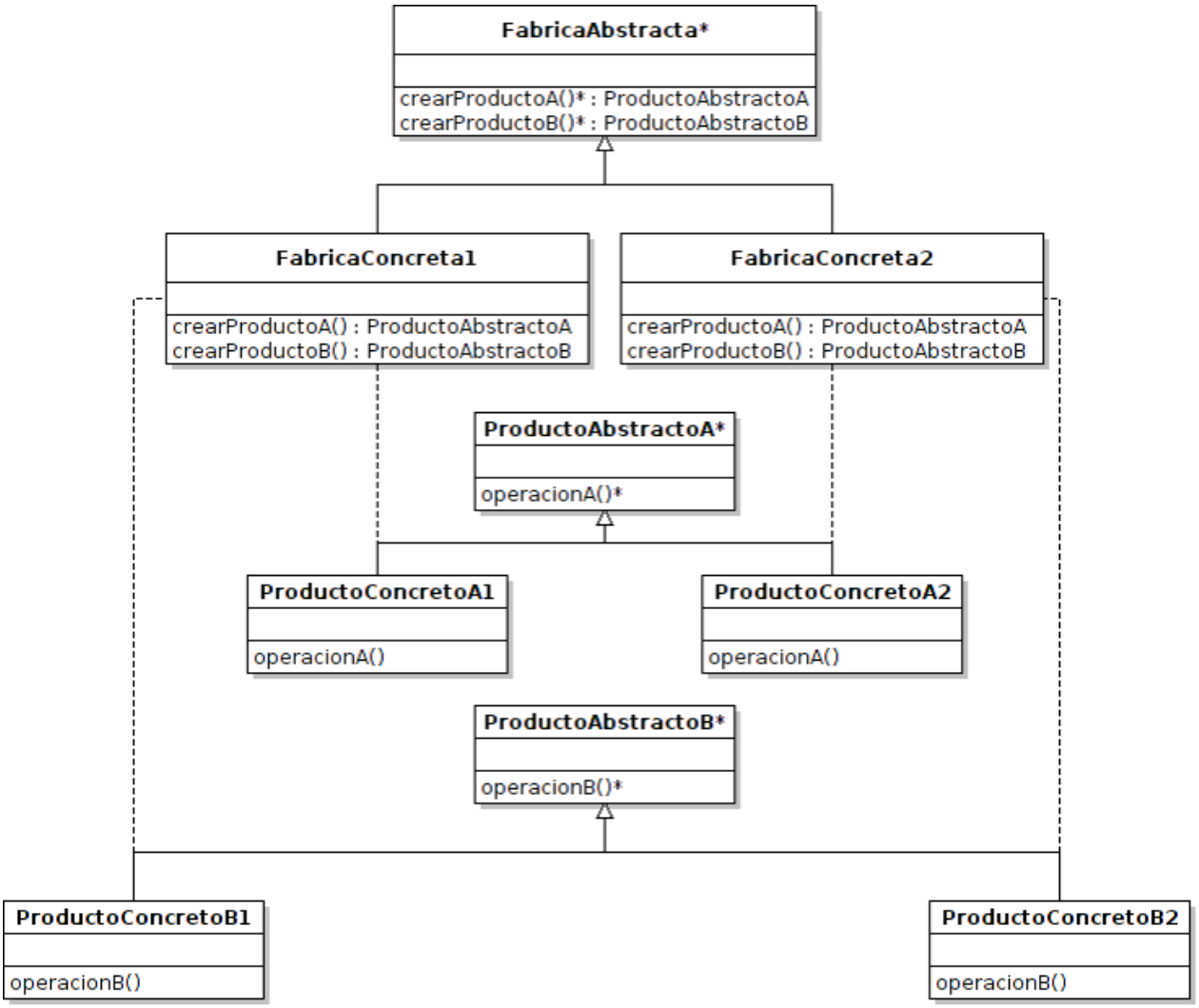
Concrete product: Define un objeto de producto que la factoría concreta que corresponde, se encargaría de crear; al mismo tiempo, se implementa la interfaz de producto abstracto.

Client: Se utilizan únicamente interfaces declaradas en la factoría y en los productos abstractos.

NOTA: En tiempo de ejecución se crea una única instancia de cada FactoryConcreto. AbstractFactory delega la creación de productos a sus subclases FactoryConcreto.

En el siguiente diagrama se puede evidenciar que FactoryConcreto1 crea una relación entre un producto de la familia A y un producto de la familia B. También tenemos que el FactoryConcreto2 crea una relación entre otros dos productos pertenecientes a ambas familias. Por tanto, decimos que se crea una clase por cada relación que necesitemos crear.

Las ventajas que se tienen en la utilización de este patrón es que nos brinda una flexibilidad para aislar a las clases concretas y nos facilita cambiar las familias de los productos; como desventaja tenemos que, para agregar un nuevo producto, debemos modificar primero tanto la fábrica abstracta, como la concreta.



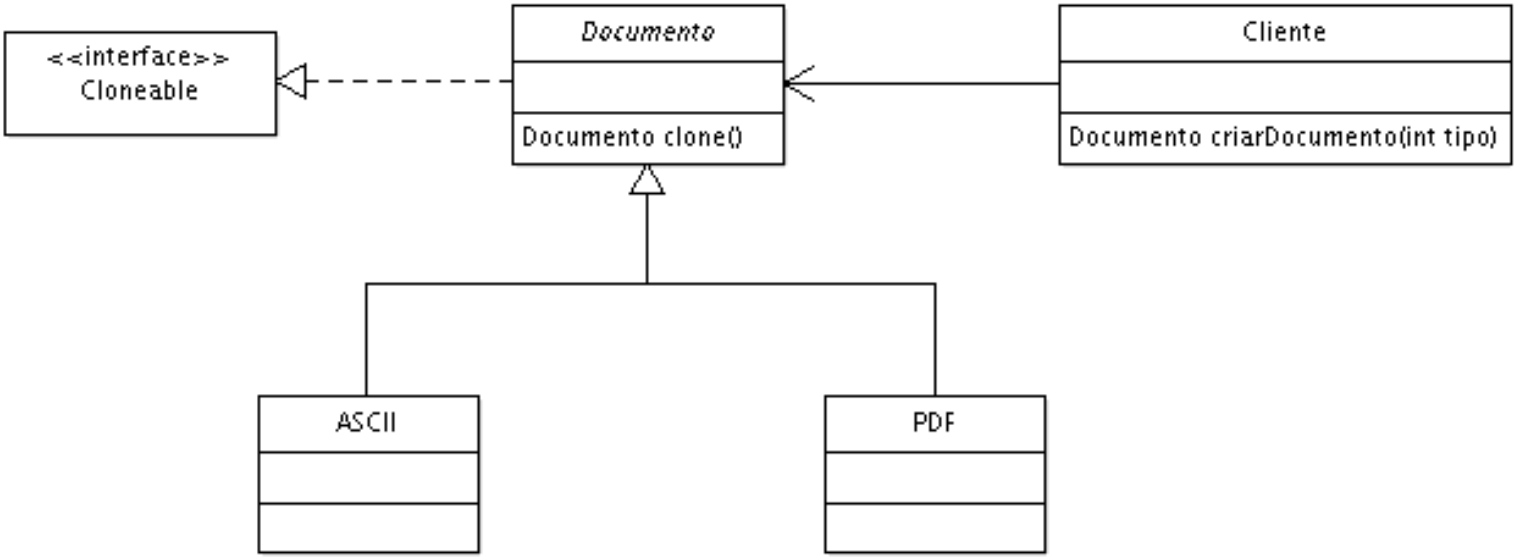
Constructor virtual

Este patrón permite construir objetos complejos por medio del uso de objetos simples y utilizando un enfoque paso a paso. Este tipo de patrón viene bajo el enfoque de patrón creacional debido a que proporciona una de las mejores formas de crear un objeto. Una clase de constructor, construye el objeto final paso a paso. Este constructor es independiente de otros objetos.

El patrón Builder permite la simplificación en la creación de un objeto. De igual manera, nos ayuda en la simplificación del código porque no tiene que llamar a un constructor complejo o llamar a varios métodos setter en el objeto creado. El patrón del generador se puede usar para crear una clase inmutable.

Prototipo (Prototype)

El objetivo principal del patrón Prototype es crear a partir de un modelo, permitiendo crear objetos prediseñados sin conocer el detalle de cómo debe ser creado; todo esto se da por medio de prototipos del objeto que se desea crear y, en realidad, estos objetos son clonados; es importante dejar en claro que la finalidad es crear nuevos objetos, duplicando o clonando una instancia creada previamente.



El uso de este patrón está dado a escenarios donde se necesita de la creación de objetos que son parametrizados como recién salidos de fábrica, es decir, listos para utilizarse, pero con una gran ventaja y es que se busca mejorar el desempeño (performance). Es muy usado cuando se tienen sistemas que poseen datos repetitivos refiriéndose a atributos.

Este patrón es de mucha utilidad si necesitamos crear y manipular copias de otros objetos; en el momento en que procedemos a clonar un objeto, es importante que se tenga en cuenta si se guarda la referencia de otro o no.

Existen dos formas en las que se pueden replicar o copiar estos objetos.

Copia superficial

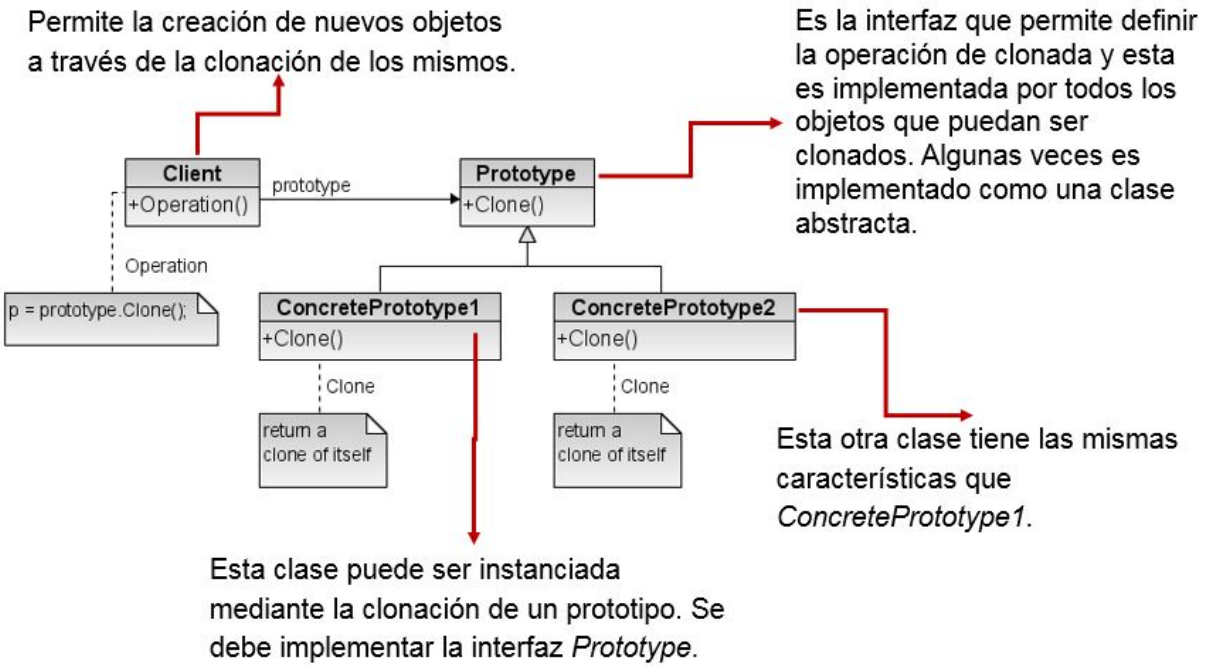
El objeto que se clonó tiene los mismos valores que el objeto original, guardando también una referencia con otros objetos que este contenga; es decir que, si estos son modificados desde el objeto original o desde algún objeto de sus clones, el cambio afectará a todos ellos.

Copia profunda

El objeto que se clonó tiene los mismos valores que el objeto original, al igual que las copias que contenga el objeto original.

NOTA: Se debe definir una interfaz que expone el método necesario para poder realizar la clonación del objeto. Las clases que pueden ser clonadas implementarán esta interfaz; ahora, las clases que se deseen clonar deberán utilizar el método definido en la interfaz.

El siguiente diagrama UML nos muestra la estructura del patrón prototipo (Programación.net).



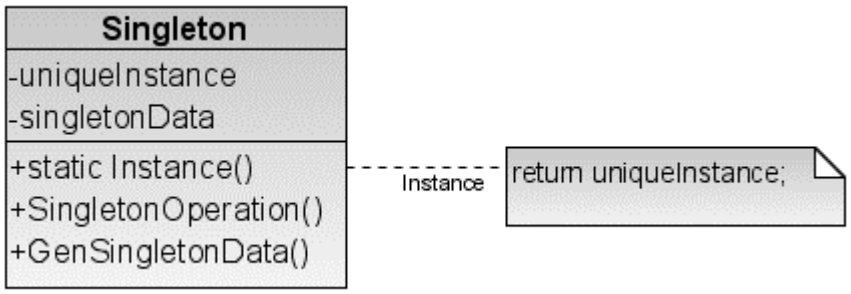
Las consecuencias son (Programación.net, s.f.):

Positivas
Clonar un objeto es mucho más rápido que crearlo.
Un programa puede añadir y borrar dinámicamente objetos prototipo en tiempo de ejecución.
El cliente no debe conocer los detalles de cómo construir los objetos prototipo.

Negativas
En objetos muy complejos, implementar la interfaz Prototype puede ser muy complicada.

Instancia única (Singleton)

Es un patrón de diseño o una solución a un problema, que permite acceder siempre a una única instancia del objeto; mediante esta implementación, podemos asegurar que al llamar una clase en el Singleton que se instanció, estamos invocando a la única instancia que hemos implementado. Este patrón de diseño puede ser implementado en múltiples lenguajes de programación.



Es muy efectivo para limitar la cantidad de instancias máximas en exactamente una; ahora, si más de un objeto requiere implementar o instanciar una clase Singleton, estos objetos compartirían la misma clase Singleton. La clase que implementa el patrón de diseño de instancia única, es conocida como una clase Singleton. Por medio de este patrón podemos hacer que únicamente exista una sola instancia de una clase en nuestro aplicativo, de forma similar que cuando tenemos variables globales, pero mucho más elegante.

Este patrón es muy utilizado, porque existen muchas ocasiones en las que se hace necesario compartir cierta información en la aplicación, teniendo un único punto de acceso a los recursos o a cualquier otra situación en la que se requiere tener un solo objeto de clase.

Lo primero que se realiza en el código es la creación de un objeto de la propia clase; esta será la única instancia de la clase. Luego se declara el constructor, este impedirá que se puedan crear nuevas instancias, y este constructor debe crearse privado para evitar que se pueden crear objetos desde fuera de nuestra clase. Por último, se declara un método público que obtiene el objeto que se ha credo en el primer paso y así formar un acceso a la única instancia de nuestra clase.

Para conocer cómo podemos realizar una implementación, haciendo uso del lenguaje Java

Ejemplo:

Ejemplo SingletonVamos a hacer una clase que simule un reloj, es decir, que nos vaya dando la hora cada segundo (para hacerlo sencillo simplemente haremos una aplicación de consola en la que imprimiremos la hora por pantalla cada segundo),y el reloj se va a crear usando el Singleton, porque no tiene sentido tener varias instancias distintas debido a que todas nos darían la misma hora y,además, al hacerlo como una aplicación de consola, si creamos más de un objeto de la clase reloj, se nos imprimiría cada hora,tantas veces como instancias de la clase reloj hubiéramos creado previamente.

```
packagecom.poi.singleton;

importjava.util.Date;
publicclassRelojextendsThread{privatestaticRelojreloj;


/***ConstructorprivadoporqueseusaelpatrónSingleton*/privateReloj(){}///**//*Construc
        torpublicoparaprobarsinsingleton//*///publicReloj(){//this.start();

//}/***Inicializaunasolavezelreloj*/privatesynchronizedstaticvoidcreateInstance(){if(reloj
        ==null){
```

```
reloj=newReloj();

reloj.start();
}}/***Obtienelaunicainstanciadelgestordegeozonas**@returngestorGeozonas*/publicst
        aticReloj getInstancia(){createInstance();
returnreloj;
}/***Imprimeporpantallalahoracadasegundo*/@Overridepublicvoidrun(){while(true){Da
        te hora =newDate(System.currentTimeMillis());
System.out.println(hora);
try{sleep(1000);}catch(InterruptedExceptione){e.printStackTrace(;}}}
packagecom.poi.singleton;publicclassMain{publicstaticvoidmain(String[]args){//3relojes
        usandoelpatrónsingleton
Reloj r =Reloj.getInstancia();
Reloj r2 =Reloj.getInstancia();
Reloj r3 =Reloj.getInstancia();
////3relojessinusarelpatrónsingleton//Relojr4=newReloj();
//Relojr5=newReloj();
//Relojr6=newReloj();}}
```

BibliografíaSalas, I. (2014, abril 10). Patrón Singleton en Java.En: Programando o intentándolo. Recuperado de: https://programandoointentandolo.com/2014/04/patron-singleton-java.html

Patrones estructurales – Adaptador o envoltorio (Adapter o wraper)

Patrones estructurales

Estos patrones describen la forma común en la que diferentes tipos de objetos pueden organizarse para trabajar unos con otros.

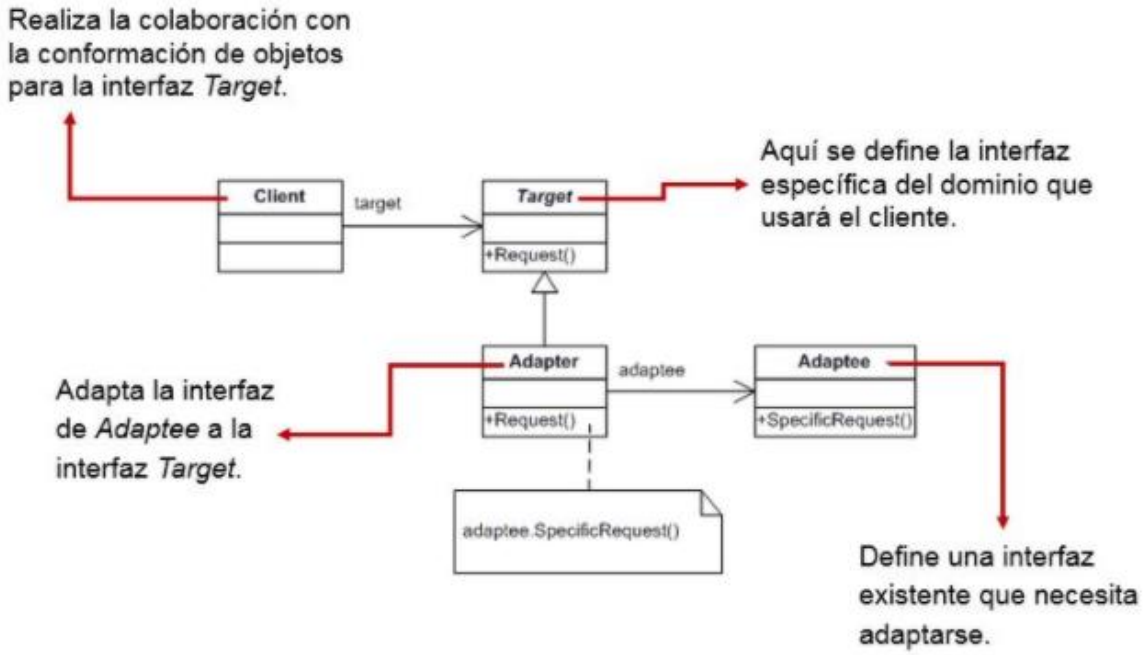
Adaptador o envoltorio (Adapter o wraper)

Con este patrón se busca, de manera estandarizada, adaptar un objeto a otro, transformar una interfaz en otra, de tal manera que una clase que no pueda hacer uso de la primera,

haga uso de ella por medio de la segunda. A esto se le conoce como Wrapper; un objeto Adapter, proporciona la funcionalidad que contiene una interfaz sin tener que conocer qué clase es utilizada para implementar esta interfaz, permitiendo trabajar juntas a dos clases con interfaces incompatibles.

Este patrón permite que exista cooperación entre clases para que puedan extender sus funcionalidades a clases de diferentes tipos, que no pueden ser usadas por mecanismos comunes de herencia. Es así como se dice que este patrón nos sirve para hacer dos interfaces diferentes que permiten una comunicación; se requiere de la adición de un adaptador intermedio que es el encargado de realizar la conversión de una interfaz a otra. Es muy utilizado cuando se tiene una interfaz pero no podemos usarla.

La representación UML del patrón Adapter es (Mi granito de Java, 2011):

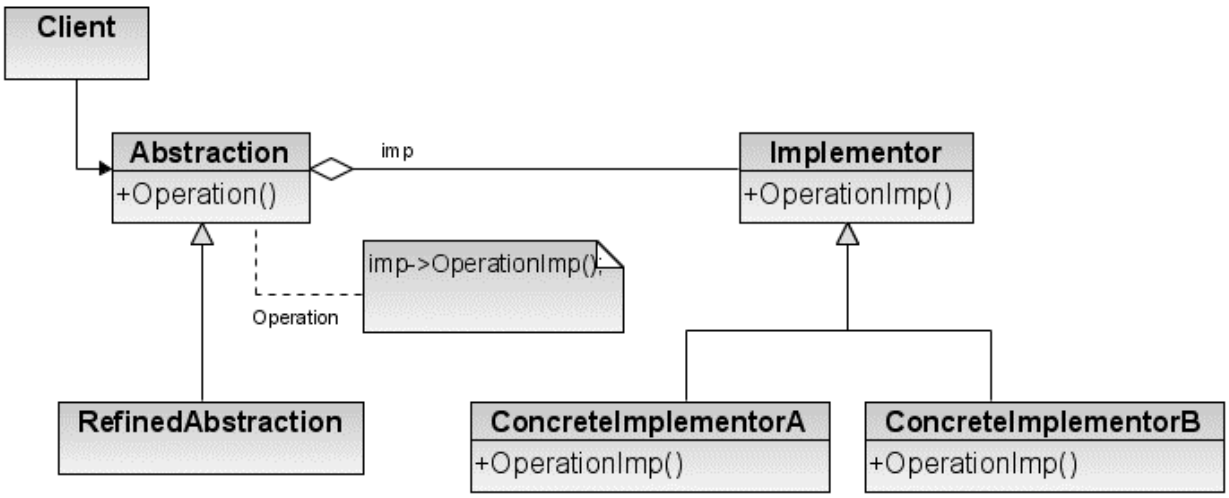


El Cliente llama a las operaciones sobre una instancia Adapter. De hecho, el adaptador llama a las operaciones de Adaptee que llevan a cabo el pedido.

Puente (Bridge)

El patrón Bridge desacopla una abstracción de la implementación, de tal manera que ambas puedan trabajar de forma independiente. Una abstracción hace referencia al comportamiento que una clase debe llevar a cabo; con la implementación se dice que se le coloca lógica a dicha obligación.

Este patrón permite modificar las implementaciones de una abstracción en tiempo de ejecución; es una técnica que se usa en programación para poder desacoplar la interfaz de una clase de su implementación, de tal manera que ambas puedan ser modificadas independientemente sin que se altere la otra. Cuando un objeto tiene unas implementaciones posibles, esto se activa mediante el uso de la herencia; en algunos casos la herencia puede ser inmanejable y acopla el código del cliente con una implementación concreta. Con este patrón se busca que estos inconvenientes no se presenten.

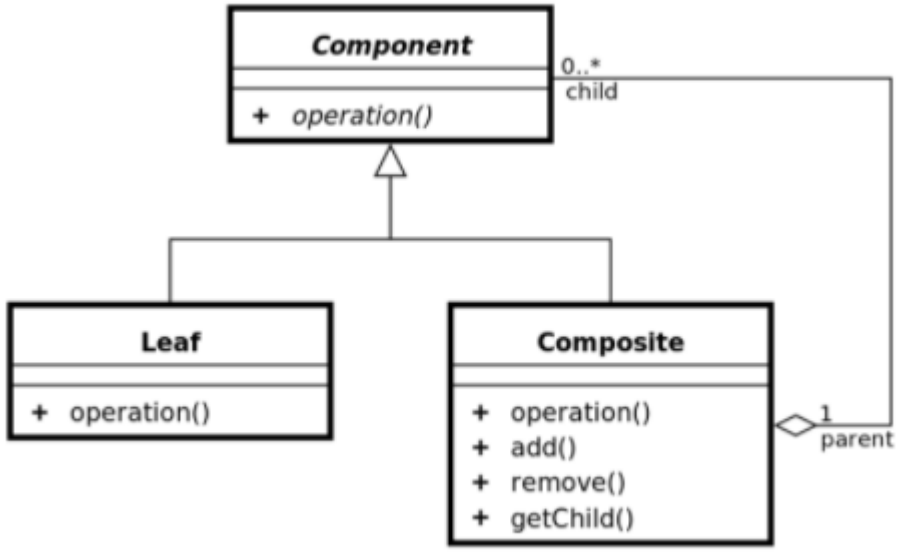


- Cuando se desea evitar un enlace permanente entre la abstracción y su implementación.
- Las abstracciones y sus implementaciones son extensibles por medio de subclases. Para este caso el patrón *Bridge* permite combinación abstracta e implementaciones diferentes y las extiende independientemente.
- Cuando se presentan cambios en la implementación de una abstracción no debe impactarse en los clientes; su código no debe tener que ser recompilado.

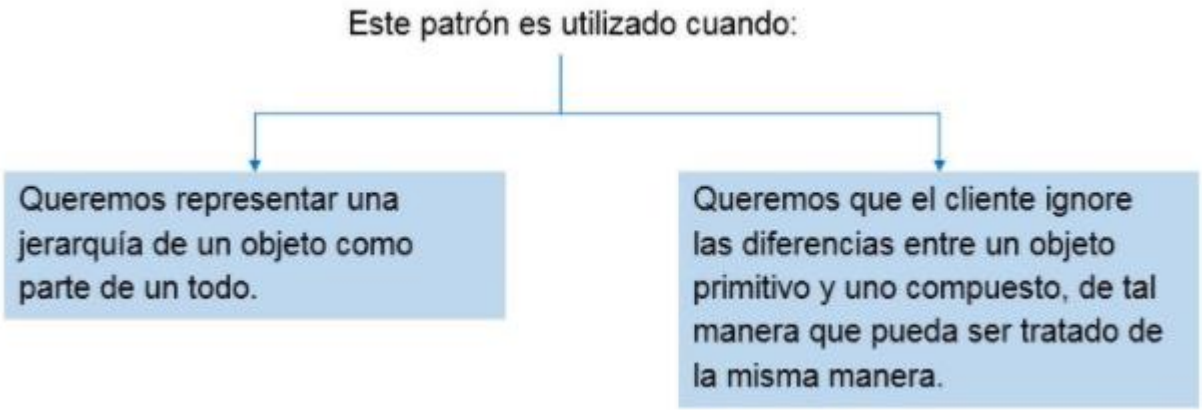
- Cuando se requiere compartir una implementación entre múltiples y este hecho debe ser escondido a los clientes.
- Cuando se necesita simplificar jerarquías demasiado pobladas.

Objeto compuesto (Composite)

El patrón Bridge desacopla una abstracción de la implementación, de tal manera que ambas puedan trabajar de forma independiente. Una abstracción hace referencia al comportamiento que una clase debe llevar a cabo; con la implementación se dice que se le coloca lógica a dicha obligación.



Con este patrón se busca la representación jerárquica de los objetos, que es conocida como “parte-todo”, donde se hace uso de la teoría que dice que las partes forman el todo, teniendo en cuenta que cada parte puede contener otras partes dentro de ellas. Podemos comparar este patrón con un panal y sus celdas; cada objeto simple (es decir la celda), puede tener relación con otros que son formados por una estructura mucho más compleja (panal). También es usado cuando se quiere tratar todos los componentes de una estructura en forma de árbol; esto mediante la interfaz o superclase, estableciendo reglas de comportamiento que permiten tratar a todos los objetos de la misma forma.



Interfaz o superclase de comportamiento

Aquí se establecen las pautas de comportamiento que deben cumplir los componentes y los objetos.

Hojas

Son las clases que se implementan en la interfaz que define el comportamiento.

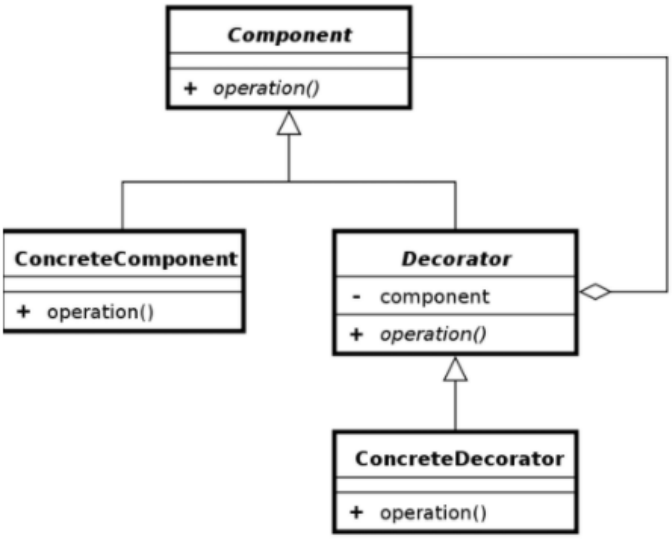
Composite

Es la clase en la que se define el objeto conformado por hojas.

Cliente

Clase que implementa el patrón.

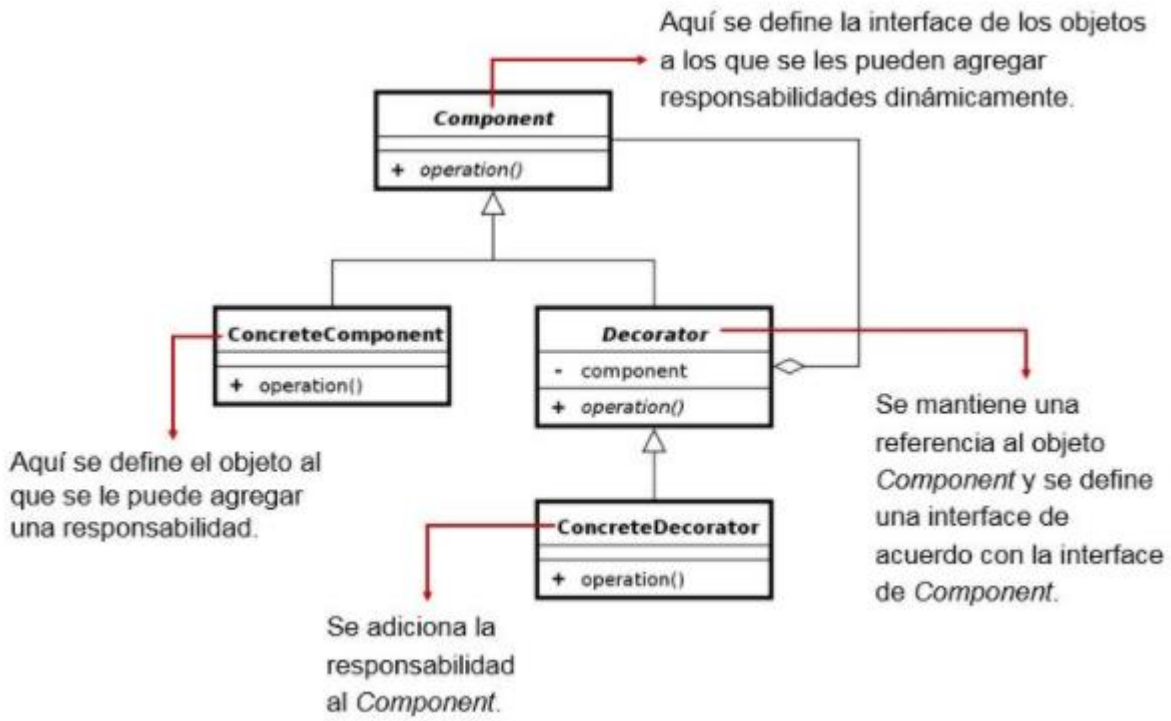
Decorador (Decorator)



Este patrón permite agregar responsabilidades a objetos concretos de manera dinámica. Con los decoradores se tiene una alternativa mucho más flexible que la herencia para poder extender las funcionalidades.

Algunas veces se requiere adicionar responsabilidades solo a un objeto, y no a la clase completa; estas responsabilidades pueden ser adicionadas a través de mecanismos de herencia y muchas veces este mecanismo no es fácil de usar debido la responsabilidad de adicionar esto de manera estática; la solución se encuentra en rodear el objeto con otro objeto, que es el que permite adicionar estas responsabilidades; este nuevo objeto es el Decorator.

La representación UML de este patrón es (Mi granito de Java, 2011):



El patrón decora un objeto y agrega funcionalidades a este; es muy utilizado para adicionar opciones de adornos en las interfaces al usuario. Este patrón debe utilizarse cuando en la herencia de clases no es posible utilizar la agregación de funcionalidades.

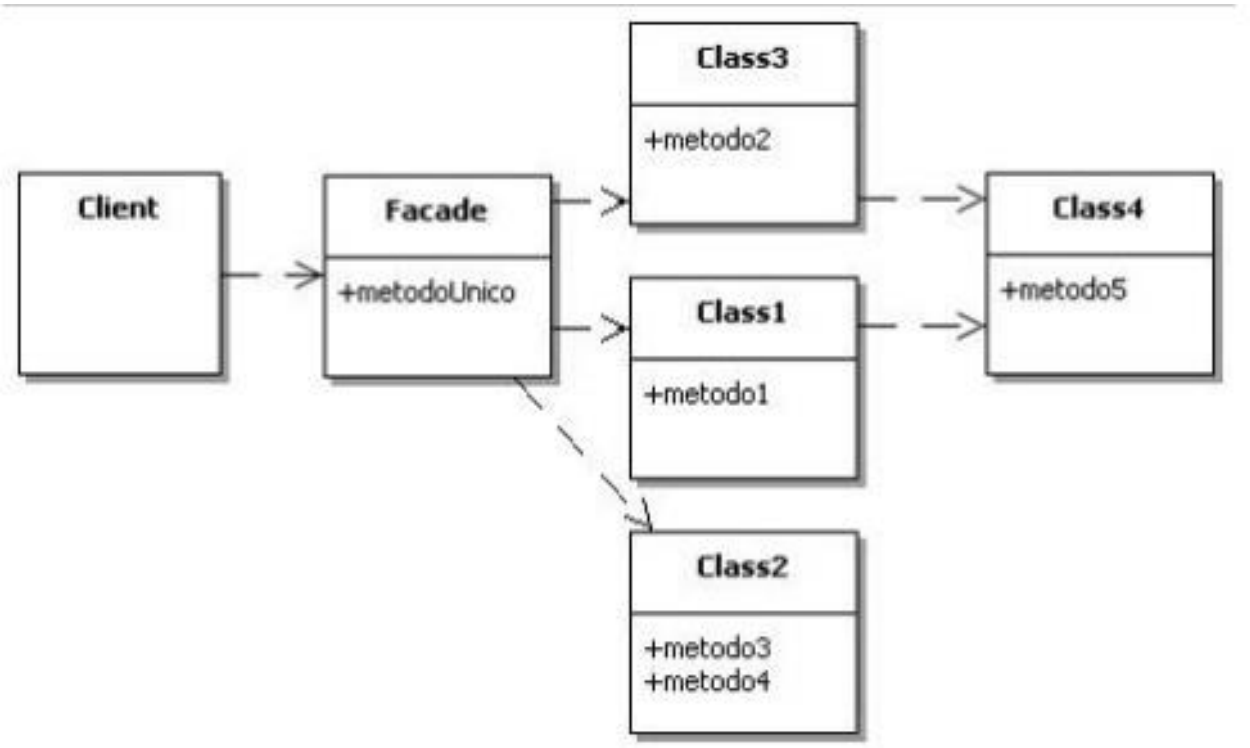
Para entender este contexto, supongamos que necesitamos comprar un computador de escritorio y este ya tiene un precio estipulado; pero si a este computador le queremos adicionar componentes, como cambiar la tarjeta graficadora entre otros, aquí tenemos

una adición al precio y si seguimos agregando accesorios, entonces nuestro precio incrementará; en este caso podemos hacer uso de Decorator.

Fachada (Facade)

Con este patrón se busca simplificar el sistema desde el punto de vista del cliente, por medio de una interfaz única para un conjunto de subsistemas, definiendo una interfaz de alto nivel haciendo que el sistema sea mucho más fácil de usar.

NOTA: Se busca reducir la comunicación y dependencia entre sistemas a un mínimo nivel; es así como se usa una fachada, simplificando la complejidad del cliente. El cliente accede al subsistema por medio de *Facade*, estructurando un entorno de programación mucho más sencillo, desde el punto de vista del cliente. Se utiliza cuando se requiere proporcionar una interfaz sencilla para un subsistema complejo, desacoplar un sistema de un cliente y de otros subsistemas, haciéndolos más independientes y portables o se requiere dividir los sistemas en niveles; aquí la fachada es el punto de entrada a cada uno de los niveles.



Los componentes de este patrón son (Mi granito de Java, 2011):

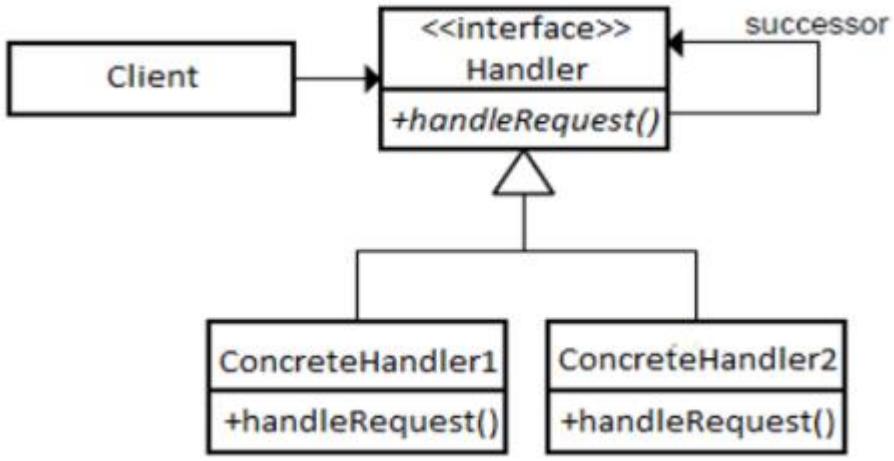
Facade

Aquí se conocen cuáles clases del subsistema serán responsables de una petición. Asigna las peticiones de los clientes en los objetos del subsistema.

Subsistema

Hace el manejo del trabajo asignado por el objeto Facade. No tienen ningún conocimiento del Facade (no guardan referencia de este).

Cadena de responsabilidad (Chain of responsibility)

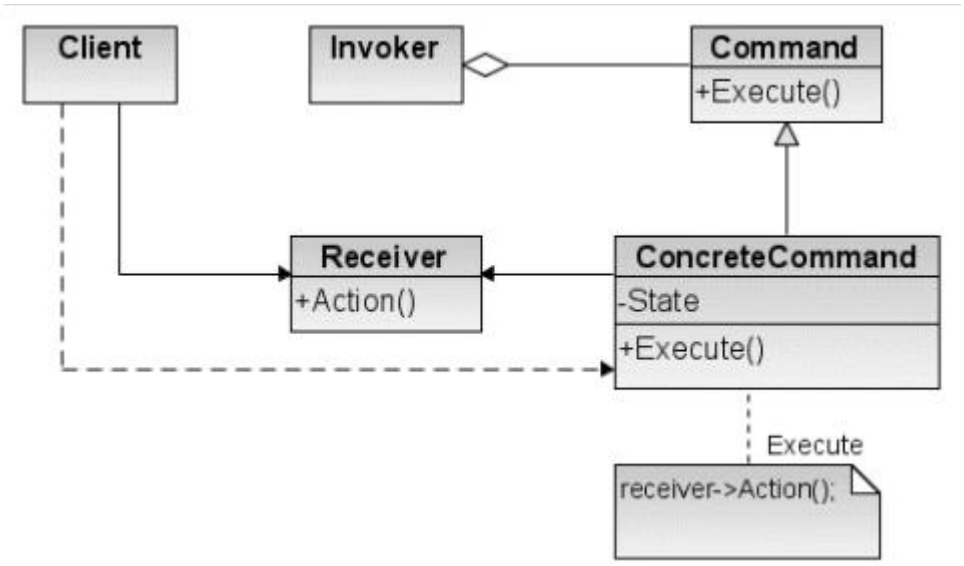


Este patrón permite que se establezca una cadena de objetos receptores por medio de los cuales se puede pasar una petición formulada por un objeto emisor. Se busca que cualquiera de los receptores pueda responder a la petición en función de un criterio ya establecido. Encadena los objetos receptores, pasa la petición a través de la cadena hasta que es procesada por algún objeto. Buscando evitar un montón de if – else largos y complejos en el código, pero ante todo se busca evitar que el cliente necesite conocer toda la estructura jerárquica y el rol que cumple cada integrante de la estructura (Mi granito de Java 2011).

¿Cuándo se utiliza este patrón?	¿Cuál es su finalidad?
Cuando las peticiones emitidas por el objeto deben ser atendidas por diferentes objetos receptores, cuando no se sabe a priori cuál es el objeto que puede dar solución al problema, cuando un pedido es manejado por varios objetos o el conjunto de objetos que pueden ser tratados por una petición deben ser especificados dinámicamente (Mi granito de Java 2011).	La finalidad de este patrón se basa en que se puedan crear sistemas que sirvan a diversas solicitudes de manera jerárquica, es decir, si un objeto es parte de un sistema y este no sabe cómo responder a una solicitud, este pasa la solicitud a todo su árbol de objetos; ahora, cada objeto puede tomar la responsabilidad y atender la solicitud (Mi granito de Java 2011).

Un ejemplo claro es cuando se envía una impresión, el cliente no tiene idea qué impresoras se encuentran instaladas, y él simplemente envía el trabajo, en cadena de objetos que son representados por las impresoras; cada uno lo deja pasar hasta que alguna impresora, finalmente, ejecuta dicha actividad. Existe un desacoplamiento evidente entre el objeto que lanza el trabajo (el cliente) y el que lo realiza (impresora) (Mi granito de Java 2011).

Orden (Command)



Este patrón permite establecer escenarios en los que se necesite encapsular una petición dentro de un objeto, por medio de una parametrización de los clientes con distintas peticiones, encolarlas, guardarlas en un registro de sucesos, o simplemente implementar un mecanismo que permita deshacer/repetir. Se puede solicitar una operación a un objeto sin que tengamos conocimiento del contenido, ni el receptor real de la misma.

- Cuando se requiere encolar o registrar mensajes.
- Cuando se tiene la posibilidad de deshacer las operaciones realizadas.
- Cuando se necesite uniformidad en la invocación de acciones. Cuando se requiere facilitar la parametrización de las acciones que se desean realizar.
- Cuando se necesita independizar el momento de petición de la ejecución.

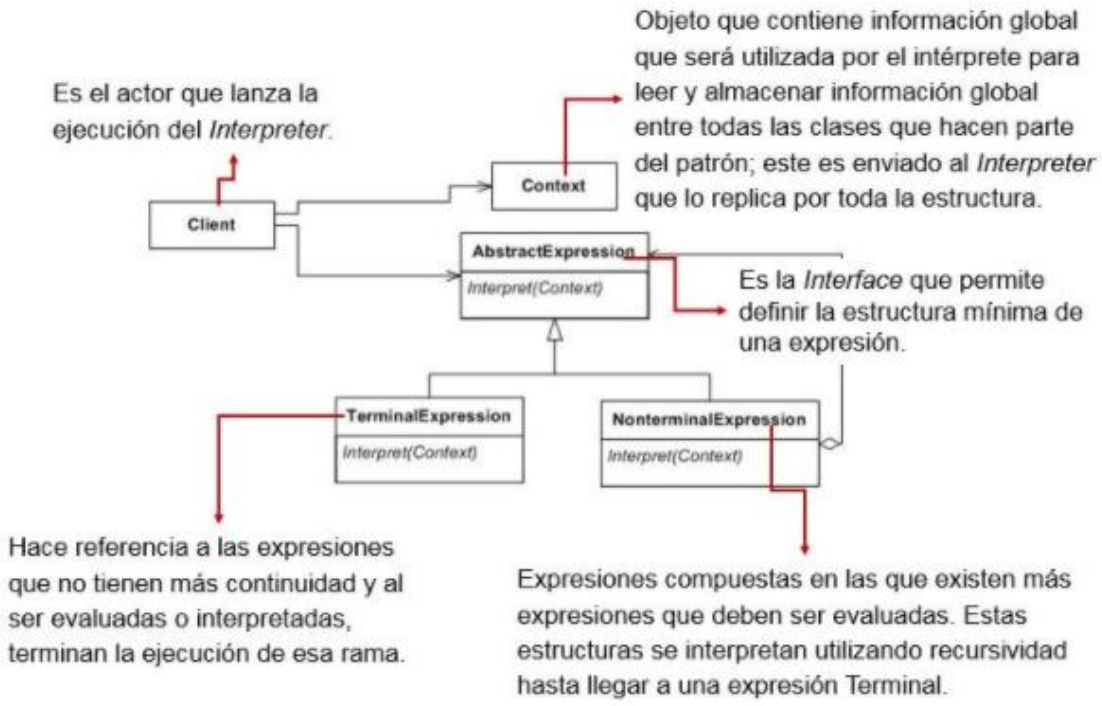
El parámetro de una orden puede ser otra orden a ejecutar, buscando el desarrollo de sistemas por medio de órdenes de alto nivel, que se crean con operaciones sencillas; esta sencillez es necesaria cuando se requiere extender el sistema a otras nuevas acciones. Con este patrón Command se busca desacoplar al objeto que invoca a una operación de otro que tiene el conocimiento necesario para poder realizarla, otorgando mucha flexibilidad; por ejemplo, cuando se ejecuta una aplicación, el elemento de menú como un botón realiza una determinada acción. Estos objetos de Command pueden ser cambiados dinámicamente.

Intérprete (Interpreter)

El patrón Interpreter permite la representación de un lenguaje por medio de reglas gramaticales. Para poder realizar esto, se definen reglas gramáticas y se especifica cómo deben ser interpretadas. Se utiliza una clase que permite representar una regla gramatical.

Si se presenta un tipo de problema particular, se puede aprovechar expresando los diferentes casos de este problema por medio de sentencias de un lenguaje simple. Es así como se puede construir un intérprete que permita dar solución a este problema, interpretando dichas sentencias.

La representación UML de este patrón es (reactiveprogramming.io):

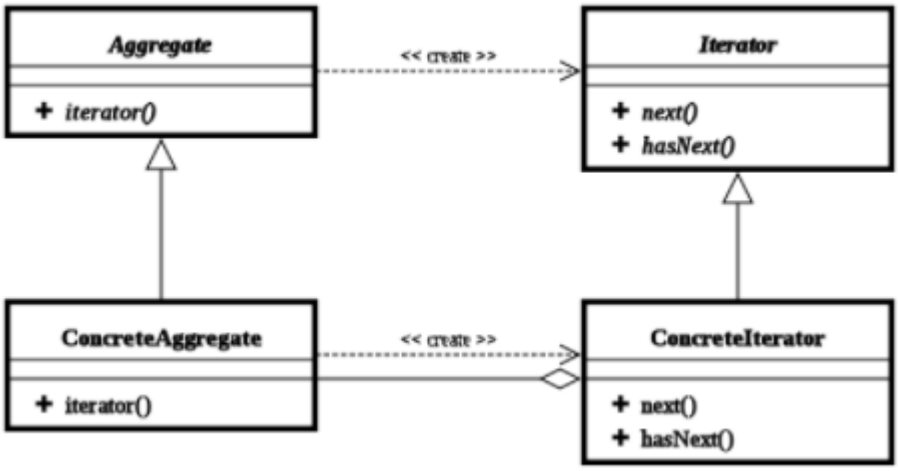


Sabías que…

Este patrón es muy utilizado cuando se tiene un lenguaje que hace de intérprete y sus palabras son interpretadas como árboles sintácticos abstractos, y esta gramática debe ser simple.

Este patrón es muy poco utilizado por los desarrolladores; esto no quiere decir que no se pueda utilizar. La situación ideal en la que se puede decir que es de mucha utilidad este patrón, es que exista un lenguaje sencillo que requiera interpretarse con palabras. El ejemplo más claro y fácil de interpretar es el lenguaje Java; con este lenguaje podemos escribir archivos .java que pueden ser interpretados por el ser humano, y luego este archivo es compilado e interpretado para que pueda ser ejecutado en forma de sentencia entendible por una máquina.

Iterador (Iterator)

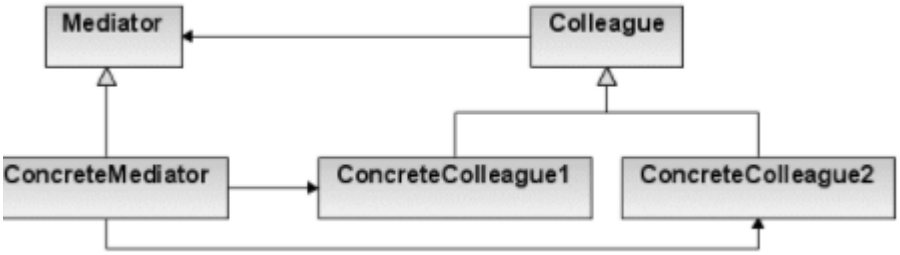


Este patrón suministra un mecanismo estándar que permite acceder secuencialmente a los elementos de una colección; define una interface que declara métodos que permitan acceder de manera secuencial a todos los objetos pertenecientes a una colección. Una clase accede a una colección por medio de la interface.

Este patrón fue creado porque existe una gran diversidad de colecciones y algoritmos que permiten recorrer una colección, buscando acceder al contendido de cada objeto que está incluido, sin tener que exponer la estructura. Este patrón nace con la finalidad de soportar diversas maneras de recorrer los objetos, ofreciendo una interfaz uniforme que permite recorrer diferentes tipos de estructuras de agregación.

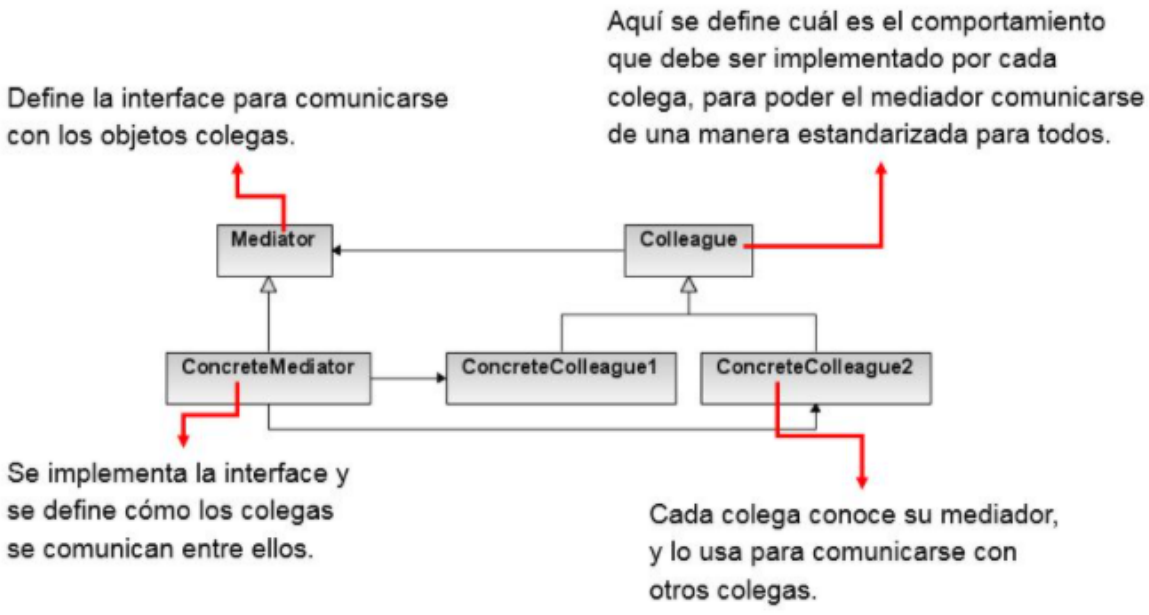
Es muy utilizado cuando se requiere de manera estándar recorrer una colección, es decir, cuando no se necesita que el cliente conozca la estructura interna que conforma la clase; el cliente no siempre requiere conocer si debe o no recorrer un List o un Set o un Queue y, mucho menos, que clase específica esté recorriendo.

Mediador (Mediator)



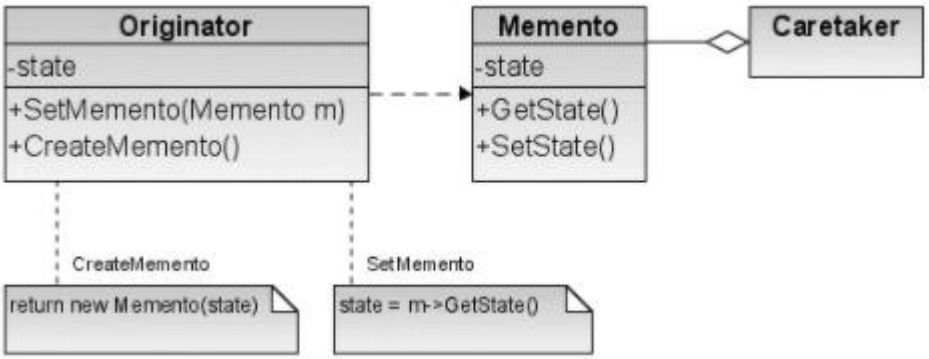
En este patrón se define el objeto que hace la tarea de procesador central, coordinador de relaciones entre los asociados y los participantes. Permite que varios objetos interactúen sin que se genere acoplamiento fuerte entre estas relaciones, y los objetos se comunican con un mediador y este realiza la comunicación con el resto. Cuando varios objetos interactúan con otros objetos, puede que se forme una estructura que se puede convertir en compleja, debido a las muchas conexiones que existen entre los distintos objetos. Puede existir un caso extremo en que cada objeto puede conocer a los demás objetos; para poder controlar y evitar que esto suceda, se hace uso del patrón Mediator, encapsulando el comportamiento de todo el conjunto de objetos en un único objeto.

La representación UML de este patrón es (Mi granito de Java, 2011):



Ten en cuenta: Este patrón es muy usado cuando se tiene un conjunto inmenso de objetos que se comunican de manera bien definida, pero a la vez compleja. Cuando se necesita reutilizar un objeto y se vuelve complejo y difícil porque se relaciona con muchos objetos; si tenemos clases que son difíciles de realizar debido a que su función básica está ligada con relaciones independientes.

Recuerdo (Memento)



Con este patrón se puede capturar y exportar el estado interno que tiene un objeto, para luego ser restaurado sin romper la encapsulación (Mi granito de Java, 2011).

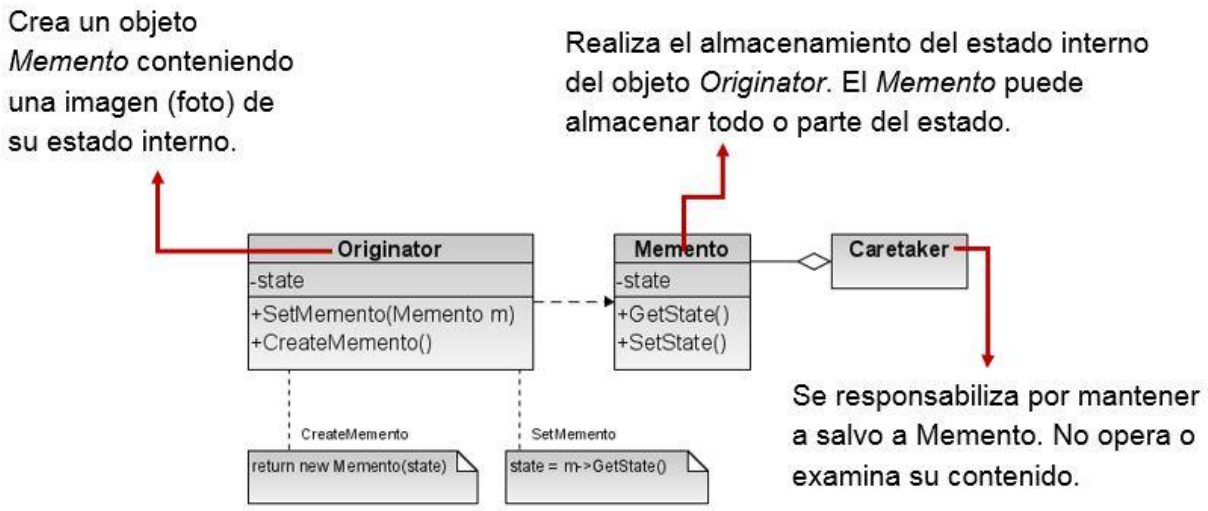
La finalidad principal es almacenar el estado del objeto (o de un sistema completo) en un determinado momento, de tal manera que se pueda restaurar posteriormente si se llegase a necesitar. Para poder lograrlo es necesario almacenar el estado del objeto en un instante de tiempo, en una clase independiente a la que pertenece el objeto (pero sin romper la encapsulación), de tal forma que ese recuerdo permita que el objeto se pueda modificar y se pueda devolver a su estado anterior (Mi granito de Java, 2011).

NOTA: Actualmente existen muchos aplicativos que nos permiten "deshacer" y "rehacer" de forma muy sencilla. Para ciertos aplicativos se convierte en una obligación el tener estas funciones y sería casi imposible que no las tuviese. Sin embargo, si queremos programar estos aplicativos, entonces ya nos resulta mucho más compleja la implementación de estas funciones. Con este patrón se puede dar solución a esta complejidad (Mi granito de Java, 2011).

Se usa cuando necesitamos restaurar el sistema en un estado pasado, entonces requerimos facilitar el hacer y deshacer de ciertas operaciones; es así como necesitamos guardar el estado anterior de los objetos que estamos trabajando o en su defecto recordar los cambios de manera incremental (Mi granito de Java, 2011).

En la actualidad, gran variedad de aplicaciones posee las opciones de "deshacer" y "rehacer" (Mi granito de Java, 2011).

La representación UML de este patrón es (Mi granito de Java, 2011):



Este patrón debe ser utilizado cuando se necesite salvar el estado de un objeto y tener disponible los distintos estados históricos que se necesiten. Por esta razón, este patrón es muy intuitivo para darse cuando debe ser utilizado (Mi granito de Java, 2011).

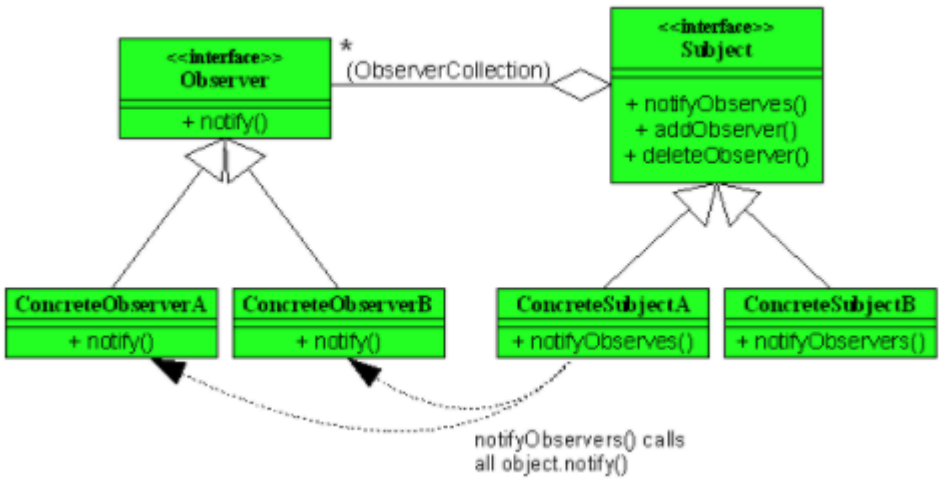
Ejemplo

classMemento{privateStringstate;
publicMemento(Stringstate) {this.state = state;}
publicStringgetState() {returnstate;}}
classOriginator{privateStringstate;/* lots of memory consumptive private data that is not necessary to define the* state and should thus not be saved. Hence the small memento object. */publicvoidsetState(Stringstate) {System.out.println("Originator: Setting state to "+ state);

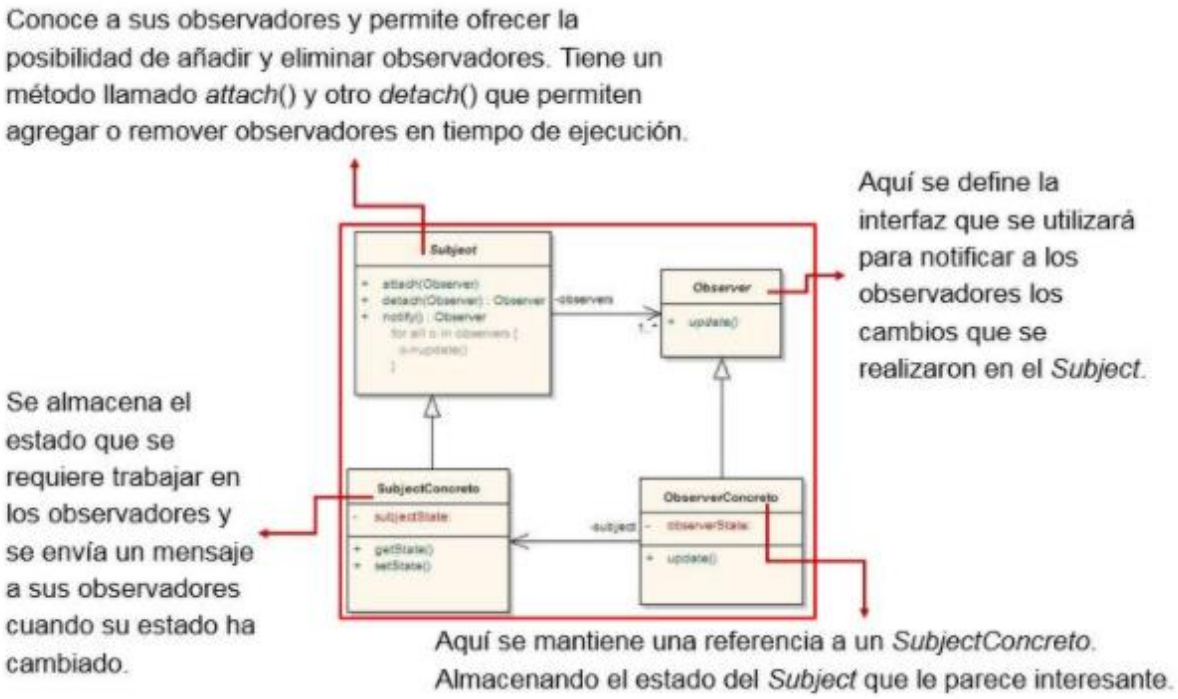
this.state = state;}
publicMemento save() {System.out.println("Originator: Saving to Memento.");
returnnewMemento(state);}publicvoidrestore(Memento m) {state = m.getState();
System.out.println("Originator: State after restoring from Memento: "+ state);}}
classCaretaker{privateArrayList<Memento> mementos = newArrayList<>();
publicvoidaddMemento(Memento m) {mementos.add(m);}
publicMemento getMemento() {returnmementos.get(1);}
}publicclassMementoDemo{publicstaticvoidmain(String[] args) {Caretaker caretaker = newCaretaker();Originator originator = newOriginator();
originator.setState("State1");
originator.setState("State2");
caretaker.addMemento(originator.save());originator.setState("State3");
caretaker.addMemento(originator.save());originator.setState("State4");
originator.restore(caretaker.getMemento());}}

Observador (Observer)

Con este patrón se reacciona a ciertas clases que son llamadas observadoras sobre un evento determinado. Se utiliza a la hora de programar para poder realizar un monitoreo sobre cuál es el estado de un objeto. Este patrón tiene relación con el principio de invocación implícita. La principal motivación de este patrón es que hace uso de un sistema que detecta los eventos en tiempo de ejecución, característica muy importante cuando se desarrollan aplicaciones en tiempo real (Mi granito de Java, 2011). .

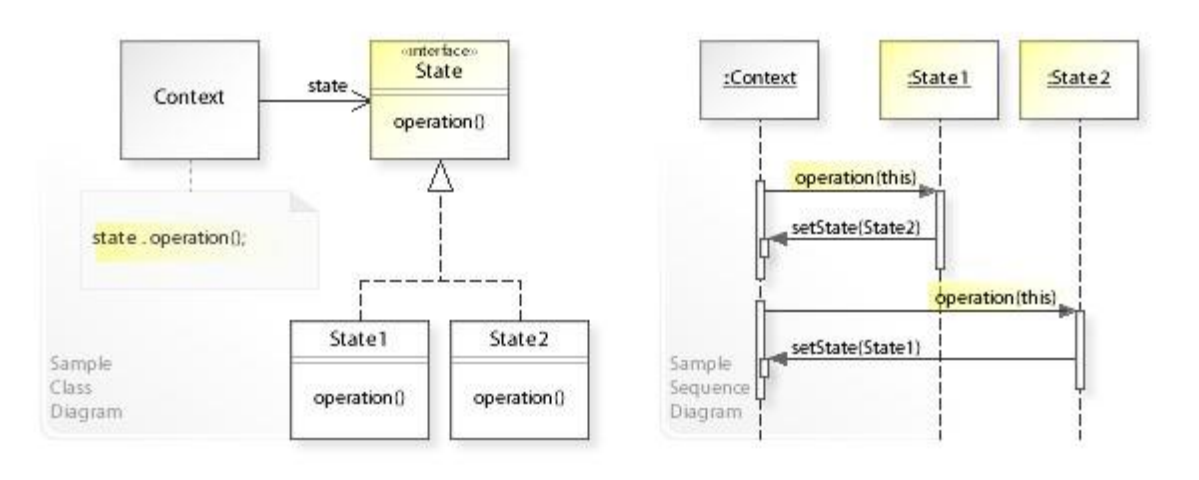


La representación UML de este patrón es (Mi granito de Java, 2011):



Este patrón tiene un uso muy concreto: si existen varios objetos que necesitan ser notificados de un evento y cada uno de estos objetos concluyen cómo deben reaccionar cuando estos eventos son producidos (Mi granito de Java, 2011).

Estado (State)

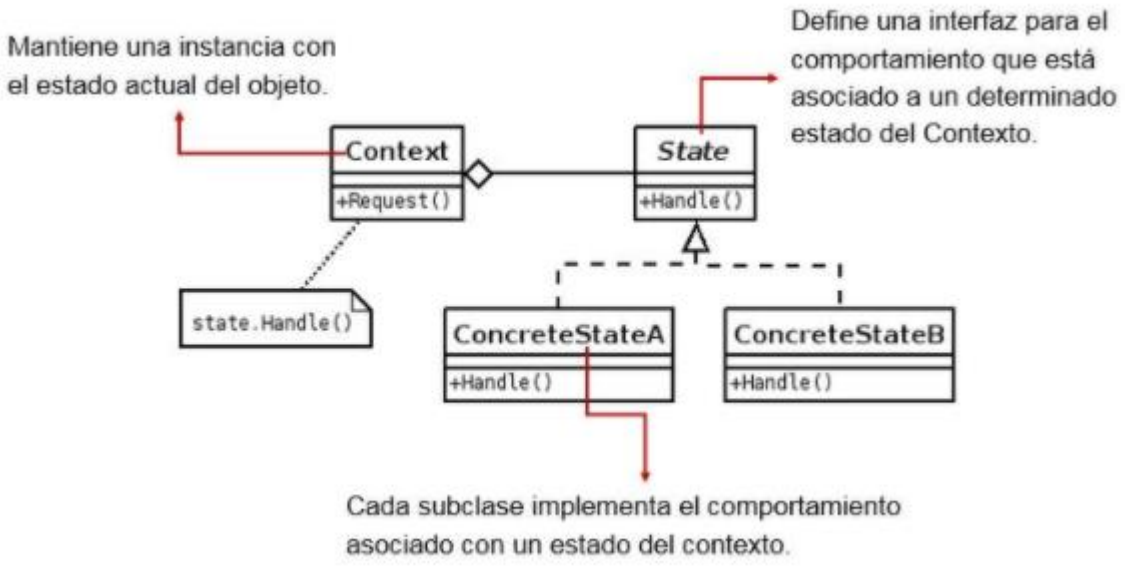


clic para ampliar la imagen

Este patrón permite que se modifique el comportamiento de un objeto cada vez que se cambia el estado interno del mismo. El objetivo es buscar que este objeto reaccione según el estado interno; muchas veces estos comportamientos se pueden solucionar haciendo uso de un *boolean* o de una constante; sin embargo, esto puede incluir el uso de muchas sentencias IF-ELSE, código poco entendible y una dificultad de mantener estos códigos. Con el patrón *State* se busca desacoplar el estado de la clase que se esté trabajando.

En algunas ocasiones se requiere que el objeto se comporte de diferentes formas, según el estado en que este se encuentre; esto es muy complicado de manejar, sobretodo cuando se tiene en cuenta el cambio del comportamiento y el estado que tiene dicho objeto, y todo dentro de un mismo bloque de código. Este patrón propone entonces crear un objeto por cada cambio que sea posible.

La representación UML de este patrón es (Mi granito de Java, 2011):



Este patrón debe ser utilizado cuando (Mi granito de Java, 2011):

- Se tiene un comportamiento del objeto dependiendo del estado del mismo, y este debe cambiar en tiempo de ejecución de acuerdo con el comportamiento del estado.

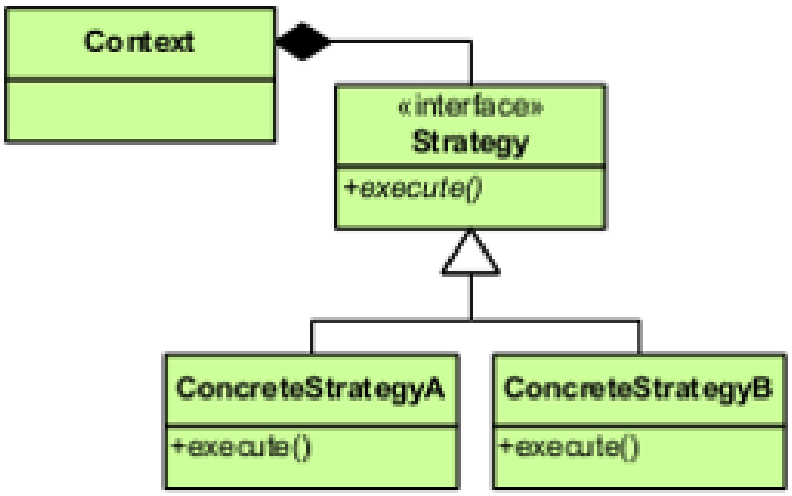
- Las operaciones están conformadas por largas sentencias con múltiples ramas que tienen dependencia en el estado del objeto.

- Se tiene un determinado objeto con diferentes estados y distintas responsabilidades de acuerdo con el estado en que se encuentre en determinado momento.

Ejemplo:

```
// Not good: unwieldy "case" statementclassCeilingFanPullChain{privateintcurrentState;
publicCeilingFanPullChain() {currentState = 0;}
publicvoidpull() {if(currentState == 0) {currentState = 1;System.out.println("low speed");}
 elseif(currentState == 1) {currentState = 2;
System.out.println("medium speed");}
elseif(currentState == 2) {currentState = 3;
System.out.println("high speed");}
else{currentState = 0;System.out.println("turning off");}}}
publicclassStateDemo{publicstaticvoidmain(String[] args) {CeilingFanPullChain chain =
newCeilingFanPullChain();
while(true) {System.out.print("Press ENTER");
getLine();
chain.pull();}}
staticStringgetLine() {BufferedReader in =
newBufferedReader(newInputStreamReader(System.in));
Stringline = null;try{line = in.readLine();}
 catch(IOException ex) {ex.printStackTrace();}returnline;}}
```

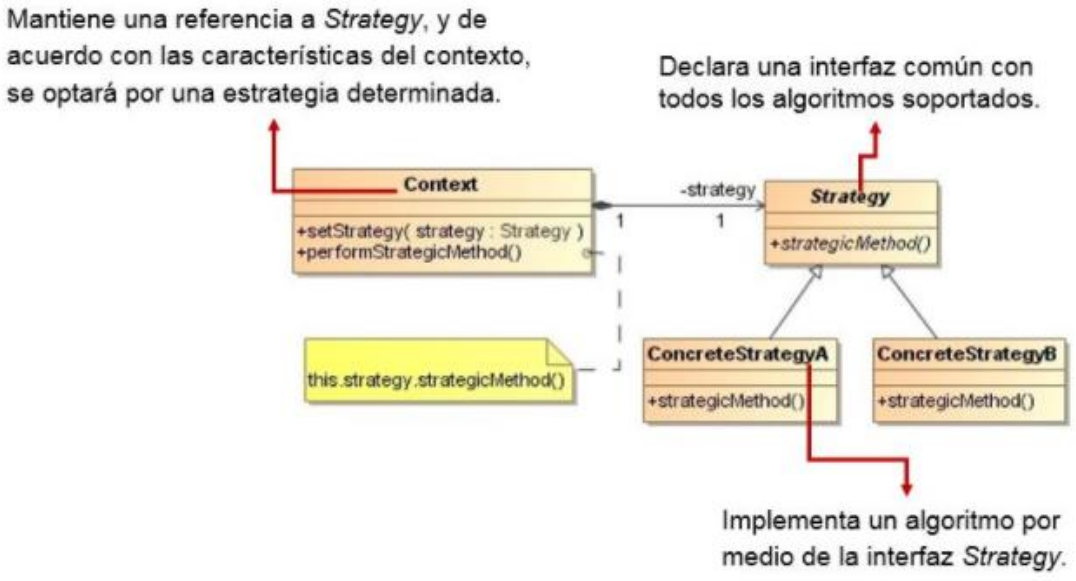
Estrategia (Strategy)



Este patrón permite encapsular algoritmo en una clase, permitiendo que se reutilice e intercambie, de acuerdo con un parámetro que puede ser cualquier objeto, permitiendo así que la aplicación decida en tiempo de ejecución, cual algoritmo debe ser ejecutado (Mi granito de Java, 2011).

La esencia de este patrón está en la encapsulación de algoritmos que relacionan las subclases de una superclase común, lo cual permite la selección de un algoritmo que cambia de acuerdo con el objeto, permitiendo la variación en el tiempo.

La representación UML de este patrón es (Mi granito de Java, 2011):



Context / Cliente: solicita un servicio a Strategy y este debe devolver el resultado de un StrategyConcreto.

Este patrón debe ser utilizado cuando (Mi granito de Java, 2011):

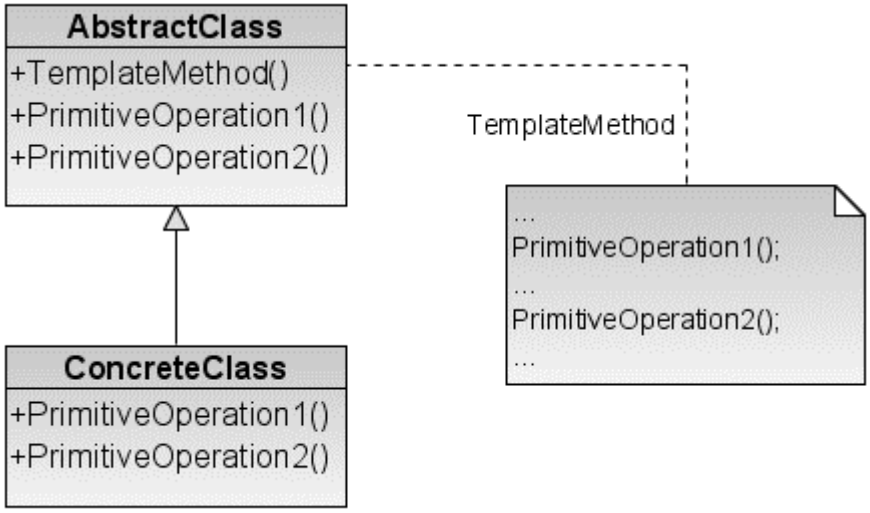
- Se requiere que un programa proporcione múltiples variantes de un algoritmo o comportamiento.
- Cuando es posible encapsular las variantes de comportamiento en clases separadas, proporcionando un modo consistente que permita acceder a los comportamientos.
- Permite cambiar o agregar algoritmos, independientemente de la clase que desee utilizar.

Ejemplo:

```
// 1. Define the interface of the algorithminterfaceStrategy{voidsolve();}
// 2. Bury implementation@SuppressWarnings("ALL")abstractclassStrategySolutionimplementsStrategy {
// 3. Template Methodpublicvoidsolve() {start();
while(nextTry() && !isSolution()) {}stop();}
abstractvoidstart();
abstractbooleannextTry();
abstractbooleanisSolution();
abstractvoidstop();}
classFOOextendsStrategySolution {privateintstate = 1;
protectedvoidstart() {System.out.print("Start  ");}
protectedvoidstop() {System.out.println("Stop");}
protectedbooleannextTry()
{System.out.print("NextTry-"+ state++ + "");
returntrue;}protectedbooleanisSolution() {System.out.print("IsSolution-"+ (state == 3) + "
");
return(state == 3);}}
// 2. Bury implementationabstractclassStrategySearchimplementsStrategy {
//3. Template Methodpublicvoidsolve() {while(true) {preProcess();
if(search()) {break;}
```

```
postProcess();}}
abstractvoidpreProcess();
abstractbooleansearch();
abstractvoidpostProcess();}
@SuppressWarnings("ALL")classBARextendsStrategySearch {privateintstate =
1;protectedvoidpreProcess() {System.out.print("PreProcess ");}
protectedvoidpostProcess() {System.out.print("PostProcess ");}
protectedbooleansearch() {System.out.print("Search-"+ state++ + " ");
returnstate == 3? true: false;}}
// 4. Clients couple strictly to the interfacepublicclassStrategyDemo{
// client code hereprivatestaticvoidexecute(Strategy strategy) {strategy.solve();}
publicstaticvoidmain( String[] args ) {Strategy[] algorithms = {newFOO(), newBAR()};
for(Strategy algorithm : algorithms) {execute(algorithm);}
```

Método plantilla (Template method)



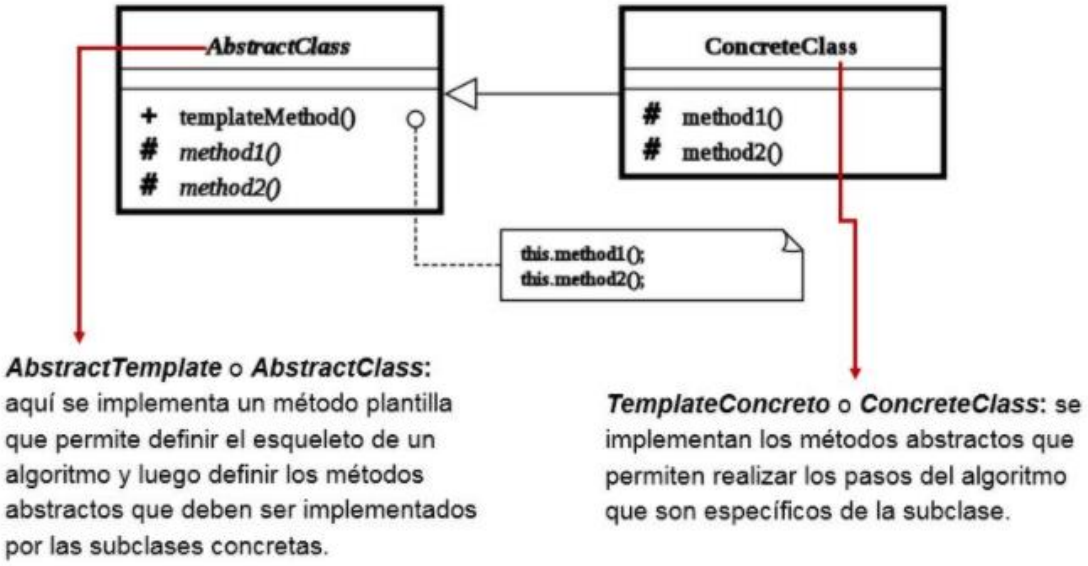
Este patrón define la estructura algorítmica que contiene la lógica que quedará a cargo de las subclases. Se debe escribir una clase abstracta que contiene parte de la lógica que se necesita para lograr el objetivo; aquí se define la estructura de herencia que luego se usará de plantilla en los métodos de las subclases, es decir, se define el esqueleto del algoritmo

delegando paso a paso las subclases, permitiendo que parte del algoritmo se pueda redefinir sin necesidad de cambiar la estructura (Mi granito de Java, 2011).

Los casos en donde este patrón debe ser utilizado son (Mi granito de Java, 2011):

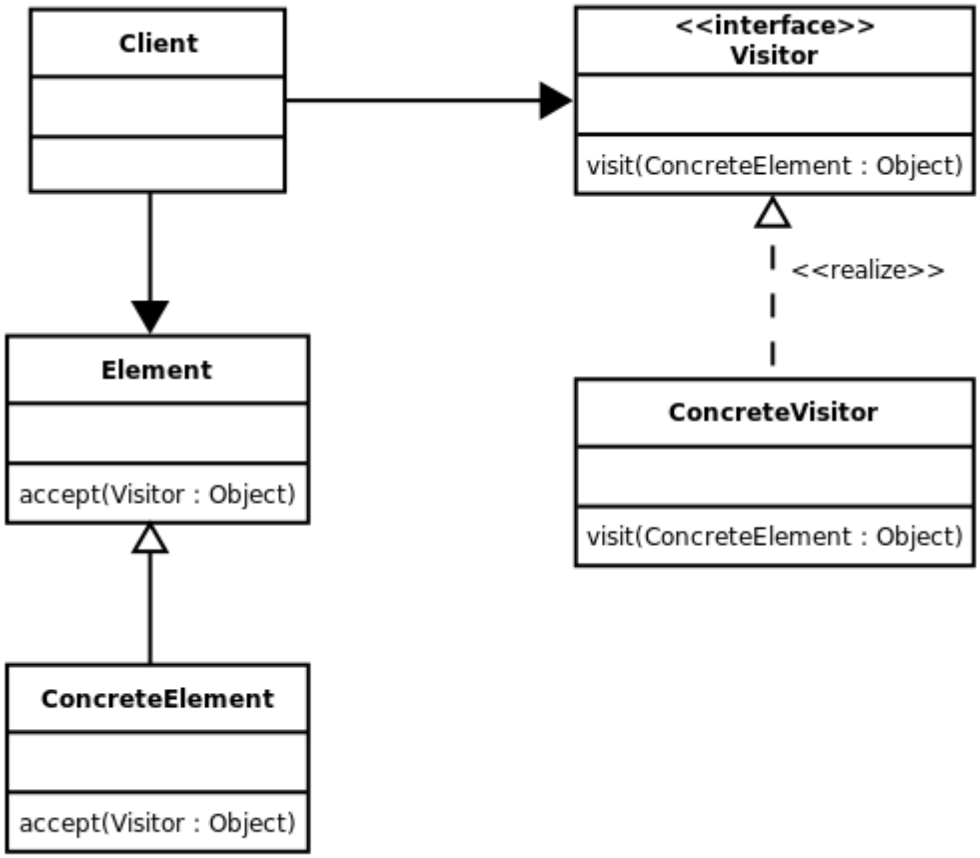
1. Cuando se necesita factorizar el comportamiento común de varias subclases.
2. Cuando se requiere implementar partes fijas de un algoritmo una única vez y dejar que las subclases implementen las partes variables.
3. Cuando se busca tener el control de las ampliaciones de las subclases, convirtiendo en método plantilla aquellos procedimientos que pueden ser redefinidos.

La representación UML de este patrón es (Mi granito de Java, 2011):



Este patrón es de especial utilidad, cuando se necesita realizar un algoritmo que sea común para muchas clases, pero con pequeñas variaciones entre unas y otras.

Visitante (Visitor)

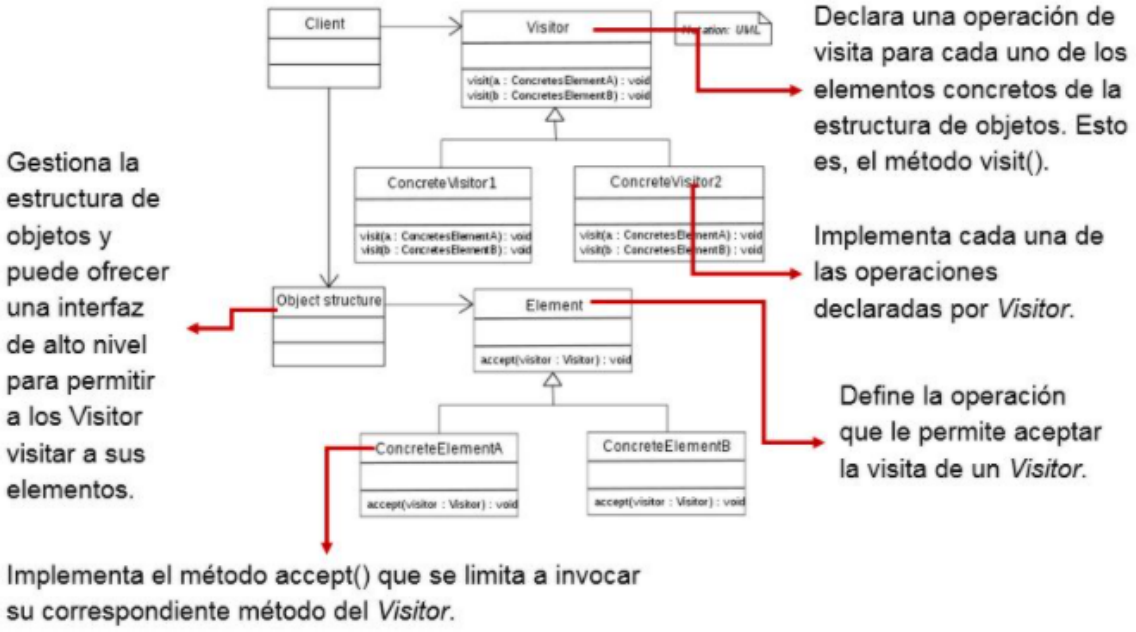


Con este patrón se busca separar un algoritmo de la estructura de un objeto. La operación es implementada de tal forma, que no sea modificable el código de las clases sobre las que se está operando

NOTA: Cuando un objeto es el responsable del mantenimiento de un tipo de información, entonces es lógico que se le asigne la responsabilidad de poder realizar todas las operaciones necesarias sobre esa información. La operación es definida en cada una de

las clases que representan los posibles tipos sobre los que son aplicados dicha operación, y a través del polimorfismo y la vinculación dinámica se elige en tiempo de ejecución qué versión de la operación se ha de ejecutar; de esta manera se evita un análisis de casos sobre el tipo del parámetro.

La representación UML de este patrón es (Mi granito de Java, 2011):



Este patrón debe ser utilizado cuando (Mi granito de Java, 2011):

Una estructura de objetos tiene muchas clases de objetos con diferentes interfaces y se deben llevar a cabo operaciones sobre estos objetos que son diferentes en cada clase concreta.

Se quieren realizar muchas operaciones dispares sobre objetos de una estructura de objetos, sin que se incluyan dichas dichas operaciones en las clases.

Las clases que definen la estructura de objetos no cambian, pero las operaciones que se llevan a cabo sobre ellas.

Ejemplo

```
interfaceElement{voidaccept(Visitor v);
}
classFOOimplementsElement {publicvoidaccept(Visitor v) {v.visit(this);
}
publicStringgetFOO() {return"FOO";
}
        }
classBARimplementsElement {publicvoidaccept( Visitor v ) {v.visit( this);}publicStringgetBAR() {return"BAR";
}
        }
classBAZimplementsElement {publicvoidaccept(Visitor v) {v.visit(this);
}
publicStringgetBAZ() {return"BAZ";
}
        }
interfaceVisitor{voidvisit(FOO foo);
voidvisit(BAR bar);
voidvisit(BAZ baz);
}
classUpVisitorimplementsVisitor {publicvoidvisit(FOO foo) {System.out.println("do Up on
"+ foo.getFOO());
}
publicvoidvisit(BAR bar) {System.out.println("do Up on "+ bar.getBAR());
}
publicvoidvisit(BAZ baz) {System.out.println( "do Up on "+ baz.getBAZ() );
}
        }
ClassDownVisitorimplementsVisitor {publicvoidvisit(FOO foo) {System.out.println("do
Down on "+ foo.getFOO());
}
publicvoidvisit(BAR bar) {System.out.println("do Down on "+ bar.getBAR());
}
publicvoidvisit(BAZ baz ) {System.out.println("do Down on "+ baz.getBAZ());
```

```
}
        }
publicclassVisitorDemo{publicstaticvoidmain( String[] args ) {Element[] list = {newFOO(),
newBAR(), newBAZ()};
UpVisitor up = newUpVisitor();
DownVisitor down = newDownVisitor();
for(Element element : list) {element.accept(up);
}
for(Element element : list) {element.accept(down);
}
        }
                }
```

IU Digital de Antioquia
INSTITUCIÓN UNIVERSITARIA DIGITAL DE ANTIOQUIA

ORD No 74 de 2017 · VIGILADA MinEducación

www.iudigital.edu.co