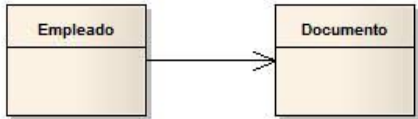
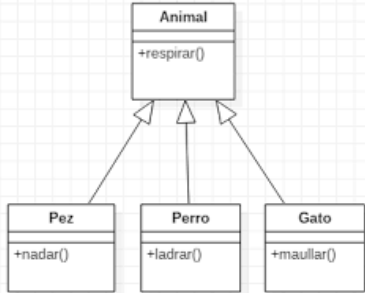
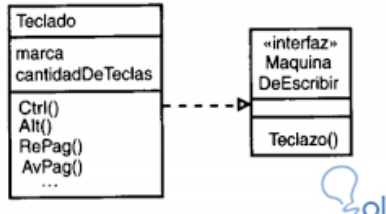
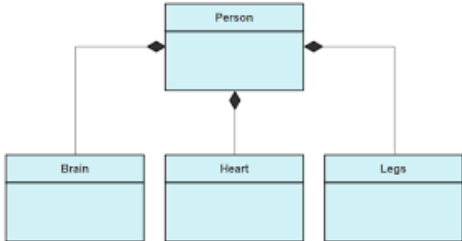
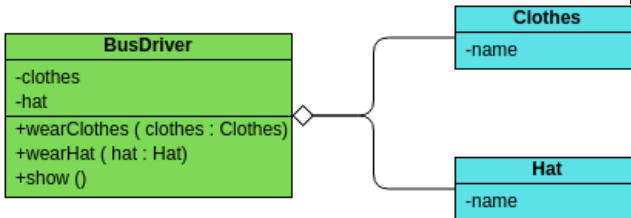



Relaciones

Relación	Descripción	Ejemplo
Asociación	Indica que una propiedad de una clase <u>contiene</u> una referencia a una instancia (o instancias) de otra clase. La asociación es la relación más utilizada entre una clase y otra clase, lo que significa que existe una conexión entre un tipo de objeto y otro tipo de objeto.	
Herencia	En la relación de herencia, la subclase <u>hereda</u> todas las funciones de la clase principal y la clase principal tiene todos los atributos, métodos y subclases. Las subclases contienen información adicional además de la misma información que la clase principal.	
Implementación	Se utiliza principalmente para especificar la relación entre las interfaces y las clases de implementación. Una interfaz (incluida una clase abstracta) es una colección de métodos. En una relación de implementación, una clase <u>implementa</u> una interfaz y los métodos de la clase implementan todos los métodos de la declaración de la interfaz.	
Composición	Describe como una clase <u>contiene</u> a otra clase como parte esencial de su estructura	
Agregación	Las relaciones agregadas también representan la relación entre el todo y una parte de la clase, los objeto miembros son parte del objeto gral, pero el objeto miembro puede existir independientemente del objeto gral.	
Dependencia	Las dependencias se reflejan en los métodos de una clase que utilizan el	

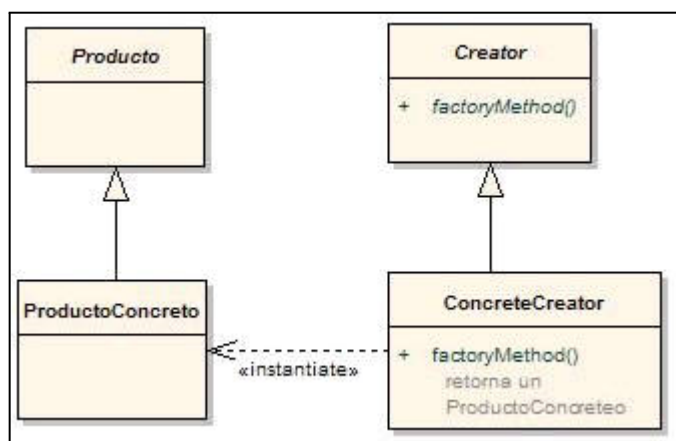
	<p>objeto de otra clase como parámetro . Una relación de dependencia es una relación de “uso”. Un cambio en una cosa en particular puede afectar a otras cosas que la usan, y usar una dependencia cuando es necesario indicar que una cosa usa otra</p>	
--	---	--

Patrones de Diseño

- **Patrones de Creación:** Permiten crear objetos sin definir la clase concreta, sólo la interfaz que debe implementar
- **Factory Method**

Libera al desarrollador sobre la forma correcta de crear objetos. Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan qué clase concreta necesitan instancias.

Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear, ya que la jerarquía de clases que tiene requiere que deba delegar la responsabilidad a una subclase.



Creator: declara el método de fabricación (creación), que devuelve un objeto de tipo Product.

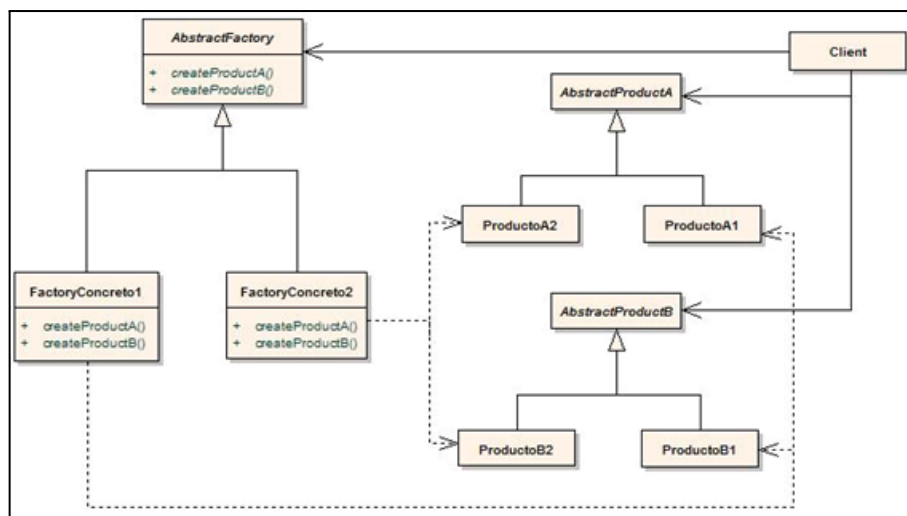
ConcreteCreator: redefine el método de fabricación para devolver un producto.

ProductoConcreto: es el resultado final. El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.

ejemplos: Transporte

- **Abstract Factory**

Este patrón crea diferentes familias de objetos. Su objetivo principal es soportar múltiples estándares que vienen definidos por las diferentes jerarquías de herencia de objetos. Es similar al Factory Method, sólo que está orientado a combinar productos.



AbstractFactory: declara una interfaz para la creación de objetos de productos abstractos.

ConcreteFactory: implementa las operaciones para la creación de objetos de productos concretos.

AbstractProduct: declara una interfaz para los objetos de un tipo de productos.

ConcreteProduct: define un objeto de producto que la correspondiente factoría concreta se encargaría de crear, a la vez que implementa la

interfaz de producto abstracto.

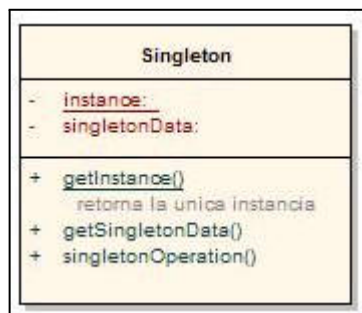
Client: utiliza solamente las interfaces declaradas en la factoría y en los productos abstractos.

Una única instancia de cada FactoryConcreto es creada en tiempo de ejecución. AbstractFactory delega la creación de productos a sus subclases FactoryConcreto.

ejemplo: muebles modernos y victorianos

- **Singleton**

La idea del patrón Singleton es proveer un mecanismo para limitar el número de instancias de una clase. Por lo tanto el mismo objeto es siempre compartido por distintas partes del código. Puede ser visto como una solución más elegante para una variable global porque los datos son abstraídos por detrás de la interfaz que publica la clase singleton. Dicho de otra manera, este patrón busca garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.



ejemplo: conexión a base de datos

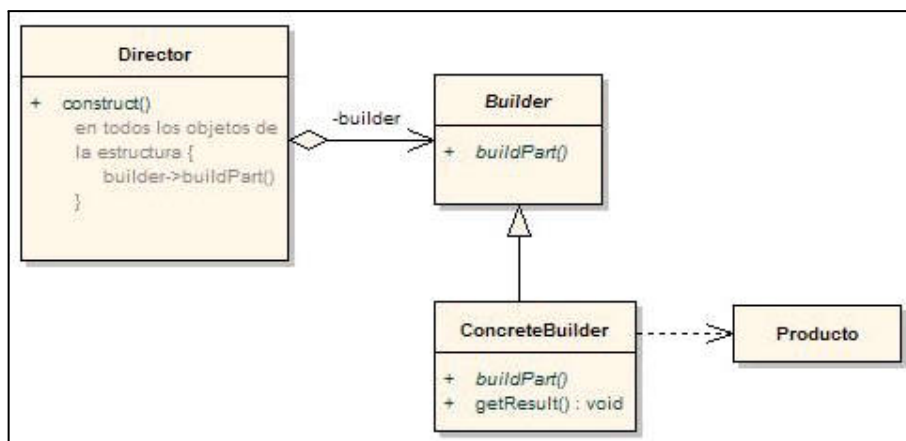
En el diagrama, la clase que es Singleton define una instancia para que los clientes puedan accederla. Esta instancia es accedida mediante un método de clase.

Los clientes (quienes quieren acceder a la clase Singleton) acceden a la única instancia mediante un método llamado `getInstance()`.

- **Builder**

Permite la creación de un objeto complejo, a partir de una variedad de partes que contribuyen individualmente a la creación y ensamblados del objeto mencionado. Hace uso de la frase "divide y conquistarás". Por otro lado, centraliza el proceso de creación en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes. Los objetos que dependen de un algoritmo tendrán que cambiar cuando el algoritmo cambia. Por lo tanto, los algoritmos que estén expuestos a dicho cambio deberían ser separados, permitiendo de esta manera reutilizar dichos algoritmos para crear diferentes representaciones.

_Producto: representa el objeto complejo a construir.



Builder: especifica una interfaz abstracta para la creación de las partes del Producto. Declara las operaciones necesarias para crear las partes de un objeto concreto.

ConcreteBuilder: implementa Builder y ensambla las partes que constituyen el objeto complejo.

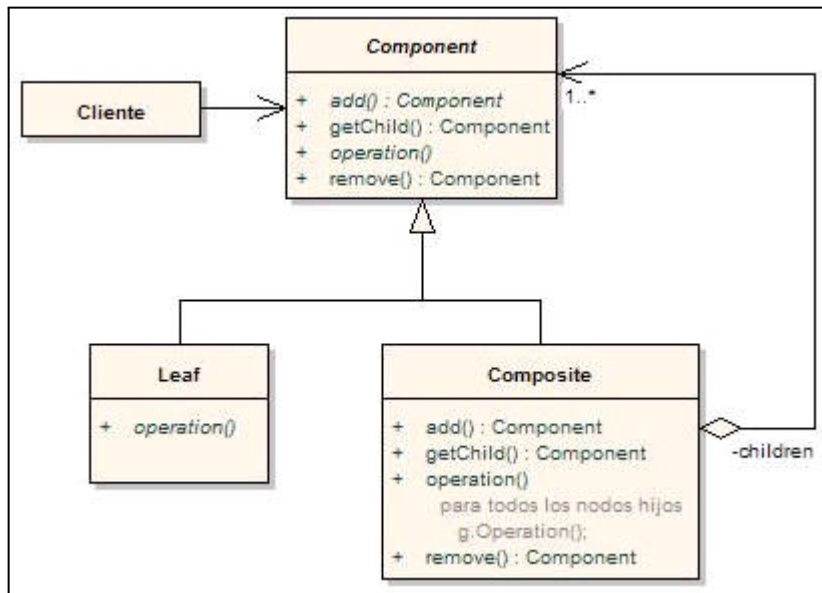
Director: construye un objeto usando la interfaz Builder. Sólo debería ser necesario especificar su tipo y así poder reutilizar el mismo proceso para distintos tipos.

ejemplo: auto fiat y ford

- **Patrones Estructurales:** Explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.

- **Composite**

El patrón Composite sirve para construir algoritmos u 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, porque todos ellos tienen una interfaz común, se tratan todos de la misma manera.



Component: implementa un comportamiento común entre las clases y declara una interfaz de manipulación a los padres en la estructura recursiva.

Leaf: representa los objetos "hoja" (no poseen hijos). Define comportamientos para objetos primitivos.

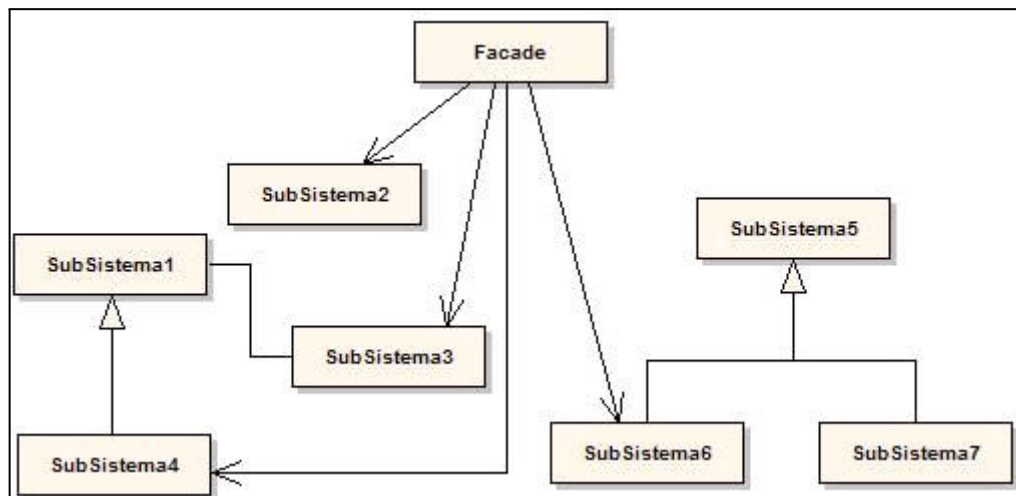
Composite: define un comportamiento para objetos con hijos. Almacena componentes hijos e implementa operaciones de relación con los hijos.

Cliente: manipula objetos de la composición a través de Component. Los clientes usan la interfaz de Component para interactuar con objetos en la estructura Composite.

• Facade

Busca simplificar el sistema, desde el punto de vista del cliente, proporcionando una interfaz unificada para un conjunto de subsistemas, definiendo una interfaz de nivel más alto. Esto hace que el sistema sea más fácil de usar.

Este patrón busca reducir al mínimo la comunicación y dependencias entre subsistemas. Para ello, utilizaremos una fachada, simplificando la complejidad al cliente. El cliente debería acceder a un subsistema a través del Facade. De esta manera, se estructura un entorno de programación más sencillo, al menos desde el punto de vista del cliente (por ello se llama "fachada").



Facade: conoce cuáles clases del subsistema son responsables de una petición. Delega las peticiones de los clientes en los objetos del subsistema.

Subsistema: manejar el trabajo asignado por el objeto Facade. No tienen ningún conocimiento del Facade (no guardan referencia de éste).

ejemplo: cine, tienda virtual

• Proxy

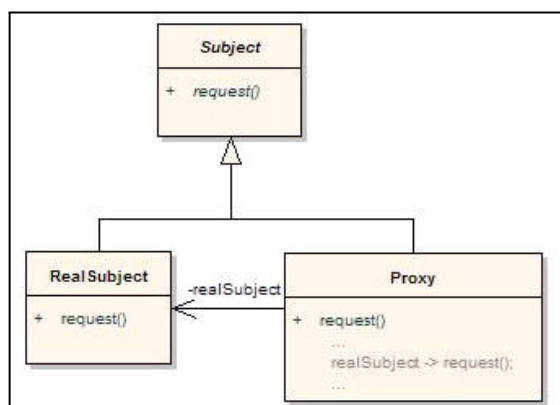
El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.

Para ello obliga que las llamadas a un objeto ocurran indirectamente a través de un objeto proxy, que actúa como un sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.

Subject: interfaz o clase abstracta que proporciona un acceso común al objeto real y su representante (proxy).

Proxy: mantiene una referencia al objeto real. Controla la creación y acceso a las operaciones del objeto real.

RealSubject: define el objeto real representado por el Proxy.



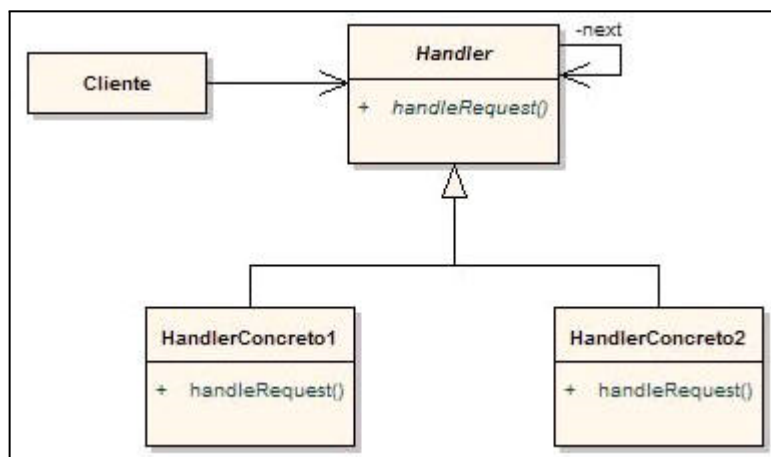
Cliente: solicita el servicio a través del Proxy y es éste quién se comunica con el RealSubject.

ejemplo: registro

- **Patrones de Comportamiento:** Guían el flujo de control del sistema
 - **Cadena de Responsabilidades**

Permite establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor. La idea es que cualquiera de los receptores pueda responder a la petición en función de un criterio establecido. Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

Busca evitar un montón de if – else largos y complejos en nuestro código, pero sobre todas las cosas busca evitar que el cliente necesite conocer toda nuestra estructura jerárquica y que rol cumple cada integrante de nuestra estructura.



Handler: define una interfaz para tratar las peticiones. Implementa el enlace al sucesor.

HandlerConcreto: trata las peticiones de las que es responsable. Si puede manejar la petición, lo hace, en caso contrario la reenvía a su sucesor.

Cliente: inicializa la petición. Conoce a un gestor que es el que lanza la petición a la cadena hasta que alguien la recoge.

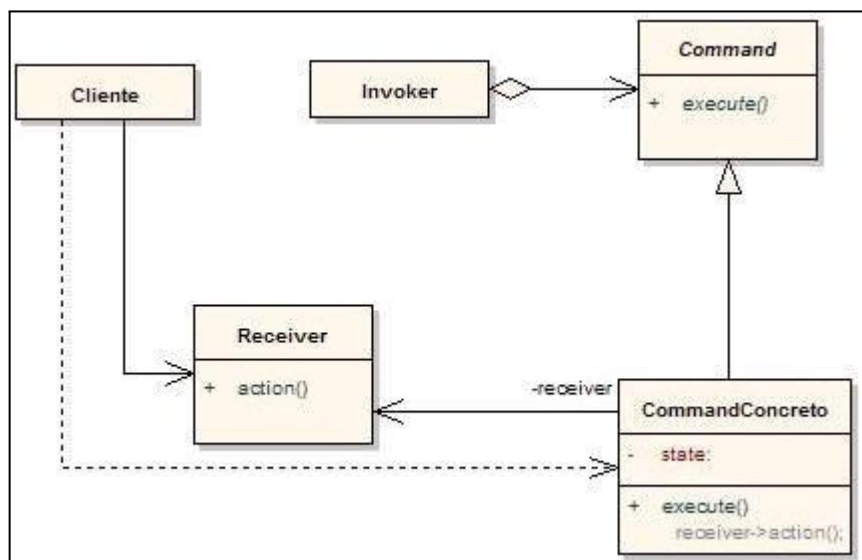
ejemplo: niveles de soporte

- **Command**

Encapsula un mensaje como un objeto. Especifica una forma simple de separar la ejecución de un comando, del entorno que generó dicho comando. Permite solicitar una operación a un objeto sin conocer el contenido ni el receptor real de la misma. Si bien estas definiciones parecen un tanto ambiguas, sería oportuno volver a leerlas luego de entender el ejemplo.

Este patrón suele establecer en escenarios donde se necesite encapsular una petición dentro de un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolarlas, guardarlas en un registro de sucesos o implementar un mecanismo de deshacer/repetir.

Lo que permite el patrón Comando es desacoplar al objeto que invoca a una operación de aquél que tiene el conocimiento necesario para realizarla. Esto nos otorga muchísima flexibilidad



Command: declara una interfaz para ejecutar una operación.

CommandConcreto: define un enlace entre un objeto "Receiver" y una acción. Implementa el método execute invocando la(s) correspondiente(s) operación(es) del "Receiver".

Cliente: crea un objeto "CommandConcreto" y establece su receptor.

Invoker: le pide a la orden que ejecute la petición.

Receiver: sabe cómo llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como receptor.

El cliente crea un objeto "CommandConcreto" y especifica su receptor. Un objeto "Invoker" almacena el objeto "CommandConcreto". El invocador envía una petición llamando al método execute sobre la orden.

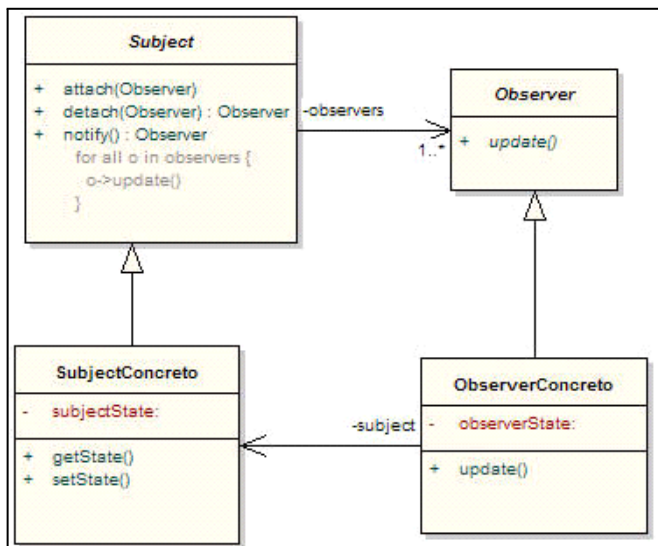
El objeto "CommandConcreto", invoca operaciones de su receptor para llevar a cabo la petición.

ejemplo: control remoto de luz y ventilador

- **Observer**

Este patrón de diseño permite reaccionar a ciertas clases llamadas observadores sobre un evento determinado.

Es usado en programación para monitorear el estado de un objeto en un programa. Está relacionado con el principio de invocación implícita. La motivación principal de este patrón es su utilización como un sistema de detección de eventos en tiempo de ejecución. Es una característica muy interesante en términos del desarrollo de aplicaciones en tiempo real. Este patrón tiene un uso muy concreto: varios objetos necesitan ser notificados de un evento y cada uno de ellos decide cómo reaccionar cuando este evento se produzca.



Subject: conoce a sus observadores y ofrece la posibilidad de añadir y eliminar observadores. Posee un método llamado attach () y otro detach() que sirven para agregar o remover observadores en tiempo de ejecución.

Observer: define la interfaz que sirve para notificar a los observadores los cambios realizados en el Subject.

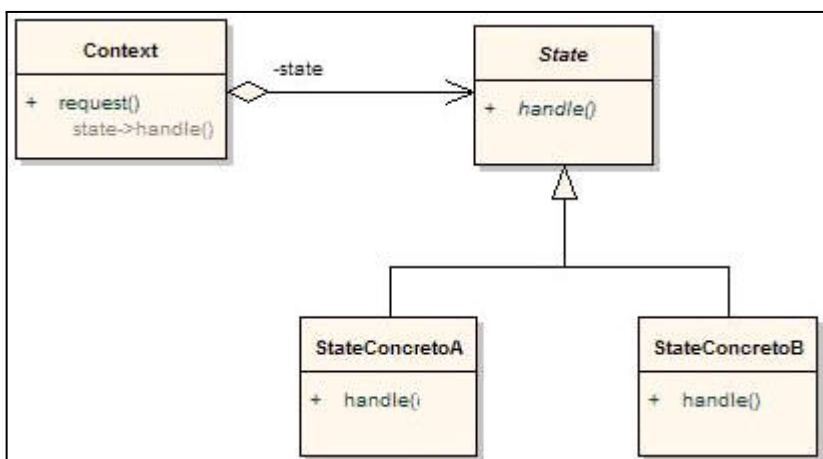
SubjectConcreto: almacena el estado que es objeto de interés de los observadores y envía un mensaje a sus observadores cuando su estado cambia.

ObserverConcreto: mantiene una referencia a un SubjectConcreto. Almacena el estado del Subject que le resulta de interés. Implementa la interfaz de actualización de Observer para mantener la consistencia entre los dos estados.

ejemplo:

- **Estado**

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.



Command: declara una interfaz para ejecutar una operación.

CommandConcreto: vincula un objeto Receiver y una acción. Implementa execute invocando las operaciones del Receiver.

Cliente: crea un objeto "CommandConcreto" y establece su receptor.

Invoker: le pide a la orden que ejecute la petición.

Receiver: sabe cómo llevar a cabo las operaciones asociadas a una petición.

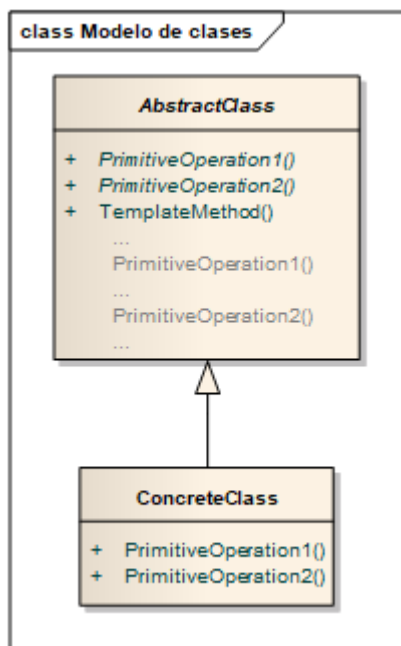
El cliente crea un objeto "CommandConcreto" y especifica su receptor. Un objeto "Invoker" almacena el objeto "CommandConcreto". El invocador envía una petición llamando al método execute sobre la orden. El objCommand Concreto, invoca operaciones de su receptor para llevar a cabo la petición.

ejemplo:

- **Template**

Define una estructura algorítmica cuya lógica quedará a cargo de las subclases. Para ello, escribe una clase abstracta que contiene parte de la lógica necesaria para realizar su finalidad. En ella se define una estructura de herencia que sirve de plantilla ("Template" significa plantilla) de los métodos en las subclases.

Dicho de otra forma, define un esqueleto de un algoritmo, delegando algunos pasos a las subclases. Permite redefinir parte de dicho algoritmo sin cambiar su estructura. Este patrón se vuelve de especial utilidad cuando es necesario realizar un algoritmo que sea común para muchas clases, pero con pequeñas variaciones entre una y otras. En este caso, se deja en las subclases cambiar una parte del algoritmo.



AbstractTemplate o AbstractClass: implementa un método plantilla que define el esqueleto de un algoritmo y define métodos abstractos que deben implementar las subclases concretas

TemplateConcreto o ConcreteClass: implementa los métodos abstractos para realizar los pasos del algoritmo que son específicos de la subclase.

ejemplo: