PATRONES GOF

1. INTRODUCCION.

GOF: Los ahora famosos Gang of Four (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Son los autores del famoso libro "Design Patterns: Elements of Reusable Object Oriented Software".

Los patrones de diseño tratan los problemas del diseño de software que se repiten y que se presentan en situaciones particulares, con el fin de proponer soluciones a ellas. Son soluciones exitosas a problemas comunes.

2. TIPO DE PATRONES

1) Clasificación según su propósito.

- Creacionales: definen la mejor manera en que un objeto es instanciado. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.

- De Comportamiento: permiten definir la comunicación entre los objetos del sistema y el flujo de la información entre los mismos.

- Estructurales: permiten crear grupos de objetos para ayudarnos a realizar tareas complejas

2) Clasificación según alcance

- De clase: se basados en la herencia de clases.
- De objeto: se basan en la utilización dinámica de objetos.

		PROPÓSITO		
		CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
SCOPE	CLASE	Factory Method	Adapter	Interpreter Template Method
	OBJETO	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

3. PATRONES CREACIONALES

son patrones que abstraen el proceso de instanciación. Procuran independizar el sistema de cómo sus objetos son creados, compuestos y representados e indican soluciones de diseño para encapsular el conocimiento acerca de las clases que el sistema usa y ocultar cómo se crean instancias de estas clases.

ABSTRACT FACTORY

INTENCION

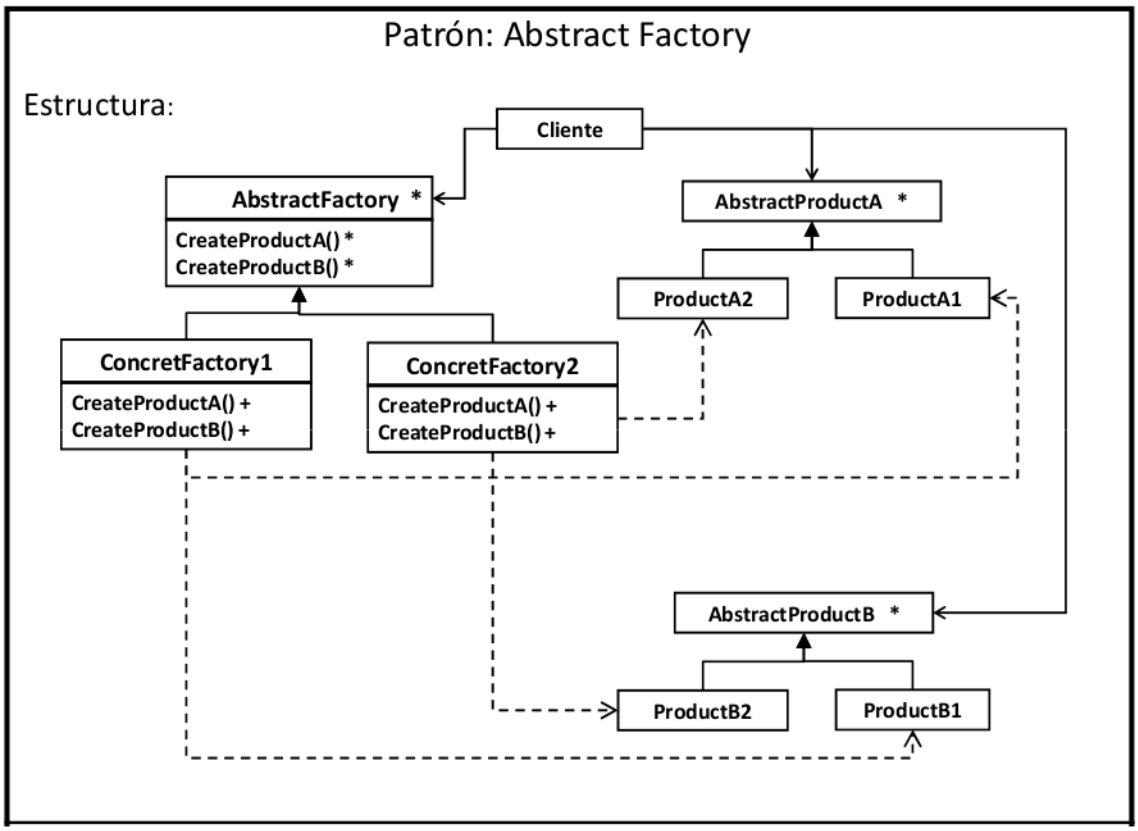
Proveer una interfaz para crear familias de objetos dependientes o relacionados sin especificar sus clases concretas.

ALIAS

Kit (como sufijo usualmente).

APLICABILIDAD

- Usamos este patrón cuando un sistema debe independizarse de cómo sus productos son creados,compuestos y representados.
- Un sistema debe ser configurado con una de múltiples familias de productos.
- Una familia de objetos debe ser usada en conjunto y debe reforzarse este hecho.
- Queremos proveer una librería de productos, pero sólo publicitar las interfaces, no las implementaciones



FACTORY METHOD

INTENCION

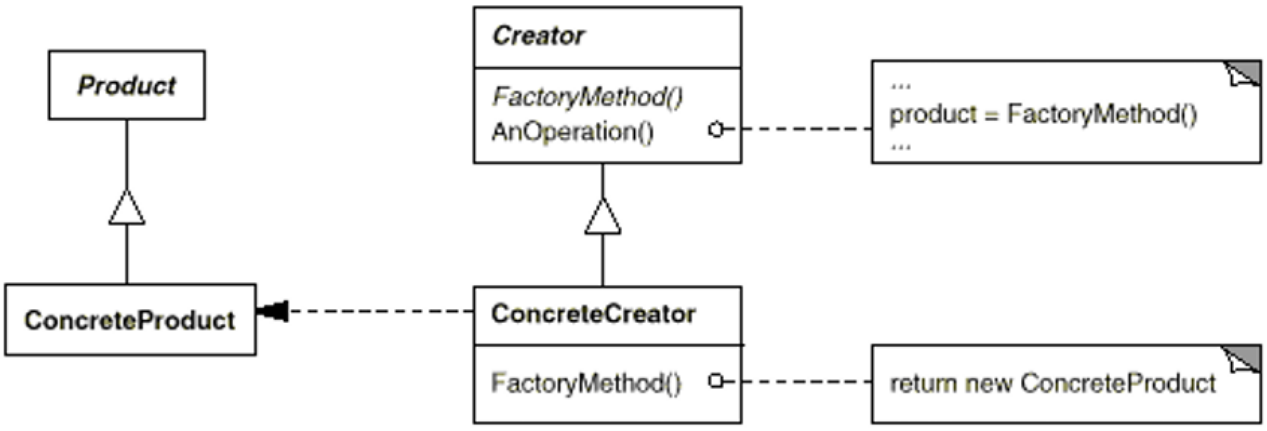
Define una interfaz para crear objetos, pero deja elegir a las subclases que clases inicializar. Factory Method deja a la clase ceder la inicialización a las subclases.
Define una interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.
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.

ALIAS

constructor virtual.

APLICABILIDAD

- Una clase no puede anticipar el tipo de objeto que debe crear y quiere que sus subclases especifiquen dichos objetos.
- Hay clases que delegan responsabilidades en una o varias subclases. Una aplicación es grande y compleja y posee muchos patrones creacionales.



PARTICIPANTES

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

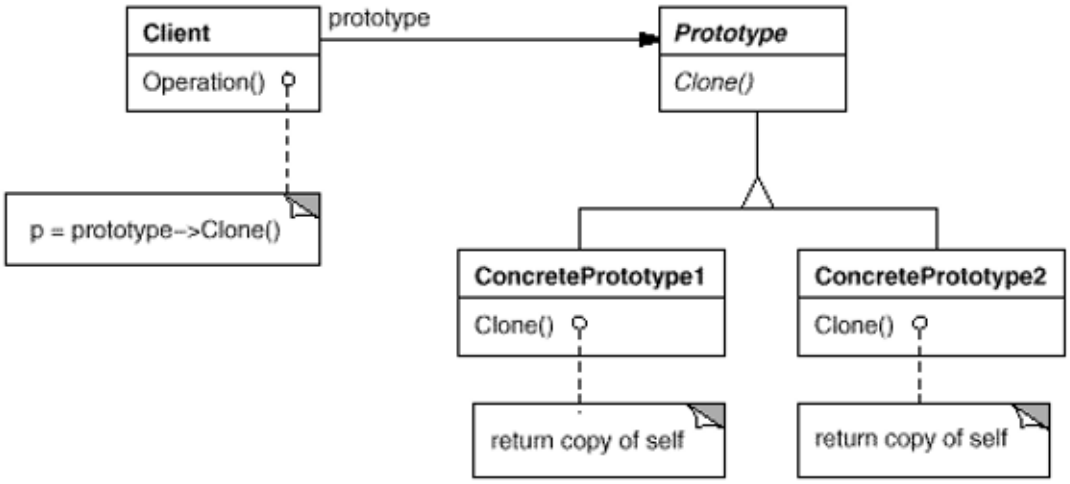
PROTOTYPE

INTENCION

especifica los tipos de objetos a crear utilizando una instancia prototipo,
y crea nuevos objetos copiando esta instancia.

APLICABILIDAD

- Usamos este patrón cuando las clases a instanciar son especificadas en tiempo de ejecución, por ejemplo, por carga dinámica.
- Cuando instancias de una clase pueden tener sólo uno de muchas combinaciones de estados, lo que puede obtenerse clonando prototipos en lugar de hacerlo manualmente.



PARTICIPANTES

- Prototype: declara una interfaz para la autoclonación.
- ConcretePrototype: implementa una operación para clonarse a sí mismo.
- Client: crea un nuevo objeto solicitándole al prototipo que se clone a sí mismo.

CONSECUENCIAS

- Un programa puede dinámicamente añadir y borrar objetos prototipo en tiempo de ejecución. Esta es una ventaja que no ofrece ninguno de los otros patrones de creación.
- Esconde los nombres de los productos específicos al cliente.
- Se pueden especificar nuevos objetos prototipo variando los existentes.
- La clase Cliente es independiente de las clases exactas de los objetos prototipo que utiliza, y además, no necesita conocer los detalles de cómo construir los objetos prototipo.
- Clonar un objeto es más rápido que crearlo.
- Se desacopla la creación de las clases y se evita repetir la instanciación de objetos con parámetros repetitivos.

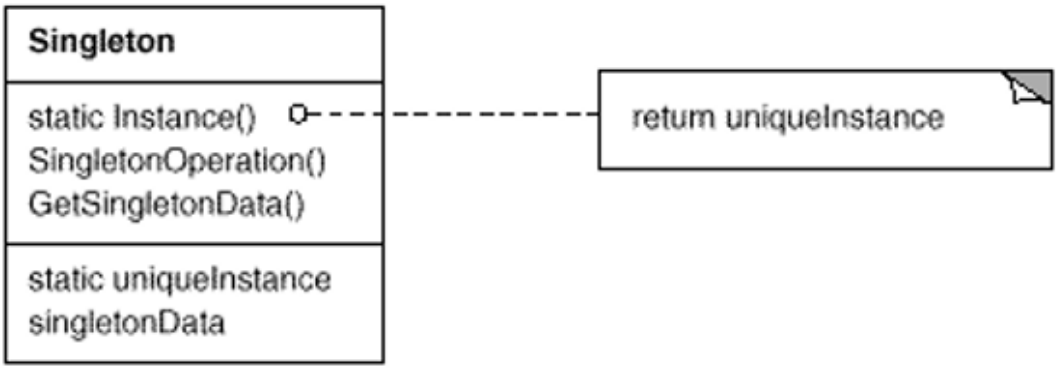
SINGLETON

INTENCION

éste patrón busca garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

APLICABILIDAD

- Debe haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.
- Se requiere de un acceso estandarizado y conocido públicamente.



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().

InstitutoEducativo.java

```
package creacional.singleton;

public class InstitutoEducativo {
    private static InstitutoEducativo instance;
    // Se coloca un variable del mismo tipo que la clase llamada, por convenciòn
    // "instance". Aquì reside todo el secreto de este patrón, ya que es dicha variable
    // la que se instancia por ùnica vez y se devuelve al cliente.

    private InstitutoEducativo(){
    }
    // Se privatiza el constructor para que no se puede hacer un new InstitutoEducativo()
    // desde otro lugar que no sea dentro de la misma clase.

    public static InstitutoEducativo getInstance(){
        if (instance == null) {
            instance = new InstitutoEducativo();
        }
        return instance;
    }
    // Para utilizar la única instancia de la clase, los clientes deberàn convocar
    // al método getIntance(). Si observan la condición del if solo será true
    // la primera vez.
}
```

4. PATRONES ESTRUCTURALES

Los patrones estructurales se refieren a cómo las clases y los objetos son organizados para conformar estructuras más grandes.
Pueden ser:

- patrones de clases, basados en la utilización de herencia, o
- patrones de objetos, basados en la técnica de composición.
- La mayor variedad se encuentra en los patrones estructurales de objetos,
- en donde los objetos se componen para obtener nuevas funcionalidades.

ADAPTER

INTENCION

Convertir la interfaz de una clase en otra interfaz que el cliente espera.

ALIAS

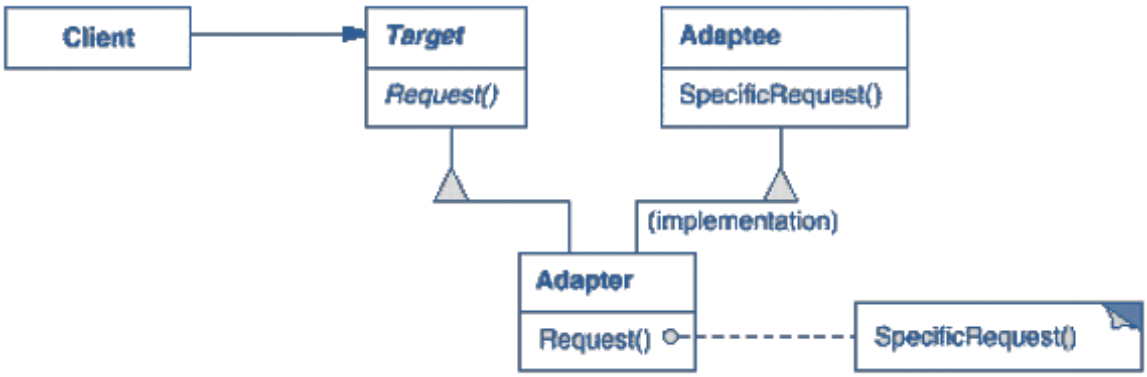
Wrapper

APLICABILIDAD

- Deseamos usar una clase existente, pero su interfano es la que necesitamos.
- Queremos crear una clase reutilizable que coopera con otras clases no relacionadas, probablemente con interfaces incompatibles.
- Necesitamos usar varias subclases, pero no es práctico adaptar sus interfaces heredando de ellas, por lo que apelamos a un objeto adaptador. En este caso se utiliza la versión Object Adapter.

Básicamente, es un encapsulado de los métodos y una traducción directa. Lo usaremos cuando querramos adaptar una clase con interfaz diferente, a la interfaz que necesitamos.

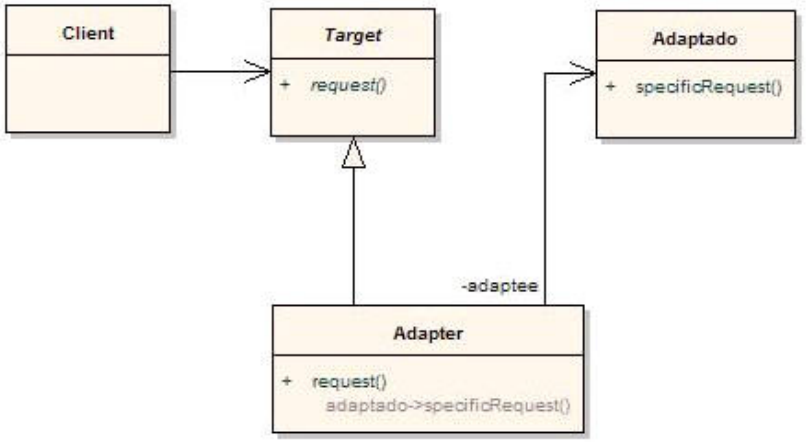
Class adapter version.



PARTICIPANTES

- Target: define la interfaz que el cliente usa.
- Adaptee: interfaz existente que necesita adaptarse.
- Adapter: adapta la interfaz de Adaptee a la que necesita el cliente.

Objetc adapter version.



PARTICIPANTES

- Target: define la interfaz que el cliente usa.
- Adaptee: interfaz existente que necesita adaptarse.
- Adapter: adapta la interfaz de Adaptee a la que necesita el cliente.

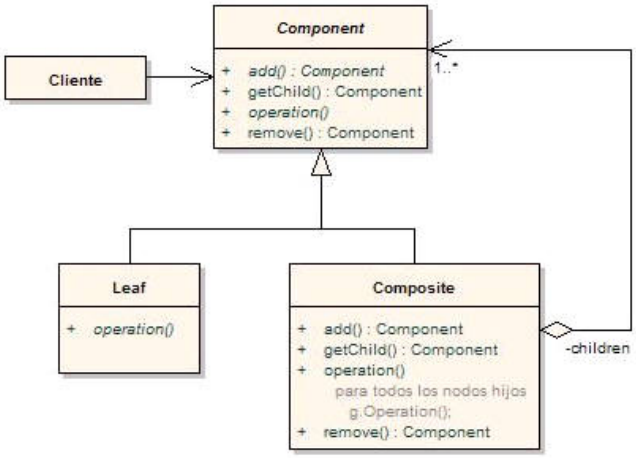
COMPOSITE

INTENCION

Compone objetos en estructuras de árbol para representar jerarquías de relación “es-parte-de”.

APLICABILIDAD

- Queremos representar jerarquías de objetos modelando la relación “es-parte-de” (part-whole hierarchies).
- Queremos que el cliente ignore la distinción entre objetos compuestos y objetos individuales. El cliente tratará todos los objetos de la estructura compuesta de manera uniforme.



PARTICIPANTES

- Component: implementa un comportamiento común entre las clases y declara una interface 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. 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. Si el receptor es una hoja, la interacción es directa. Si es un Composite, se debe llegar a los objetos “hijos”, y puede llevar a utilizar operaciones adicionales.

DECORATOR

INTENCION

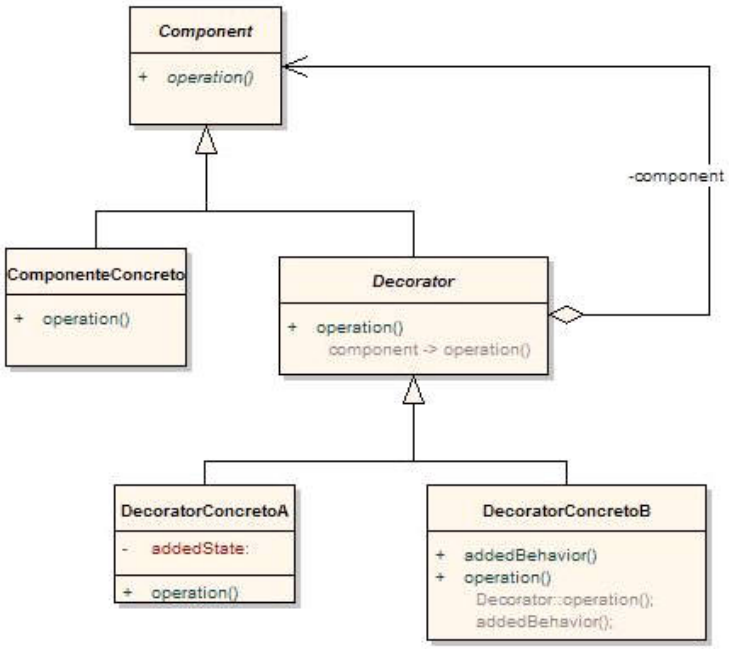
Agrega responsabilidades a un objeto de manera dinámica. Provee una alternativa a la herencia cuando deseamos agregar funcionalidad.

ALIAS

Wrapper

APLICABILIDAD

- Queremos agregar responsabilidades a objetos individuales de manera dinámica y transparente, sin afectar otros objetos.
- Deseamos implementar responsabilidades que pueden removerse.
- Cuando la extensión utilizando herencia es impracticable, por ejemplo, pues provocaría demasiadas variaciones y un gran número de clases.



PARTICIPANTES

- Component: define la interfaz para los objetos que pueden tener responsabilidades agregadas dinámicamente.
- ConcreteComponent: define un objeto al cual se le pueden agregar responsabilidades dinámicamente.
- Decorator: mantiene una referencia a un objeto Component y define la interfaz que conforma la interfaz de Component.
- ConcreteDecorator: agrega responsabilidades al componente.

5. PATRONES DE COMPORTAMIENTO

Los patrones de comportamiento se centran en los algoritmos y la asignación de responsabilidades entre los objetos. Son patrones tanto de clases y objetos (similares a los anteriores) como de comunicación entre ellos.
Caracterizan flujo de control complejo.

Los **patrones de comportamiento de clases** (behavioral class patterns) utilizan herencia para distribuir el comportamiento entre las clases.

Los **patrones de comportamiento de objetos** (behavioral object patterns) utilizan composición de objetos en lugar de herencia. Algunos describen cómo los objetos cooperan entre sí para realizar una tarea compleja, imposible para sólo uno de ellos.

COMMAND

INTENCION

Encapsular el pedido (mensaje, solicitud) como un objeto, permitiendo parametrizar los clientes con diferentes pedidos, encolar o registrar los pedidos y proveer operaciones para deshacer pedidos previos. 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.

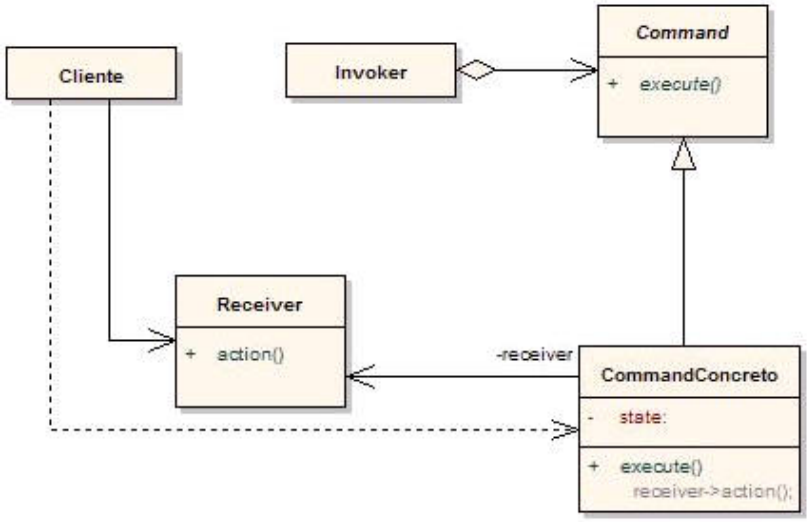
ALIAS

Action, Transaction.

APLICABILIDAD

- Parametrizar objetos para una acción a realizar.
- Especificar, encolar y ejecutar pedidos en momentos diferentes.
- Proveer la posibilidad de deshacer acciones.
- Proveer registros de auditoría y salvaguarda.
- Estructurar un sistema en función de operaciones de alto nivel construidas en base a operaciones primitivas.

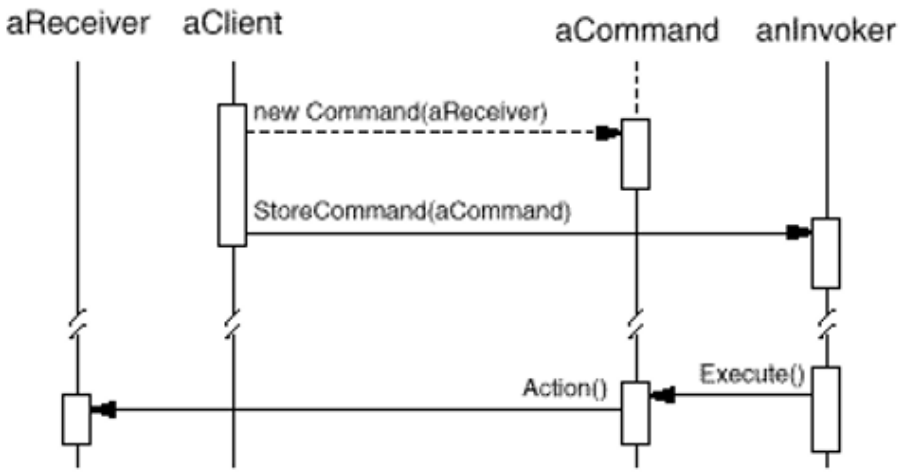
Este patrón suele establecer en escenarios donde se necesite encapsular una petición dentro de un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolarlas, guardarlas en un registro de sucesos o implementar un mecanismo de deshacer/repetir.



PARTICIPANTES

- 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 como 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.

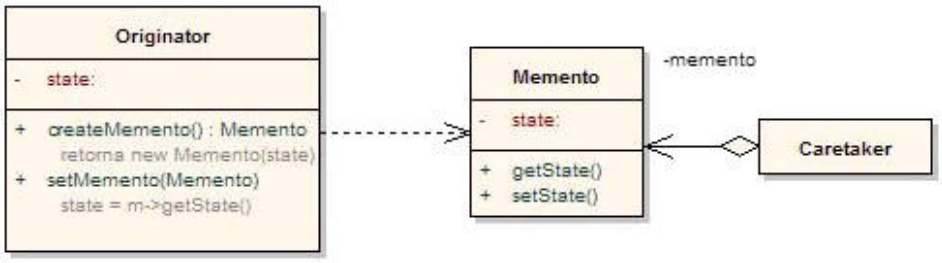


MEMENTO

INTENCION
permite capturar y exportar el estado interno de un objeto para que luego se pueda restaurar, sin romper la encapsulación.
Su finalidad es almacenar el estado de un objeto (o del sistema completo) en un momento dado, de manera que se pueda restaurar posteriormente si fuese necesario. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece el objeto (pero sin romper la encapsulación), de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior.

APLICABILIDAD

- Se necesite restaurar el sistema desde estados pasados.
- Se quiera facilitar el hacer y deshacer de determinadas operaciones, para lo que habrá que guardar los estados anteriores de los objetos sobre los que se opere (o bien recordar los cambios de forma incremental).



PARTICIPANTES

- Caretaker: es responsable por mantener a salvo a Memento. No opera o examina su contenido
- Memento: almacena el estado interno de un objeto Originator. El Memento puede almacenar todo o parte del estado.
- Originator: crea un objeto Memento conteniendo una fotografía de su estado interno.

CONSECUENCIAS

- No es necesario exponer el estado interno como atributos de acceso público, preservando así la encapsulación.
- Si el originador tendría que de almacenar y mantener a salvo una o muchas copias de su estado interno, sus responsabilidades crecerían y sería inmanejable.
- El uso frecuente de Mementos para almacenar estados internos de gran tamaño, podría resultar costoso y perjudicar la performance del sistema.
- Caretaker no puede hacer predicciones de tiempo ni de espacio.

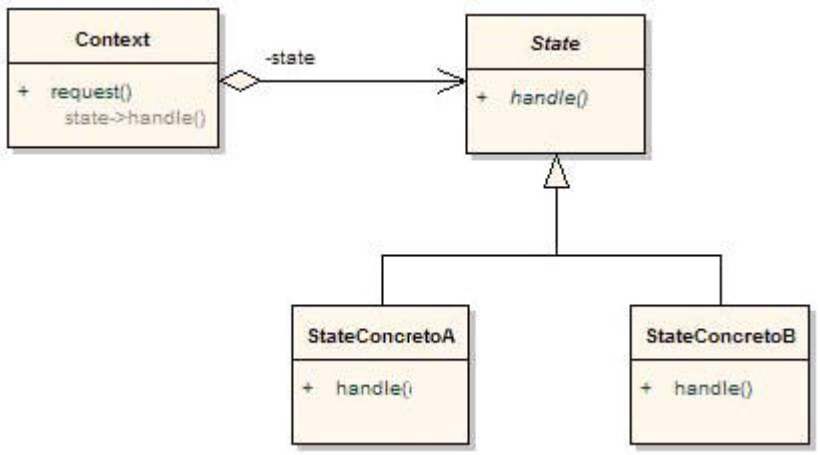
STATE

INTENCION

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Busca que un objeto pueda reaccionar según su estado interno.

APLICABILIDAD

- 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



PARTICIPANTES

- Context: mantiene una instancia con el estado actual
- State: define interfaz para el comportamiento asociado a un determinado estado del Contexto.
- StateConcreto: cada subclase implementa el comportamiento asociado con un estado del contexto.

El Context delega el estado específico al objeto StateConcreto actual. Un objeto Context puede pasarse a sí mismo como parámetro hacia un objeto State. De esta manera la clase State puede acceder al contexto si fuese necesario. Context es la interfaz principal para el cliente. El cliente puede configurar un contexto con los objetos State. Una vez hecho esto estos clientes no tendrán que tratar con los objetos State directamente. Tanto el objeto Context como los objetos de StateConcreto pueden decidir el cambio de estado.
Los estados concretos suelen ser Singleton.

CONSECUENCIAS

- Se localizan fácilmente las responsabilidades de los estados concretos, 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.
- 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.
- Facilita la ampliación de estados mediante una simple herencia, sin afectar al Context.
- Permite a un objeto cambiar de estado en tiempo de ejecución.
- Los estados pueden reutilizarse: varios Context pueden utilizar los mismos estados.
- Se incrementa el número de subclases.

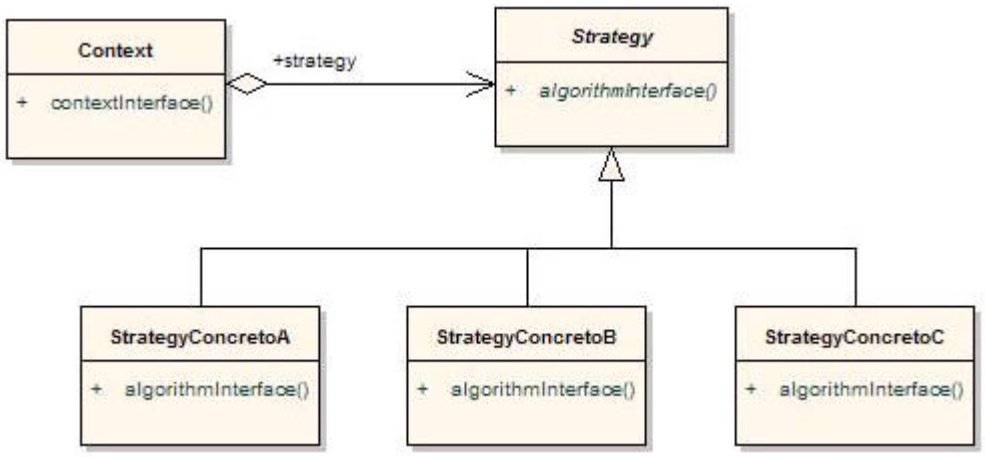
STRATEGY

INTENCION

Encapsula algoritmos en clases, permitiendo que éstos sean re-utilizados e intercambiables. En base a un parámetro, que puede ser cualquier objeto, permite a una aplicación decidir en tiempo de ejecución el algoritmo que debe ejecutar.
La esencia de este patrón es encapsular algoritmos relacionados que son subclases de una superclase común, lo que permite la selección de un algoritmo que varia según el objeto y también le permite la variación en el tiempo. Esto se define en tiempo de ejecución. Este patrón busca desacoplar bifurcaciones inmensas con algoritmos dificultosos según el camino elegido

APLICABILIDAD

- Un programa tiene que proporcionar múltiples variantes de un algoritmo o comportamiento.
- Es posible encapsular las variantes de comportamiento en clases separadas que proporcionan un modo consistente de acceder a los comportamientos.
- Permite cambiar o agregar algoritmos, independientemente de la clase que lo utiliza.



PARTICIPANTES

- Strategy: declara una interfaz común a todos los algoritmos soportados.
- StrategyConcreto: implementa un algoritmo utilizando la interfaz Strategy. Es la representación de un algoritmo.
- Context: mantiene una referencia a Strategy y según las características del contexto, optará por una estrategia determinada.
- Context / Cliente: solicita un servicio a Strategy y este debe devolver el resultado de un StrategyConcreto.

CONSECUENCIAS

- Permite que los comportamientos de los Clientes sean determinados dinámicamente sobre un objeto base.
- Simplifica los Clientes: les reduce responsabilidad para seleccionar comportamientos o implementaciones de comportamientos alternativos. Esto simplifica el código de los objetos Cliente eliminando las expresiones if y switch.
- En algunos casos, esto puede incrementar también la velocidad de los objetos Cliente porque ellos no necesitan perder tiempo seleccionado un comportamiento.

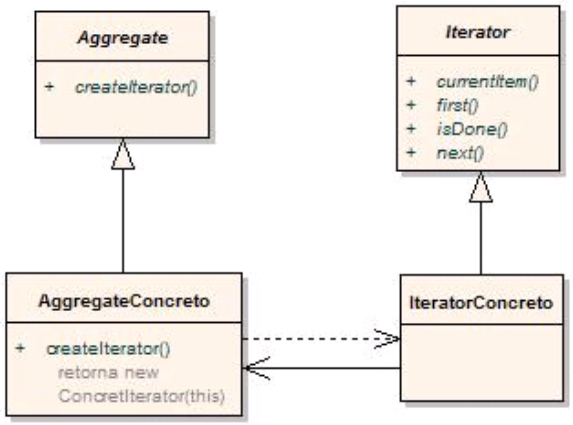
ITERATOR

INTENCION
Provee un modo de acceder a los elementos de un objeto agregado en forma secuencial, sin exponer su representación interna.

ALIAS
Cursor

APLICABILIDAD

- Queremos acceder al contenido de los objetos sin exponer su representación interna.
- Queremos proveer múltiples recorridos de objetos agregados.
- Queremos proveer una interfaz uniforme para acceder diferentes estructuras agregadas (iteración polimórfica).



PARTICIPANTES

- Agregado/Aggregate: define una interfaz para crear un objeto iterator.
- Iterator: define la interfaz para acceder y recorrer los elementos de un agregado.
- IteradorConcreto: implementa la interfaz del iterador y guarda la posición actual del recorrido en cada momento.
- AgregadoConcreto: implementa la interfaz de creación de iteradores devolviendo una instancia del iterador concreto apropiado.

CONSECUENCIAS

- Es posible acceder a una colección de objetos sin conocer el código de los objetos.
- Utilizando varios objetos Iterator, es simple tener y manejar varios recorridos al mismo tiempo.
- Es posible para una clase Colecction proporcionar diferentes tipos de objetos Iterator que recorran la colección en diferentes modos.

- Las clases Iterator simplifican el código de las colecciones, ya que el código de los recorridos se encuentra en los Iterators y no en las colecciones.

VISITOR

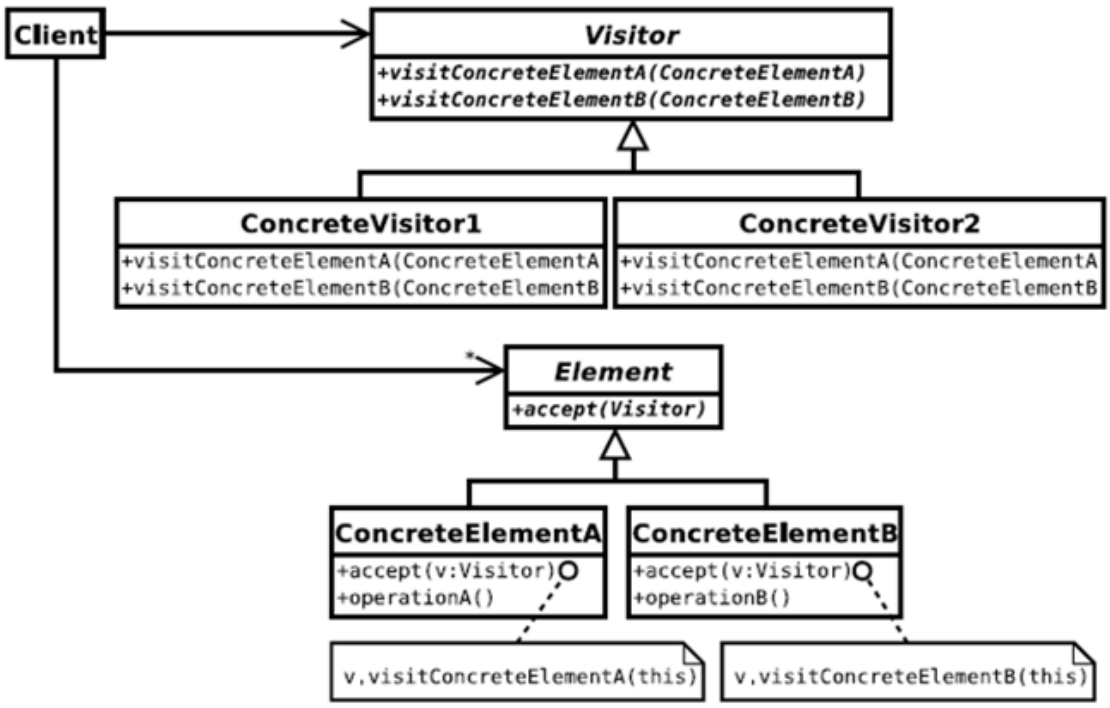
INTENCION

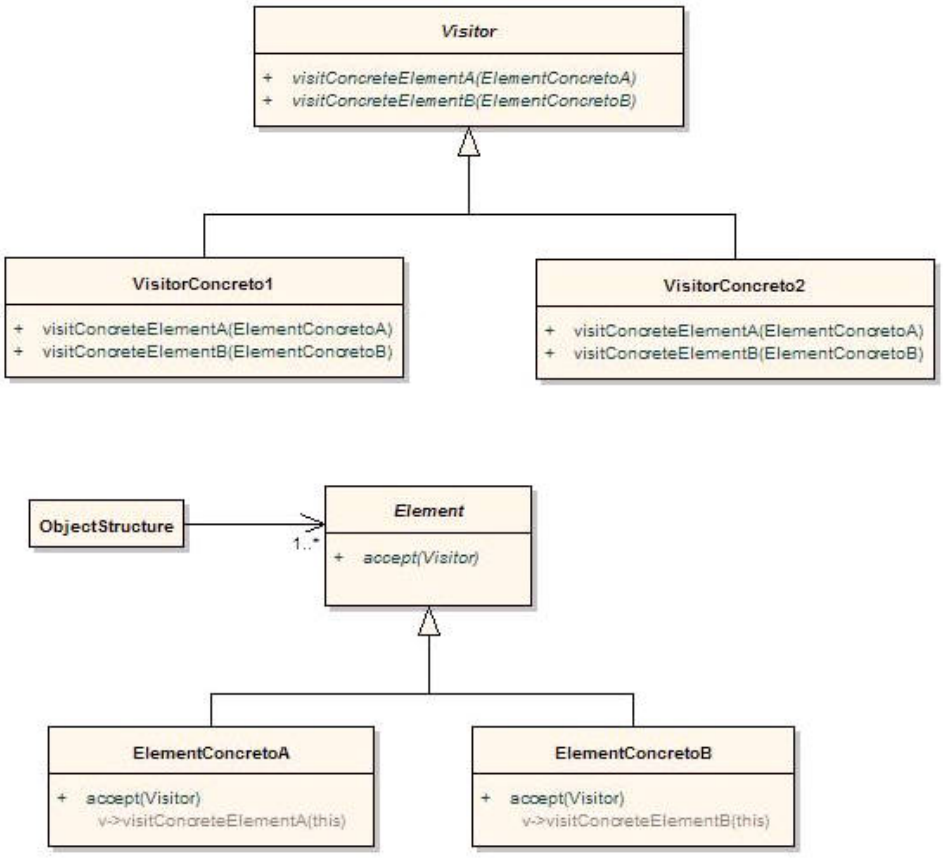
Busca separar un algoritmo de la estructura de un objeto. La operación se implementa de forma que no se modifique el código de las clases sobre las que opera.

Si un objeto es el responsable de mantener un cierto tipo de información, entonces es lógico asignarle también la responsabilidad de realizar todas las operaciones necesarias sobre esa información. La operación se define en cada una de las clases que representan los posibles tipos sobre los que se aplica dicha operación, y por medio del polimorfismo y la vinculación dinámica se elige en tiempo de ejecución qué versión de la operación se debe ejecutar. De esta forma se evita un análisis de casos sobre el tipo del parámetro.

APLICABILIDAD

- Una estructura de objetos contiene muchas clases de objetos con distintas interfaces y se desea llevar a cabo operaciones sobre estos objetos que son distintas en cada clase concreta.
- Se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir 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.





PARTICIPANTES

- Visitor: declara una operación de visita para cada uno de los elementos concretos de la estructura de objetos. Esto es, el método visit().
- VisitorConcreto : implementa cada una de las operaciones declaradas por Visitor.
- Element: define la operación que le permite aceptar la visita de un Visitor.
- ConcreteElement: implementa el método accept() que se limita a invocar su correspondiente método del Visitor.
- ObjectStructure: gestiona la estructura de objetos y puede ofrecer una interfaz de alto nivel para permitir a los Visitor visitar a sus elementos.

El Element ejecuta el método de visitar y se pasa a sí mismo como parámetro.

CONSECUENCIAS

- Facilita la inclusión de nuevas operaciones.
- Agrupa las operaciones relacionadas entre sí.
- La inclusión de nuevos ElementsConcretos es una operación costosa.
- Posibilita visitar distintas jerarquías de objetos u objetos no relacionados por un padre común.