Creacionales

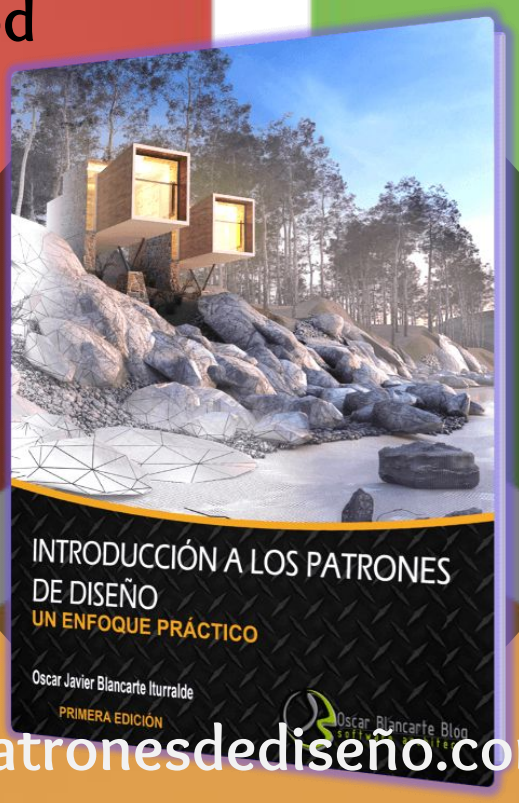
Restringen o controlan la forma en que creamos los objetos, evitando que el usuario utilice la instrucción new para crear nuevas instancias

Abstract Factory
Factory Method
Object Pool
Prototype
Singleton
Builder

Estructurales

Describen cómo los objetos y clases se pueden combinar para formar estructuras más grandes y complejas.

Composite
Decorator
Flyweight
Adapter
Facade
Bridge
Proxy



patronesdediseño.com

colaboración, relaciones y delegación de responsabilidades entre otras clases logrando con esto simplificar la forma en que los objetos se comunican e interactúan entre sí

Iterator
Mediator
Memento

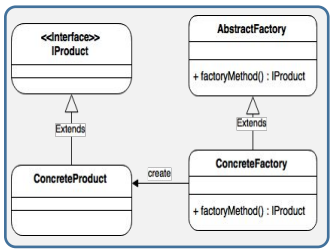
Templete Method
Strategy
Chain of Resp

Iterator
Command
Observer

Comportamiento

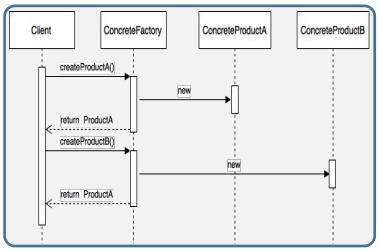
Oscar Blancarte Blog
software architect

Patrón Factory Method



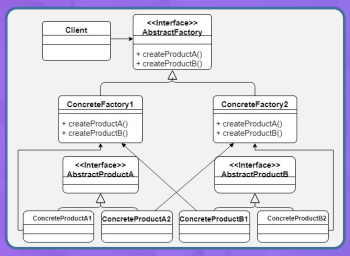
Tipo: Creacional

Factory Method permite la creación de objetos de un subtipo determinado a través de una clase *Factory*. Esto es especialmente útil cuando no sabemos, en tiempo de diseño, el subtipo que vamos a utilizar o cuando queremos delegar la lógica de creación de los objetos a una clase *Factory* .



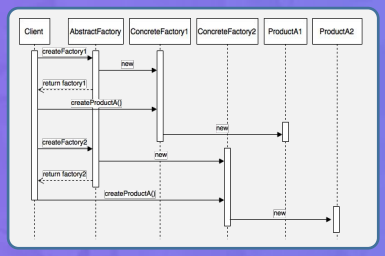
1. El cliente le solicita al **ConcreteFactory** la creación del **ProductA**.
2. El **ConcreteFactory** localiza la implementación concreta de **ProductA** y crea una nueva instancia.
3. El **ConcreteFactory** regresa el **ConcreteProductA** creado.
4. El cliente le solicita al **ConcreteFactory** la creación del **ProductB**.
5. El **ConcreteFactory** localiza la implementación concreta del **ProductB** y crea una nueva instancia.
6. El **ConcreteFactory** regresa el **ConcreteProductB** creado.

Patrón Abstract Factory



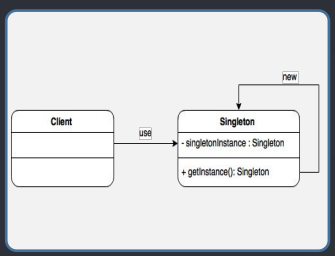
Tipo: Creacional

Agrupar un conjunto de clases que tiene un funcionamiento en común llamadas familias, las cuales son creadas mediante un Factory, este patrón es especialmente útil cuando requerimos tener ciertas familias de clases para resolver un problema, sin embargo, puede que se requieran crear implementaciones paralelas de estas clases para resolver el mismo problema pero con una implementación distinta.



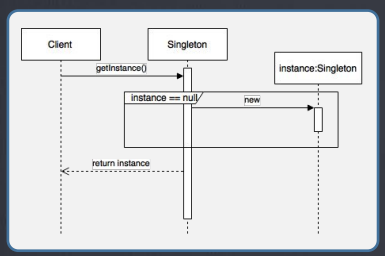
1. El cliente solicita la creación del ConcreteFactory1 al AbstractFactory.
2. El AbstractFactory crea una instancia del ConcreteFactory1 y la regresa.
3. El cliente le solicita al ConcreteFactory1 la creación de un ProductA.
4. El ConcreteFactory1 crea una instancia del ProductA1 que es parte de la familia1 y lo regresa.
5. El cliente esta vez solicita la creación del ConcreteFactory2 al AbstractFactory.
6. El AbstractFactory crea una instancia del ConcreteFactory2.
7. El cliente le solicita al ConcreteFactory2 la creación de un ProductA.
8. El ConcreteFactory2 retorna una instancia del ProductA2 el cual es parte de la familia2.

Patrón Singleton



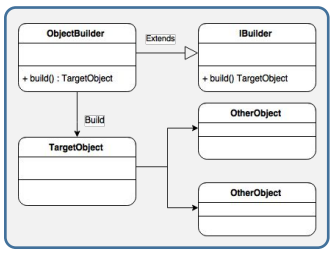
Tipo: Creacional

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



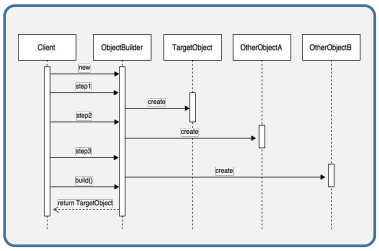
1. El cliente solicita la instancia al Singleton mediante el método estático **getInstance**.
2. El **Singleton** validará si la instancia ya fue creada anteriormente, de no haber sido creada entonces se crea una nueva.
3. Se regresa la instancia creada en el paso anterior o se regresa la instancia existente en otro caso.

Patrón Builder



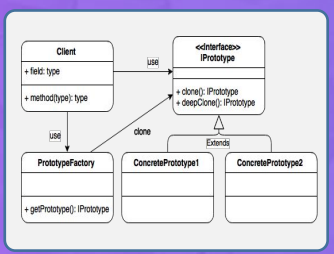
Tipo: Creacional

Permite crear objetos complejos a través de uno más simple, el cual, mediante métodos de utilidad, va creando secciones de un objeto más grande, de tal forma que olvidamos de la complejidad de la estructura interna de los objetos.



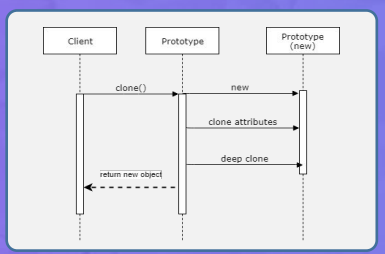
1. El cliente crea una instancia del **ObjectBuilder**.
2. El cliente ejecuta el paso 1 de la creación en el **ObjectBuilder**.
3. Internamente el **ObjectBuilder** crea al **TargetObject**.
4. El cliente ejecuta el paso 2 de la creación en el **ObjectBuilder**.
5. Internamente el **ObjectBuileder** crea un **OtherObjectA**.
6. El cliente ejecuta el paso 3 de la creación en el **ObjectBuilder**.
7. Internamente el **ObjectBuilder** crear el **OtherObjectB**.
8. El cliente solicita al **ObjectBuilder** la creación del **TargetObject**, éste toma todos los objetos creados anteriormente, los asocia al **TargetObject** y lo regresa.

Patrón Prototype



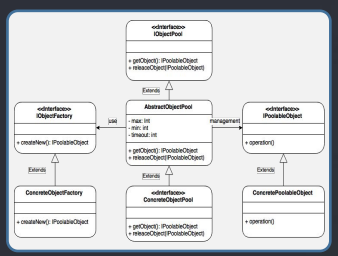
Tipo: Creacional

basa su funcionalidad en la clonación de objetos, estos nuevos objetos son creados mediante un pool de prototipos elaborados previamente y almacenados. Este patrón es especialmente útil cuando necesitamos crear objetos basados en otros ya existentes o cuando se necesita la creación de estructuras de objetos muy grandes



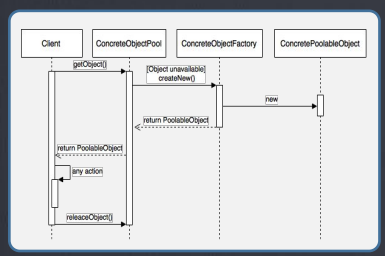
1. El cliente solicita la clonación de prototype mediante el método clone.
2. El prototype crea una nueva instancia de si misma mediante el operador new.
3. El prototype clona todos sus atributos a la nueva instancia.
4. Opcionalmente, el prototype puede clonar todos los objetos interno, lo que se conoce como Deep clone o clonación profunda.
5. El nuevo objeto clonado es retornado por el prototype.

Patrón Object Pool



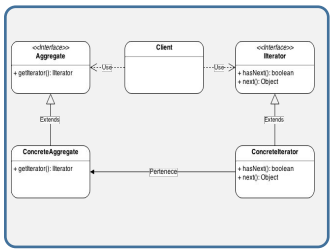
Tipo: Creacional

Este es un patrón muy utilizado cuando se requiere trabajar con una gran cantidad de objetos, los cuales son computacionalmente caros de crear, este patrón tiene una gran ventaja en escenarios donde nuestro programa requiere dichos objetos por un tiempo muy corto y que luego de su uso son desechados



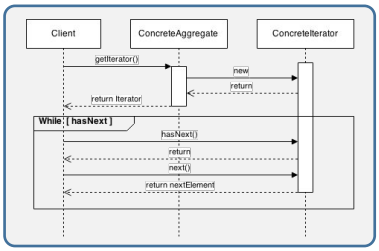
1. El cliente solicita un objeto al **ConcreteObjectPool**.
2. El **ConcreteObjectPool** valida si existen objetos disponibles, de no ser así solicitará la creación de un nuevo objeto al **ConcreteObjectFactory**.
3. El **ConcreteObjectFactory** creará un nuevo objeto de tipo **ConcretePoolableObject**.
4. El **ConcreteObjectPool** regresará el objeto al cliente.
5. El cliente realiza cualquier acción con el objeto.
6. El cliente regresa el objeto al **ConcreteObjectPool**.

Patrón Iterator



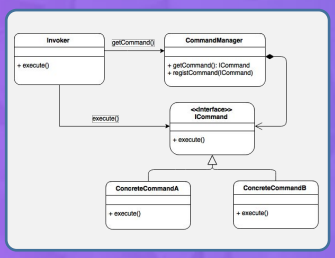
Tipo: Comportamiento

Permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma. Es especialmente útil cuando trabajamos con estructuras de datos complejas, ya que nos permite recorrer sus elementos mediante un Iterador, el Iterador es una interface que proporciona los métodos necesarios para recorrer los elementos de la estructura de datos.



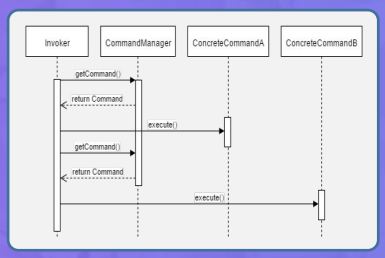
1. El **cliente** solicita al **ConcreteAggregate** la creación de un **iterador**.
2. El **ConcreteAggregate** crea un nuevo **Iterador**.
3. El **cliente**, para recorrer los elementos, entra en un ciclo hasta que no existen más elementos en el **iterador**, el método **hasNext** le indicará cuándo se ha llegado al final.
4. El cliente solicita el siguiente elemento al **iterador** mediante el método **next**.
5. Si existen más elementos nos regresamos al paso tres, esto se repite hasta finalizar el recorrido.

Patrón Command



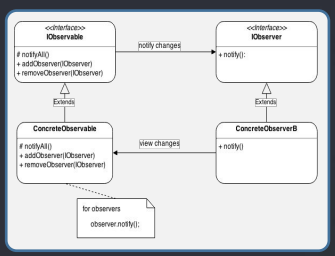
Tipo: Comportamiento

Permite ejecutar operaciones sin conocer los detalles de la implementación de la misma. Las operaciones son conocidas como comandos y cada operación es implementada como una clase independiente que realiza una acción muy concreta



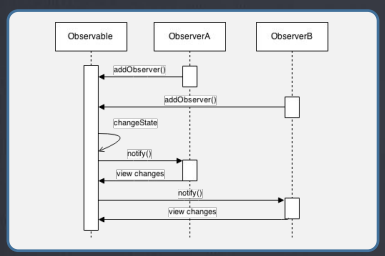
1. El invoker obtiene un Comando del CommandManager.
2. El invoker ejecuta el comando.
3. El invoker obtiene otro Comando del CommandManager.
4. El invoker ejecuta el comando.

Patrón Observer



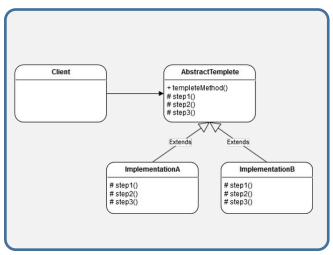
Tipo: Comportamiento

Permite observar los cambios producidos por un objeto, de esta forma, cada cambio que afecte el estado del objeto observado lanzará una notificación a los observadores; a esto se le conoce como Publicador-Suscriptor.



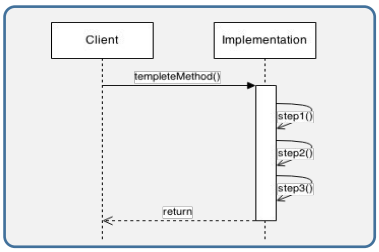
1. El **ObserverA** se registra con el objeto **Observable** para ser notificado de algún cambio.
2. El **ObserverB** se registra con el objeto **Observable** para ser notificado de algún cambio.
3. Ocurre algún cambio en el estado del **Observable**.
4. Todos los **Observers** son notificados con el cambio ocurrido.

Patrón Templete Method



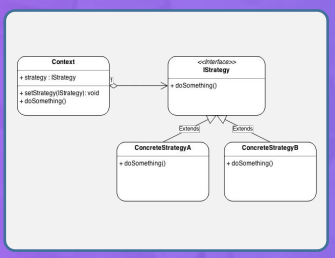
Tipo: Comportamiento

Se utiliza para implementar algoritmos que realizan los mismos pasos para llegar a una solución. Esto se logra implementando clases bases que definan un comportamiento predeterminado. Usualmente es creado un método para cada paso del algoritmo a implementar, de los cuales algunos serán implementados y otros permanecerán abstractos hasta su ejecución por parte de las subclases.



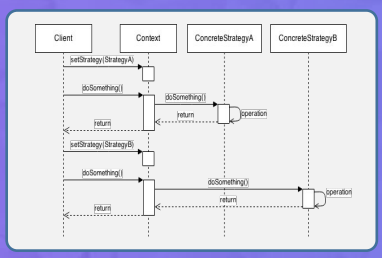
1. El **cliente** crea u obtiene una instancia de una implementación del **templete**.
2. El **cliente** ejecuta el método público **templeteMethod** del templete.
3. La implementación por default del método **templeteMehod** ejecuta en orden los métodos **step1**, **step2**, **step3, stepN**.
4. La **implementación** retorna un resultado.

Patrón Strategy



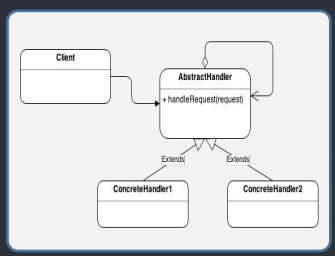
Tipo: Comportamiento

Permite establecer en tiempo de ejecución el rol de comportamiento de una clase. Stretagy se basa en el polimorfismo para implementar una serie de comportamientos que podrán ser intercambiados durante la ejecución del programa, logrando con esto que un objeto se pueda comportar de forma distinta según la estrategia establecida.



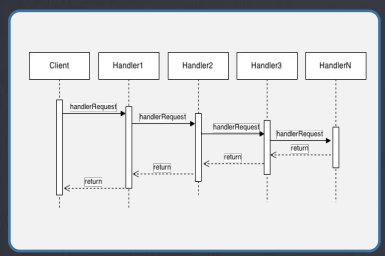
1. El **cliente** crea un nuevo contexto y establece la estrategia A.
2. El **cliente** ejecuta la operación **doSomething**.
3. **Context** a su vez delega esta responsabilidad a **ConcreteStrategyA**.
4. **ConcreteStrategyA** realiza la operación y regresa el resultado.
5. **Context** toma el resultado y lo regresa al **cliente**.
6. El **cliente** le cambia la **estrategia** a **Context** en tiempo de ejecución.
7. El **cliente** ejecuta nuevamente la operación **doSomething**.
8. **Context** a su vez delega esta responsabilidad a **ConcreteStrategyB**.
9. **ConcreteStrategyB** realiza la operación y regresa el resultado.
10. **Context** toma el resultado y lo regresa al **cliente**.

Patrón Chain of Responsability



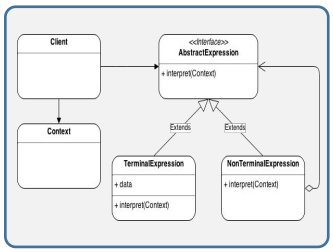
Tipo: Comportamiento

Permitiendo resolver problemas donde no estamos muy seguros de qué objeto deberá procesar una solicitud concreta; este patrón de diseño resuelve problemas fácilmente donde la herencia no puede. Apoyándose de una estructura en forma de cadena donde una secuencia de objetos tratan de atender una petición.



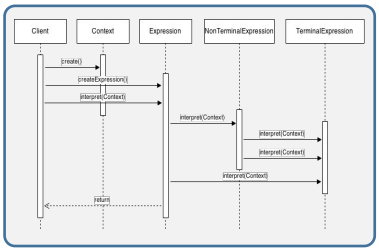
1. El cliente solicita el procesamiento de una solicitud a una cadena de responsabilidad.
2. El primer Handler intenta procesar el mensaje, sin embargo, no es capaz de procesarlo por alguna razón y envía el mensaje al siguiente de la cadena.
3. El segundo Handler intenta procesar el mensaje sin éxito, por lo que envía el mensaje al siguiente Handler de la cadena.
4. El tercer Handler también intenta procesar el mensaje sin éxito y envía el mensaje al siguiente Handler de la cadena.
5. El HandlerN (Algún handler de la secuencia) por fin es capaz de procesar el mensaje exitosamente y regresa una respuesta (opcional) de tal forma que la respuesta es replicada por todos los Handlers pasados hasta llegar al Cliente.

Patrón Interpreter



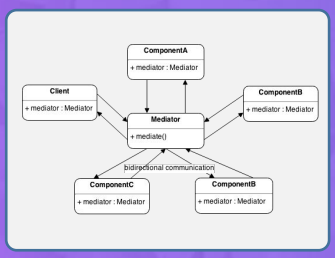
Tipo: Comportamiento

Es utilizado para evaluar un lenguaje definido como Expresiones, este patrón nos permite interpretar un lenguaje como Java, C#, SQL o incluso un lenguaje inventado por nosotros el cual tiene un significado; y darnos una respuesta tras evaluar dicho lenguaje



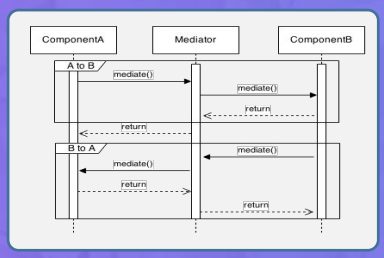
1. El **cliente** crea el **contexto** para la ejecución del interpreter.
2. El **cliente** crea u obtiene la **expresión** a evaluar.
3. El **cliente** solicita la interpretación de la **expresión** al **interpreter** y le envía el **contexto**.
4. La **Expresión** manda llamar a las **Expresiones No Terminales** que contiene.
5. La **Expresión No Terminal** manda llamar a todas las **Expresiones Terminales**.
6. La **Expresión Raíz** solicita la interpretación de una **Expresión Terminal**.
7. La expresión se evalúa por completo y se tiene un resultado de la interpretación de todas las expresiones terminales y no terminales.

Patrón Mediator



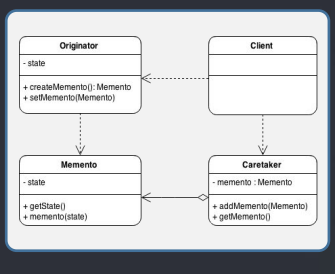
Tipo: Comportamiento

Se encarga de gestionar la forma en que un conjunto de clases se comunican entre sí, Mediator es especialmente útil cuando tenemos una gran cantidad de clases que se comunican de forma directa, ya que mediante la implementación de este patrón podemos crear una capa de comunicación bidireccional, en la cual las clases se pueden comunicar con el resto de ellas por medio de un objeto en común que funge como un mediador o intermediario.



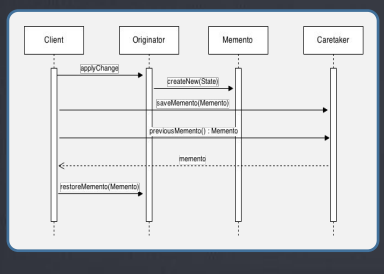
1. El **ComponenteA** desea comunicarse con el **ComponenteB** y le envía un mensaje por medio del **mediador**.
2. El **mediador** puede analizar el mensaje con fines de depuración, seguimiento o para canalizar el mensaje al destinatario.
3. El mensaje es entregado al destinatario y regresa una respuesta al **mediador**.
4. El **mediador** recibe la respuesta y la reedirecciona al **ComponenteA**.
5. De igual forma, el proceso se puede repetir del **ComponentB** al **ComponentA** repitiendo los pasos anteriores logrando una comunicación bidireccional.

Patrón Memento



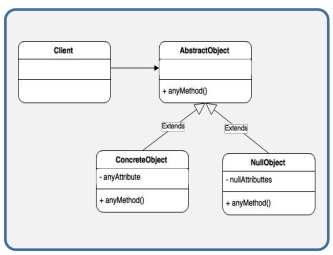
Tipo: Comportamiento

Permite capturar el estado de un objeto en un momento determinado con la finalidad de regresar a este estado en cualquier momento. Este patrón es utilizado cuando tenemos objetos que cambian en el tiempo y por alguna razón necesitamos restaurar su estado en un momento determinado.



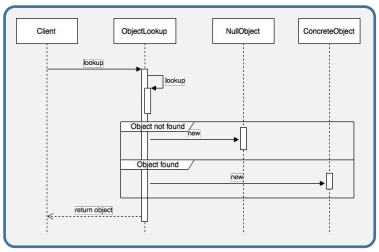
1. El **Cliente** aplica un cambio sobre el **Originator**.
2. El **Originator** crea un nuevo **Memento** que representa su estado actual.
3. El **Cliente** guardar el **Memento** en el **Caretaker** para posteriormente poder cambiar entre los estados del **Originator**.
4. Después de un tiempo, el **Cliente** solicita al **Caretaker** el estado previo del **Originator**.
5. El **Cliente** restaura el estado del **Originator** mediante el **Memento** obtenido del **Caretaker**.

Patrón Null Object



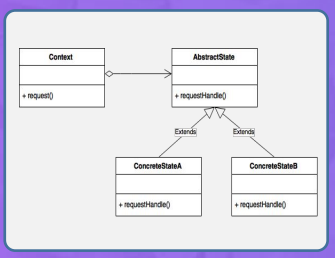
Tipo: Comportamiento

Evita los valores nulos que puedan originar error en tiempo de ejecución. Básicamente lo que este patrón propone es utilizar instancias que implementen la interface requerida pero con un cuerpo vacío en lugar de regresar un valor null.



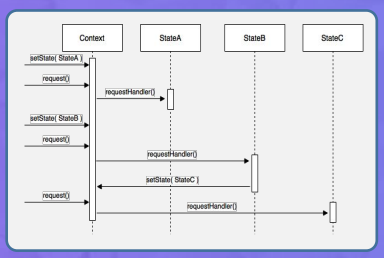
1. El **cliente** intenta buscar un objeto determinado.
2. El **ObjectLookup** busca si el objeto solicitado existe.
a. Si el **objeto** solicitado no existe entonces regresa una instancia de **NullObject**.
3. Por otra parte, si el **objeto** es localizado entonces se regresa una instancia del **ConcreteObject**.
4. Se regresa al **cliente** cualquiera de las dos instancias anteriores, sin embargo, éste nunca obtendrá una referencia nula en caso de no encontrarse.

Patrón State



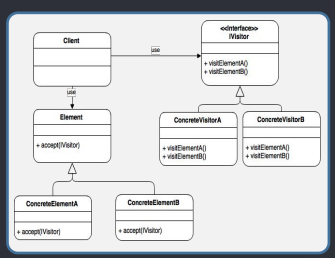
Tipo: Comportamiento

Permite modificar su comportamiento dependiendo del estado en el que se encuentra la aplicación. Para lograr esto es necesario crear una serie de clases que representarán los distintos estados por los que puede pasar la aplicación; es decir, se requiere de una clase por cada estado por el que la aplicación pueda pasar.



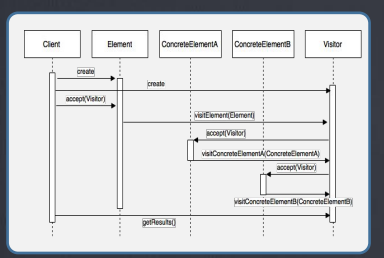
1. Se establece un **estado** por default al **Context**, el cual es **StateA**.
2. Se ejecuta la operación request sobre el **Context**, la cual delega la ejecución al estado actual (**StateA**).
3. El **Context** cambia del **estado A** al **estado B**.
4. Se ejecuta nuevamente la operación request sobre el **Context** que delega la ejecución al estado actual (**StateB**).
5. La ejecución del **StateB** da como resultado un cambio de estado al **StateC**.
6. Se ejecuta nuevamente la operación request sobre el **Context** que delega la ejecución al estado actual (**StateC**).

Patrón Visitor



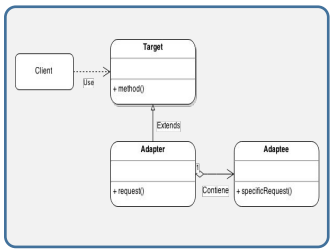
Tipo: Comportamiento

Separar la lógica u operaciones que se pueden realizar sobre una estructura compleja. En ocasiones nos podemos encontrar con estructuras de datos que requieren realizar operaciones sobre ella, pero estas operaciones pueden ser muy variadas e incluso se pueden desarrollar nuevas a medida que la aplicación crece.



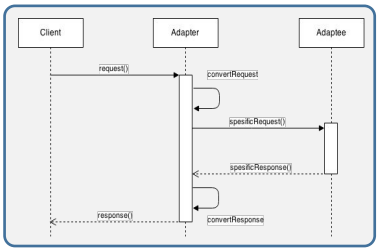
1. El **cliente** crea la estructura (**Element**).
2. El **cliente** crea la instancia del **Visitante** a utilizar sobre la estructura.
3. El **cliente** ejecuta el método **accept** de la estructura.
4. El **Element** le dice al **Visitante** con que método lo debe procesar cada objeto de la estructura.
5. El **Visitante** analiza al **Element** mediante su método **visitElement** y repite el proceso de ejecutar el método **accept** sobre los hijos del **Element**.
6. El **ConcreteElementA** le indica al **Visitante** con qué método debe procesarlo.
7. La **visitante** continúa con los demás hijos de **Element** y esta vez ejecuta el método **accept** sobre el **ConcreteElementB**.
8. El **ConcrteElementB** le indica al **Visitante** con qué método debe procesarlo, el cual es **visitElementB**.
9. Finalmente el resultado se obtiene del método **getResults**.

Patrón Adapter



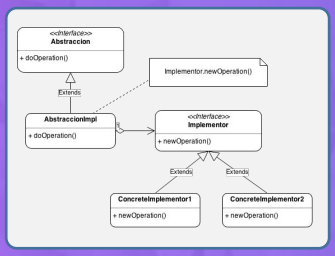
Tipo: Estructural

es utilizado cuando tenemos interfaces de software incompatibles, las cuales a pesar de su incompatibilidad tiene una funcionalidad similar. Este patrón es implementado cuando se desea homogeneizar la forma de trabajar con estas interfaces incompatibles, para lo cual se crea una clase intermedia que funciona como un adaptador



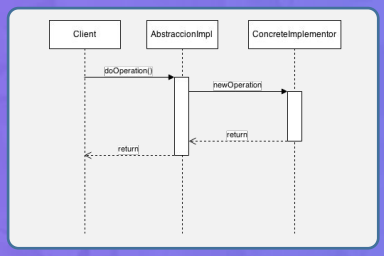
1. El **Client** invoca al **Adapter** con parámetros genéricos.
2. El **Adapter** convierte los parámetros genéricos en parámetros específicos del **Adaptee**.
3. El **Adapter** invoca al Adaptee.
4. El **Adaptee** responde.
5. El **Adapter** convierte la respuesta del **Adaptee** a una respuesta genérica para el **Client**.
6. El **Adapter** responde al **Client** con una respuesta genérica.

Patrón Bridge



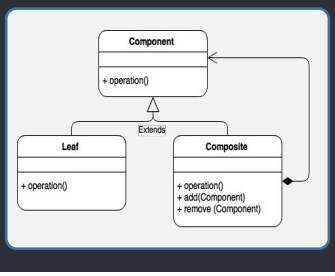
Tipo: Estructural

Desacopla una abstracción de su implementación, de manera que las dos puedan ser modificadas por separado sin necesidad de modificar la otra; dicho de otra manera, se desacopla una abstracción de su implementación para que puedan variar independientemente



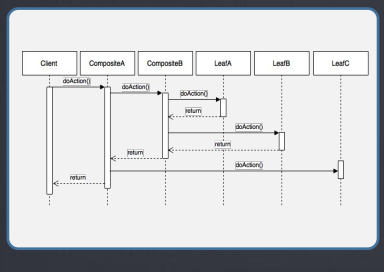
1. El cliente ejecuta una operación de AbstraccionImpl.
2. AbstraccionImpl replica la petición a ConcreteImplementor, en este paso el AbstraccionImpl pudiera realizar una conversión de los parámetros para ejecutar al ConcreteImplementor.
3. ConcreteImplementor regresa los resultados al AbstraccionImpl.
4. El AbstraccionImpl convierte los resultados del ConcreteImplementor para ser devueltos al cliente.

Patrón Composite



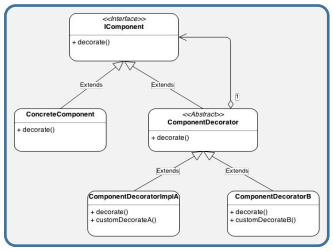
Tipo: Estructural

Sirve para construir estructuras complejas partiendo de otras mucho más simples; dicho de otra manera, podemos crear estructuras compuestas que están conformadas por otras estructuras más pequeñas.



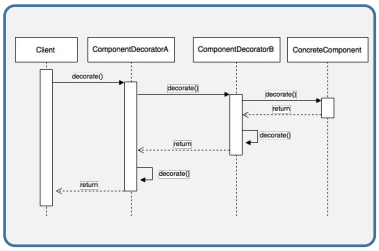
1. El cliente realizar una acción sobre el **CompositeA**.
2. **CompositeA** a su vez realiza una acción sobre **CompositaB**.
3. **CompositeB** realiza una acción sobre **LeafA** y **LeafB** y el resultado es devuelto a **CompositeA**.
4. **CompositeA** propaga la acción sobre **LeafC**, el cual le regresa un resultado.
5. **CompositeA** obtiene un resultado final tras la evaluación de toda la estructura y el cliente obtiene un resultado.

Patrón Decorator



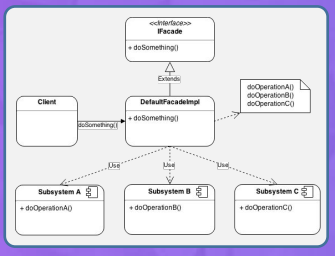
Tipo: Estructural

Permite al usuario añadir nuevas funcionalidades a un objeto existente sin alterar su estructura, mediante la adición de nuevas clases que envuelven a la anterior dándole funcionamiento extra. Este patrón está diseñado para solucionar problemas donde la jerarquía con subclasificación no puede ser aplicada



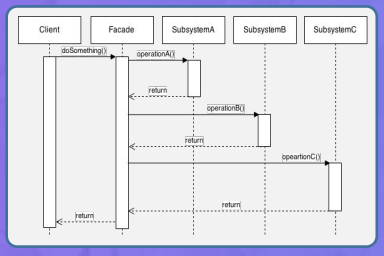
1. El Cliente realiza una operación sobre el **DecoratorA**.
2. El **DecoratorA** realiza la misma operación sobre **DecoradorB**.
3. El **decoradorB** realiza una acción sobre **ConcreteComponente**.
4. El **DecoradorB** ejecuta una operación de decoración.
5. El **DecoradorA** ejecuta una operación de decoración.
6. El **Cliente** recibe como resultado un objeto decorado por todos los Decoradores, los cuales encapsularon el **Component** en varias capas.

Patrón Facade



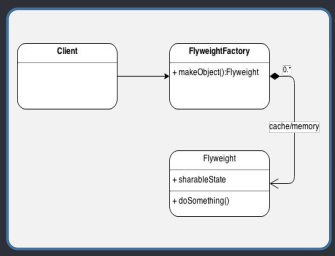
Tipo: Estructural

Oculta la complejidad de interactuar con un conjunto de subsistemas proporcionando una interface de alto nivel, la cual se encarga de realizar la comunicación con todos los subsistemas necesarios, encapsulando la complejidad de interactuar individualmente con cada subsistema.



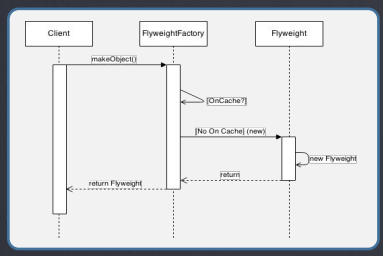
1. El cliente invoca una operación de la fachada.
2. La fachada se comunica con el SubsystemA para realizar una operación.
3. La fachada se comunica con el SubsystemB para realizar una operación.
4. La fachada se comunica con el SubsystemC para realizar una operación.
5. La fachada responde al cliente con el resultado de la operación.

Patrón Flyweight



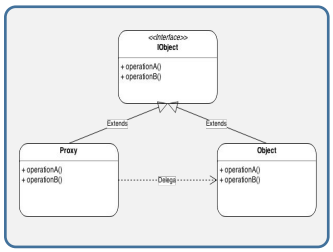
Tipo: Estructural

Centra su atención en la construcción de objetos ligeros, mediante la abstracción de las partes reutilizables que pueden ser compartidas con otros objetos, esto con el fin de que en lugar de crear objetos cada vez que sea requerido, podamos reutilizar objetos creados por otras instancias



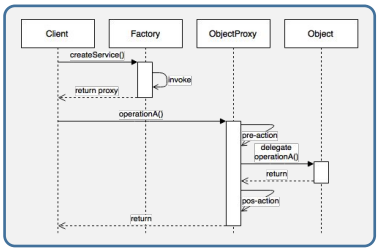
1. El cliente solicita al **Factory** la creación de un objeto Flyweight.
2. El **Factory** antes de crear el objeto, valida si ya existe un objeto idéntico al que se le está solicitando. De ser así, regresa el objeto existente; de no existir, crea el nuevo objeto y lo almacena en caché para ser reutilizado más adelante.
3. El objeto **Flyweight** se crea o es tomado del caché y es devuelto al cliente.

Patrón Proxy



Tipo: Estructural

Permite la mediación entre un objeto y otro. Se dice mediación porque este patrón nos permite realizar ciertas acciones antes y después de realizar la acción deseada por el usuario



1. El **cliente** solicita al **Factory** un Objeto.
2. El **Factory** crea un **Proxy** que encapsule al **Object**.
3. El **cliente** ejecuta el **Proxy** creado por el **Factory**.
4. El **Proxy** realiza una o varias acciones previas a la ejecución del **Object**.
5. El **Proxy** delega la ejecución al **Object**.
6. El **Proxy** realiza una o varias acciones después de la ejecución del **Object**.
7. El **Proxy** regresa un resultado.