Patrones creacionales

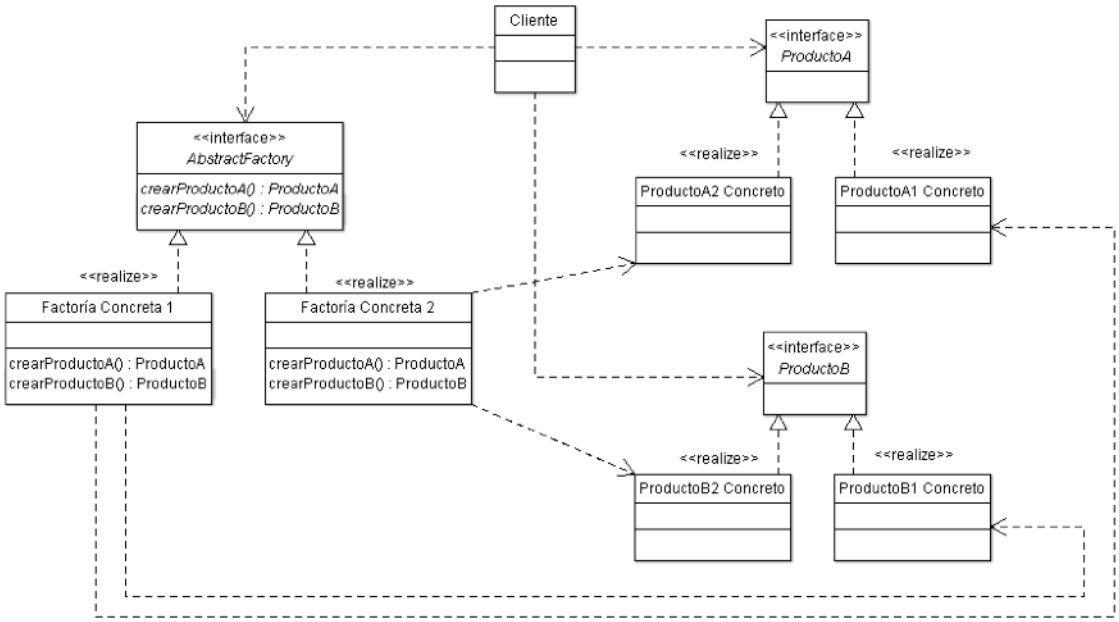
Abstract Factory

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

Contexto: Debemos crear diferentes objetos, todos pertenecientes a la misma familia.

Ejemplo: Las bibliotecas para crear interfaces gráficas suelen utilizar este patrón y cada familia sería un sistema operativo distinto. Así pues, el usuario declara un Botón, pero de forma más interna lo que está creado es un BotonWindows o un BotonLinux, por ejemplo.

Está aconsejado cuando se prevé la inclusión de nuevas familias de productos, pero puede resultar contraproducente cuando se añaden nuevos productos o cambian los existentes, puesto que afectaría a todas las familias creadas.



Esto permite hacer programas independientemente de la base de datos que se use. Son modificaciones a hacer en el archivo de configuración únicamente.

La estructura típica del patrón **Abstract Factory** es la siguiente:

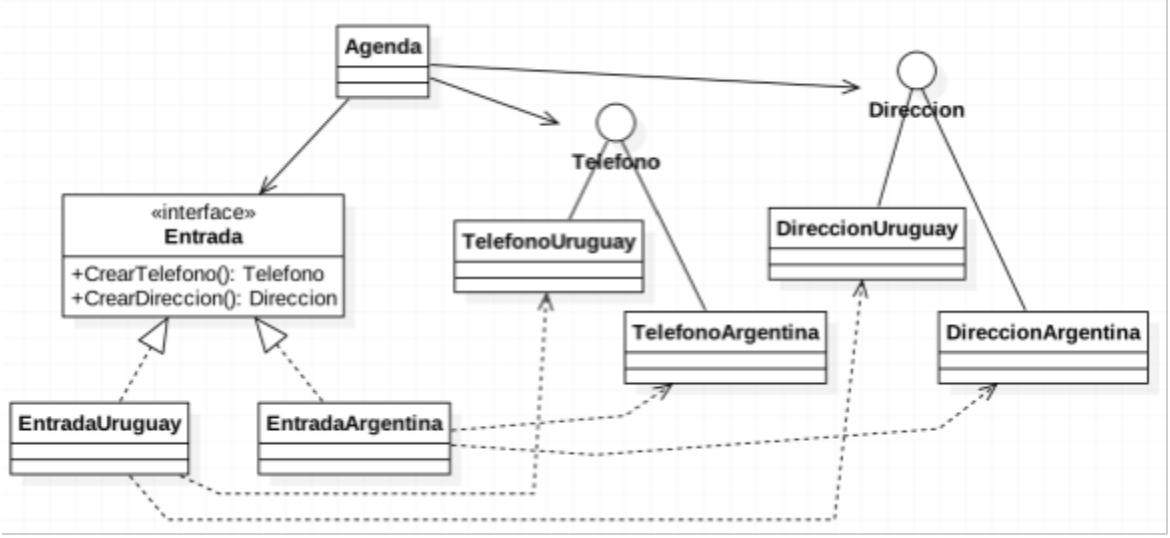
- **Cliente**: La clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría, es decir, Cliente lo que quiere es obtener una instancia de alguno de los productos (ProductoA, ProductoB).

- **AbstractFactory**: Es la definición de las interfaces de las factorías. Debe de proveer un método para la obtención de cada objeto que pueda crear. ("crearProductoA()" y "crearProductoB()")
- **Factorías Concretas**: Estas son las diferentes familias de productos. Provee de la instancia concreta de la que se encarga de crear. De esta forma podemos tener una factoría que cree los elementos gráficos para Windows y otra que los cree para Linux, pudiendo poner fácilmente (creando una nueva) otra que los cree para MacOS, por ejemplo.
- **Producto abstracto**: Definición de las interfaces para la familia de productos *genéricos*. En el diagrama son "ProductoA" y "ProductoB". En un ejemplo de interfaces gráficas podrían ser todos los elementos: Botón, Ventana, Cuadro de Texto, Combo... El cliente trabajará directamente sobre esta interfaz, que será implementada por los diferentes productos concretos.
- **Producto concreto**: Implementación de los diferentes productos. Podría ser por ejemplo "BotónWindows" y "BotónLinux". Como ambos implementan "Botón" el cliente no sabrá si está en Windows o Linux, puesto que trabajará directamente sobre la superclase o interfaz.

Ejemplo concreto implementación:

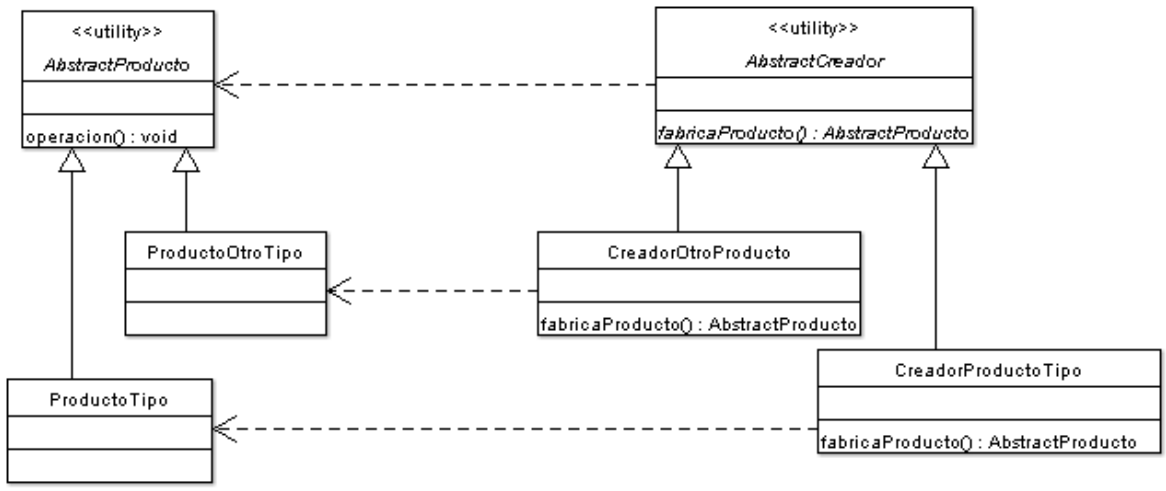
Para el mantenimiento de una Agenda, se tiene información como los números de teléfono y direcciones, por lo que se cuenta con clases para el manejo de la Direccion y el Telefono. Tanto los números de teléfonos como las direcciones, tienen un comportamiento particular para cada país, por lo que se quiere tener diferentes implementaciones según el país de los datos a almacenar en la agenda.

Se quiere poder agregar nuevas implementaciones para los diferentes países con el menor impacto posible en la solución, agregando una nueva entrada en la agenda manteniendo la coherencia entre la dirección y teléfono perteneciente a un mismo país.


Agenda
Direccion
Telefono
«interface»
Entrada
+CrearTelefono(): Telefono
+CrearDireccion(): Direccion
TelefonoUruguay
DireccionUruguay
TelefonoArgentina
DireccionArgentina
EntradaUruguay
EntradaArgentina

Factory Method

Definir una interfaz para crear un objeto, permitiendo a las subclases decidir qué clase instanciar.



Este patrón consiste en utilizar una clase constructora abstracta con unos cuantos métodos definidos y otro(s) abstracto(s): el dedicado a la construcción de objetos del subtipo de un tipo determinado.

Es una simplificación de Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos.

Las clases principales de este patrón son:

- **Creador**: Necesita crear instancias de productos, pero el tipo concreto del producto no debe ser forzado en las subclases del creador, porque las posibles subclases del creador deben poder especificar subclases del producto a utilizar.
- **Product**

La solución para esto es hacer un método abstracto (el método fábrica) que se define en el creador.

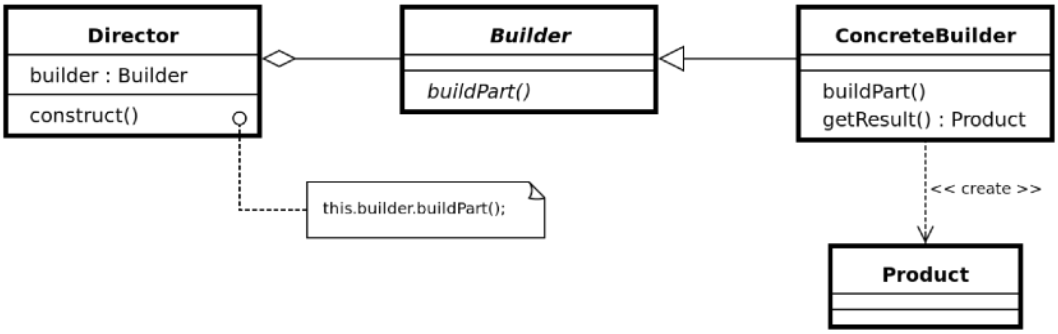
Este método abstracto se define para que devuelva el producto. Las subclases del creador pueden sobreescribir este método para devolver subclases apropiadas del producto.

Aplicación:

- Se utiliza cuando una clase no puede anticipar qué clase debe crear.
- Cuando una clase quiere que sus subclases sean las responsables de crear la instancia.

Builder

Separar la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.



Las clases principales de este patrón son:

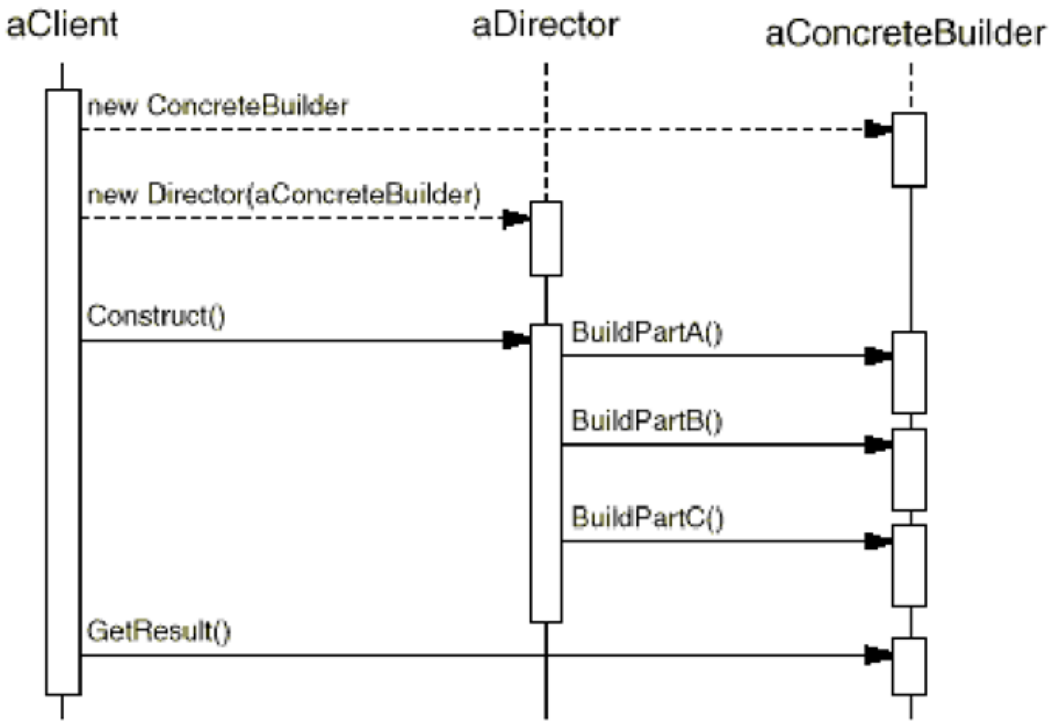
- **Producto**: Representa el objeto complejo que se quiere construir.
- **Builder**: Especifica una interface para crear las partes del objeto producto.
- **ConcreteBuilder**: Construye y ensambla las partes del producto mediante la implementación de la interface builder. Define y controla la representación que crea. Provee una interface para obtener el producto.
- **Director**: Construye un objeto utilizando la interface del builder.

Ventajas:

- Reduce el acoplamiento.
- Permite variar la representación interna de estructuras complejas, respetando la interfaz común de la clase Builder.
- Se independiza el código de construcción de la representación. Las clases concretas que tratan las representaciones internas no forman parte de la interfaz del Builder.
- Cada ConcreteBuilder tiene el código específico para crear y modificar una estructura interna concreta.
- Permite un mayor control en el proceso de creación del objeto. El Director controla la creación paso a paso, solo cuando el Builder ha terminado de construir el objeto lo recupera el Director.

La utilización de este patrón, por lo tanto, consistirá en lo siguiente:

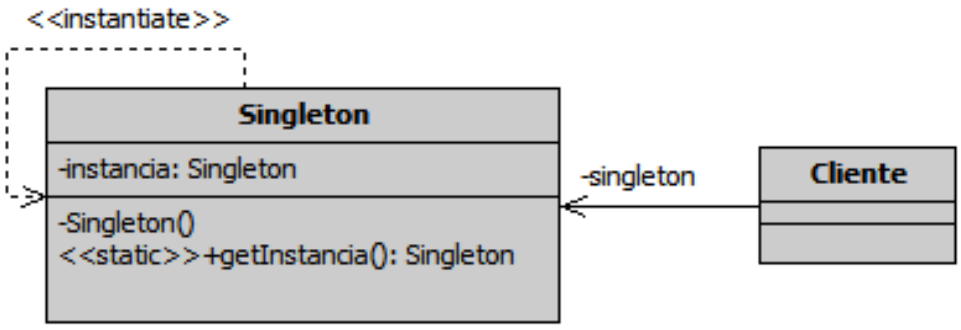
1. Instanciar un nuevo director pasándole como parámetro el constructor concreto que se encargará de construir las piezas.
2. Indicarle al constructor que construya el objeto.
3. Recuperar el objeto del director (que, a su vez, lo recupera del constructor concreto).



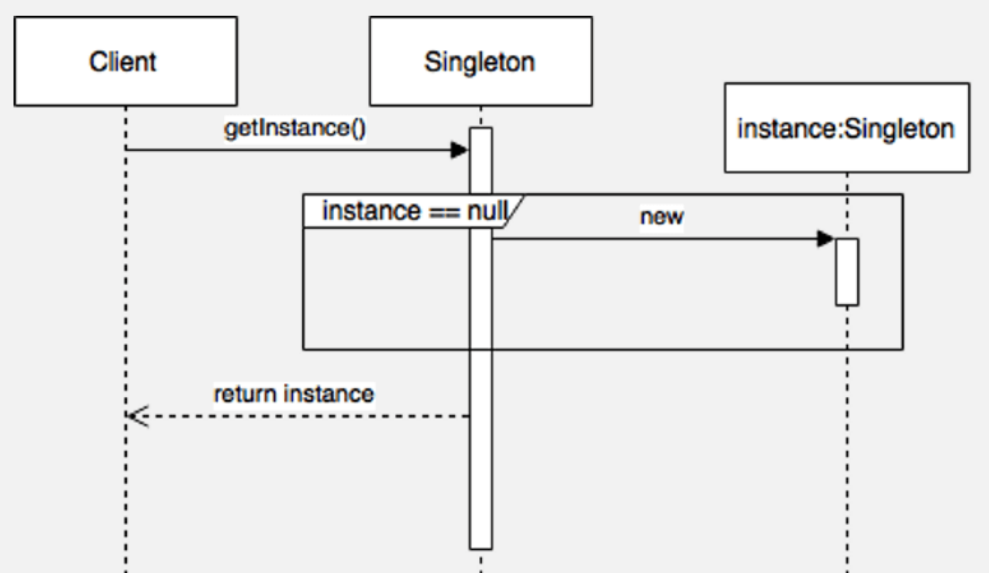
Singleton

Consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

Estructura: Clase Singleton con constructor privado, atributo estático de la clase y un método GetInstance():Singleton. La idea es que siempre se llame al método GetInstance() en vez de al constructor, y se obtenga la instancia creada del objeto.



El diagrama de secuencia quedaría así



Patrones estructurales

Adapter

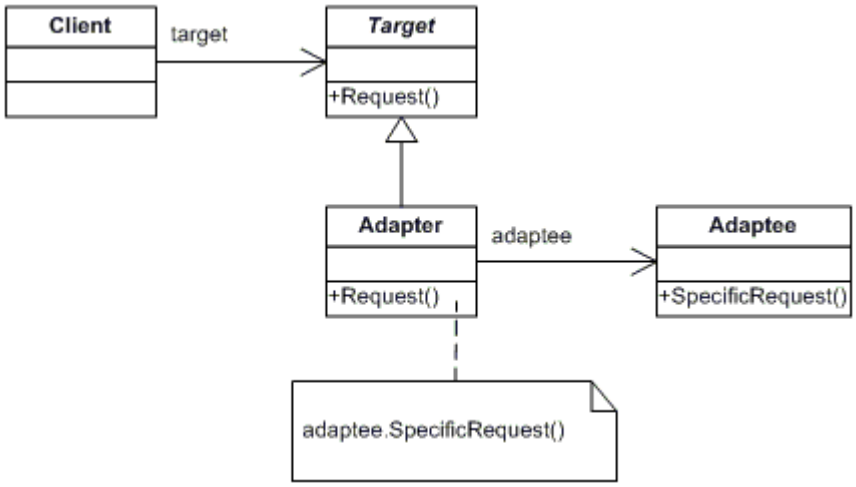
Convertir la interfaz de una clase en otra que un cliente espera.

Se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

Adapter permite a las clases trabajar juntas, lo que de otra manera no hubiese sido posible dado sus interfaces incompatibles.

Aplicación:

- Se desea utilizar una clase, pero su interfaz no se iguala a lo necesitado.
- Cuando se desea crear una clase re-usable que coopera con clases no relacionadas, es decir, las clases no tienen necesariamente interfaces compatibles.



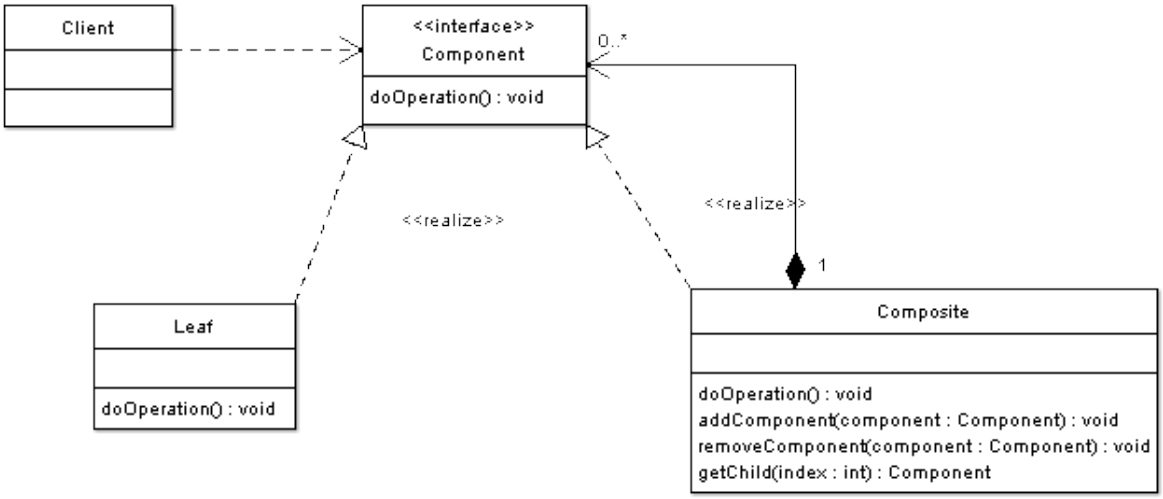
Las clases principales de este patrón son:

- **Target** define la interfaz específica del dominio que *Client* usa.
- **Client** colabora con la conformación de objetos para la interfaz *Target*.
- **Adaptee** define una interfaz existente que necesita adaptarse
- **Adapter** adapta la interfaz de *Adaptee* a la interfaz *Target*

Composite

Componer objetos en estructuras de árbol, permitiendo a los clientes tratar objetos individuales y compuestos en forma uniforme.

Sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a la estructura en forma de árbol.



Ejemplo de problema que soluciona:

Imaginemos que necesitamos crear una serie de clases para guardar información acerca de una serie de figuras que serán círculos, cuadrados y triángulos. Además, necesitamos poder tratar también grupos de imágenes porque nuestro programa permite seleccionar varias de estas figuras a la vez para moverlas por la pantalla.

En principio tenemos las clases *Círculo*, *Cuadrado* y *Triángulo*, que heredarán de una clase padre que podríamos llamar *Figura* e implementarán todas la operación **pintar()**. En cuanto a los grupos de *Figuras* podríamos caer en la tentación de crear una clase particular separada de las anteriores llamada *GrupoDeImágenes*, también con un método **pintar()**.

Problema:

Esta idea de separar en clases privadas componentes (figuras) y contenedores (grupos) tiene el problema de que, para cada uno de las dos clases, el método **pintar()** tendrá una implementación diferente, aumentando la complejidad del sistema.

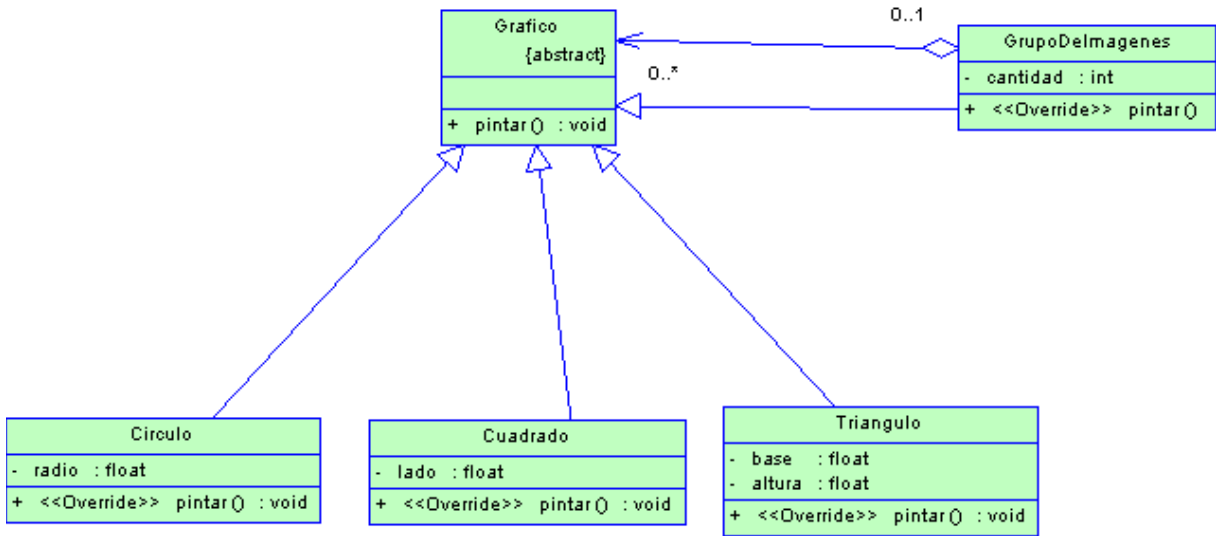
Solución:

A la clase Figura la llamaríamos Gráfico y de ella extenderían tanto Círculo, Cuadrado y Triángulo, como GrupoDeImágenes. Además, esta última tendría una relación todo-parte de multiplicidad * con Gráfico:

un GrupoDeImágenes contendría varios Gráficos, ya fuesen estos Cuadrados, Triángulos, u otras clases GrupoDeImágenes.

Así, es posible definir a un grupo de imágenes recursivamente. Por ejemplo, un objeto cuya clase es GrupoDeImágenes podría contener un Cuadrado, un Triángulo y otro GrupoDeImágenes, este grupo de imágenes podría contener un Círculo y un Cuadrado.

Posteriormente, a este último grupo se le podría añadir otro GrupoDeImágenes, generando una estructura de composición recursiva en árbol, por medio de muy poca codificación y un diagrama sencillo y claro.



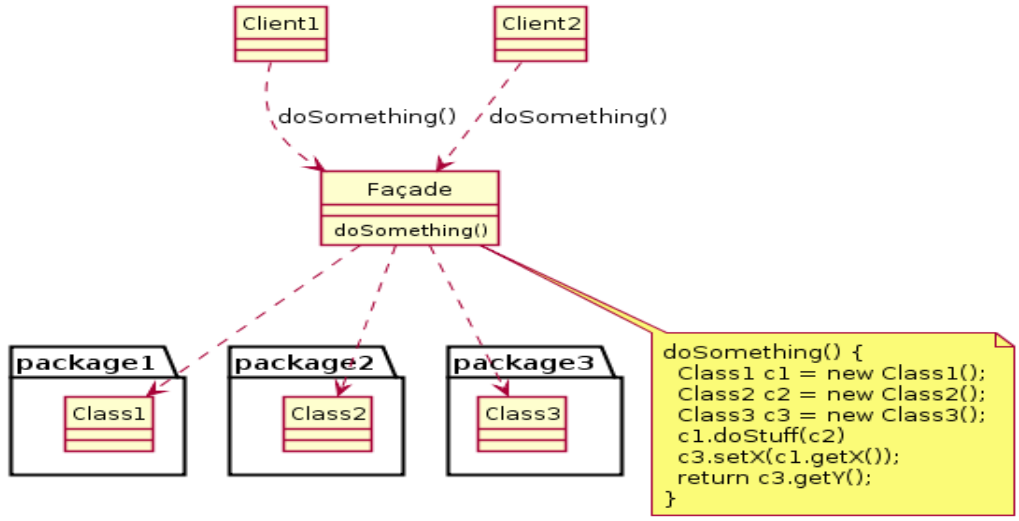
Aplicación:

- Cuando se quiere representar una jerarquía parcial.

Facade

Proveer una interfaz unificada para un conjunto de interfaces en un subsistema. La fachada define una interface que hace fácil de usar los subsistemas.

Estructurar un sistema en subsistemas ayuda a reducir la complejidad y el número de dependencias.



Se aplicará el patrón fachada cuando se necesite proporcionar una interfaz simple para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel. Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable.

<u>Usos comunes:</u>

Problema 1: Un cliente necesita acceder a parte de la funcionalidad de un sistema más complejo.

Solución 1: Definir una interfaz que permita acceder solamente a esa funcionalidad.

Problema 2: Existen grupos de tareas muy frecuentes para las que se puede crear código más sencillo y legible.

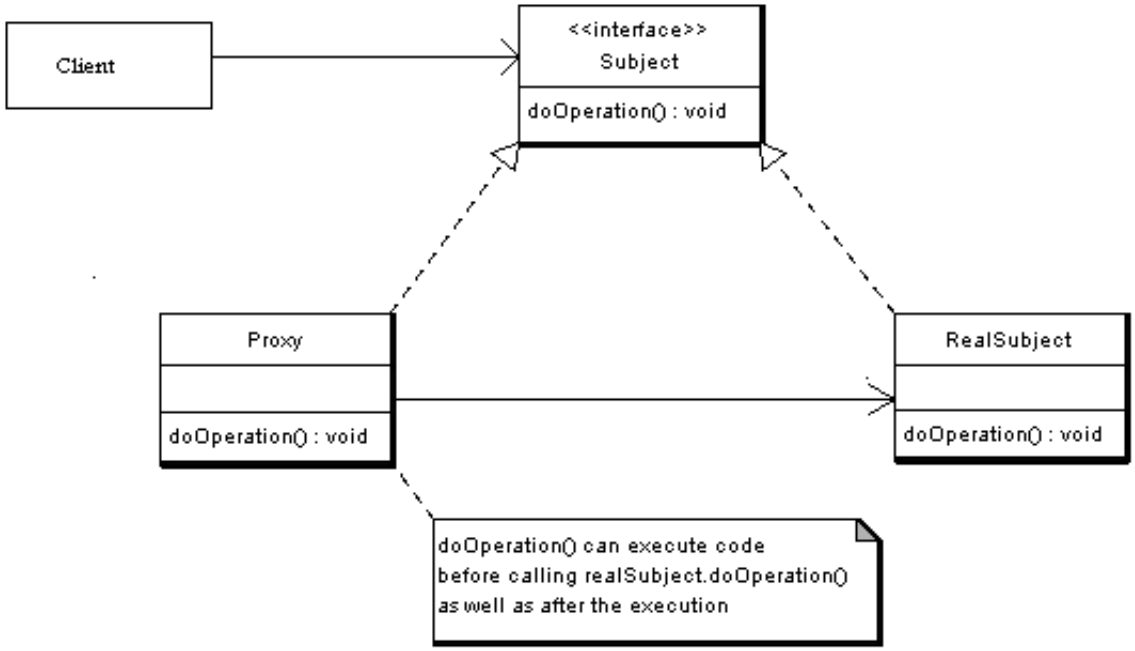
Solución 2: Definir funcionalidad que agrupe estas tareas en funciones o métodos sencillos y claros.

Problema 3: Una biblioteca es difícilmente legible.

Solución 3: Crear un intermediario más legible.

Proxy

Proveer un sustituto o un marcador de posición (placeholder) para un objeto de forma de controlar el acceso al mismo.



Ejemplo de problema que soluciona:

Consideramos un editor que puede incluir objetos gráficos dentro del documento. Se requiere que la apertura del documento sea rápida, mientras que la creación de algunos objetos (imágenes de gran tamaño) es cara.

En este caso no es necesario crear todos los objetos con imágenes nada más abrir el documento porque no todos los objetos son visibles.

Interesa por tanto retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario (por ejemplo, no abrir las imágenes de un documento hasta que no son visibles).

Solución:

La solución que se plantea para ello es la de cargar las imágenes bajo demanda. Esto se hace usando un objeto proxy. Dicho objeto se comporta como una imagen normal y es el responsable de cargar la imagen bajo demanda.

Aplicación:

El patrón *proxy* se usa cuando se necesita una referencia a un objeto más flexible o sofisticada que un puntero. Dependiendo de la función que se desea realizar con dicha referencia podemos distinguir diferentes tipos de proxies:

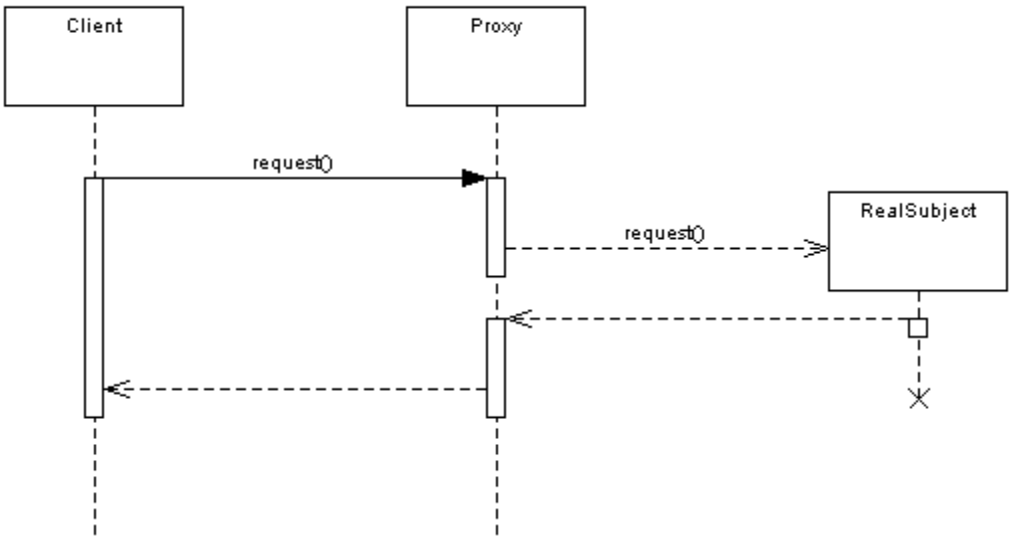
- Proxy remoto: Es un representante local para un objeto en diferentes espacios o máquinas.
- Proxy virtual: Es un representante más “liviano” que el objeto original, el cual es creado a demanda.
- Proxy de protección: Es un representante que controla el acceso al objeto original. Es útil cuando un objeto debería tener distintos permisos de acceso.
- Referencia inteligente: Es un puntero inteligente que realiza operaciones extra, como:
    - Contar cuantos lo apuntan.
    - Crear una instancia del objeto real en memoria cuando es solicitado por primera vez.
    - Verificar el acceso concurrente a un objeto.

Las clases principales de este patrón son:

Proxy: mantiene una referencia al objeto real (Sujeto Real) y proporciona una interfaz idéntica al sujeto. Además, controla el acceso a dicho objeto real y puede ser el responsable de su creación y borrado. También tiene otras responsabilidades que dependen del tipo de proxy.

Sujeto: define una interfaz común para el proxy y el objeto real (Sujeto Real), de tal modo que se puedan usar de manera indistinta.

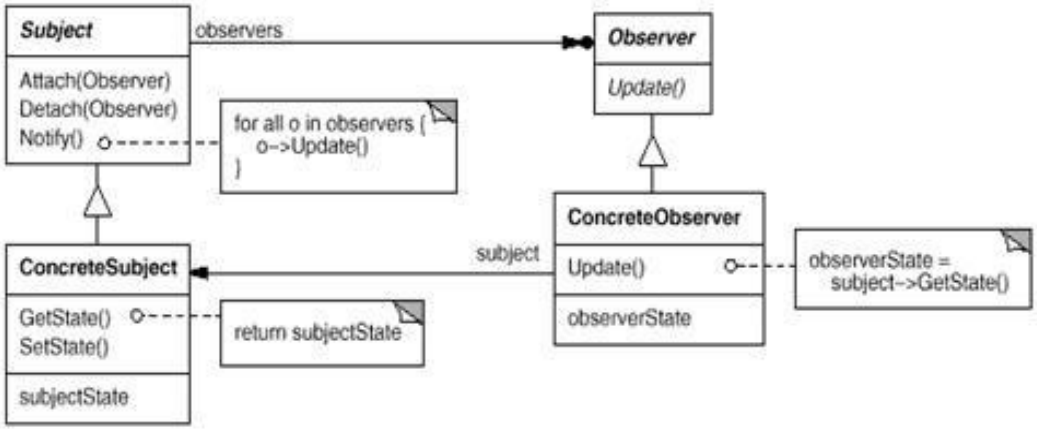
Sujeto Real: clase del objeto real que el proxy representa.



Patrones de comportamiento

Observer

Definir una dependencia de uno a muchos de forma que cuando un objeto cambia su estado, todos sus dependientes son notificados y pueden actualizarse automáticamente.



Aplicación:

- Uno o varios objetos necesitan ser notificados de los cambios de otro objeto concreto.
- Las notificaciones se realizan de forma dinámica en tiempo de ejecución.
- El objeto observable no necesita saber quién lo observa exactamente, sino que es un observador, por lo que se consigue un mejor desacople.

Las clases principales de este patrón son:

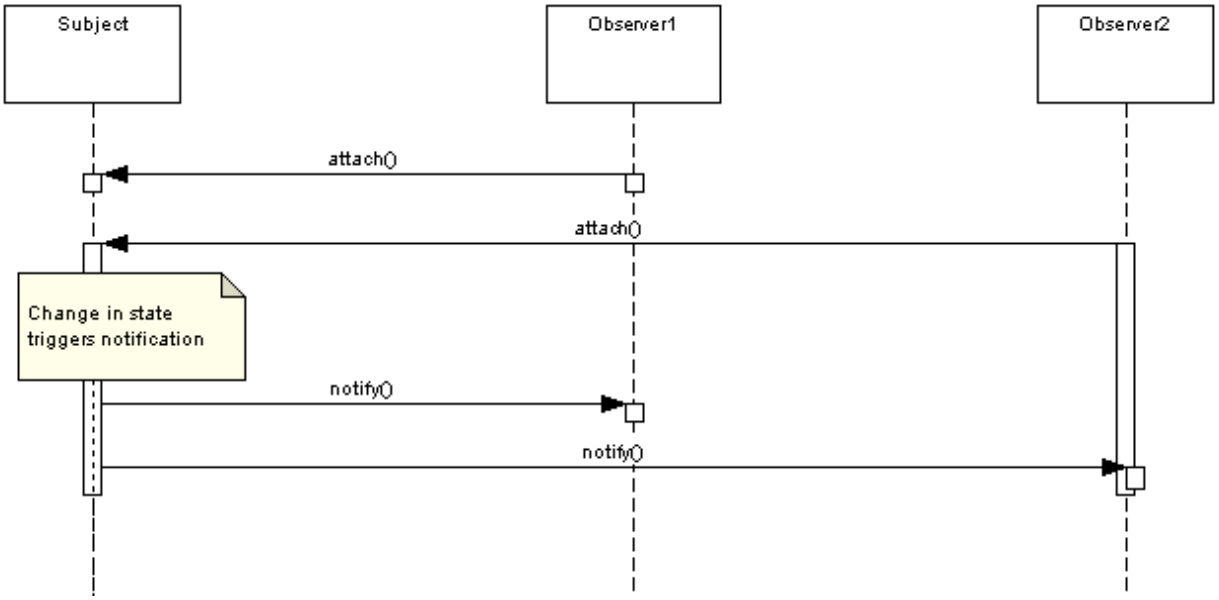
Sujeto (Subject):
El sujeto proporciona una interfaz para agregar (attach) y eliminar (detach) observadores. El Sujeto conoce a todos sus observadores.

Observador (Observer):
Define el método que usa el sujeto para notificar cambios en su estado (update/notify).

Sujeto Concreto (ConcreteSubject):
Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen por qué ser elementos de la misma jerarquía.

Observador Concreto (ConcreteObserver):
Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

Un posible diagrama de secuencia seria



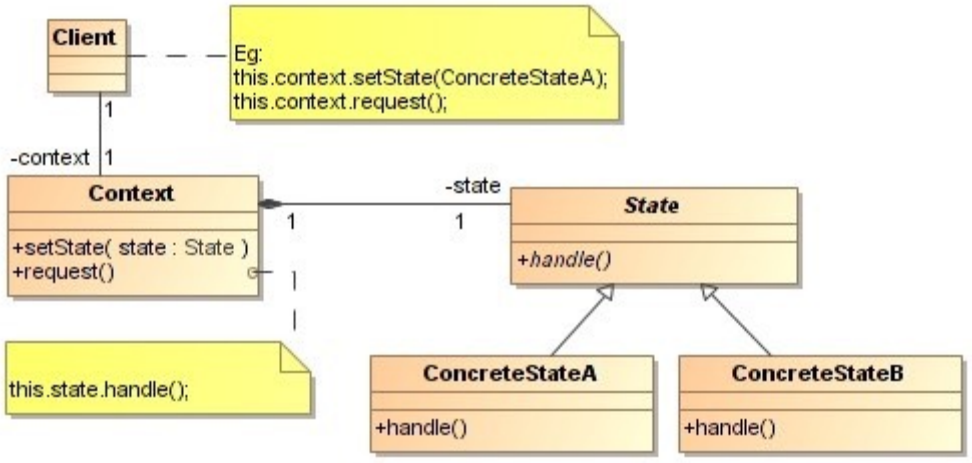
State

Permitir a un objeto alterar su comportamiento cuando su estado interno cambia. El objeto aparenta cambiar su clase.

Está apuntado a cuando un determinado objeto tiene diferentes estados y también distintas responsabilidades según el estado en que se encuentre en determinado instante.

Por ejemplo: una alarma puede tener diferentes estados, como desactivada, activada, en configuración. Definimos una interfaz Estado_Alarma, y luego definimos los diferentes estados.

Si no usara el patrón State tendría que hacer una clase que tenga un atributo de tipo int llamado Estado y voy cambiando los valores a mano (1 = Activada, 2 = Desactivada, 3 = EnConfiguracion, etc.). Podría tener un swich grande y dependiendo del valor de mi estado, lo que hace el método. Esto es muy ineficiente.



El patrón no indica exactamente dónde definir las transiciones de un estado a otro. Existen dos formas de solucionar esto: Una es definiendo estas transiciones dentro de la clase contexto, la otra es definiendo estas transiciones en las subclases de State.

Ventajas:

- Localiza y divide conductas de los distintos estados, poniendo todo lo relacionado a un estado en un solo lugar.
- Es una buena alternativa frente a opciones case o if que se repiten en el contexto.
- Es muy sencillo agregar nuevos estados.
- La transición de los estados es explícita y atómica desde el punto de vista del contexto.

Desventajas:

- Aumenta el número de clases y la solución es menos compacta que con una sola clase.
- ¿Quién realiza las transiciones? El patrón no lo define. Si son simples, pueden ser definidas en el contexto. Definidas en los estados son más flexibles. Otra forma es utilizando tablas para guardar las transiciones.

Maneras de crear los estados:

- Cada vez que se necesitan: Cuando no se conocen los estados ni cuando estos cambian.
- Todos al crear el contexto: Cuando hay muchos y rápidos cambios. El contexto debe tener referencias a todos los estados posibles.

Un posible pseudo código seria:
Clase main:

```
State state = new ConcreteStateA();
Context context = new Context();
context.setState(state);
context.request();
```

Clase Context:

```
Public void request() {
     state.handle();
}
```

Clase State:

```
Public interface State
{
     Void handle();
}
```

Clase ConcreteStateA:

```
Public class ConcreteStateA implements State
{
     Public void handle() {
     }
}
```

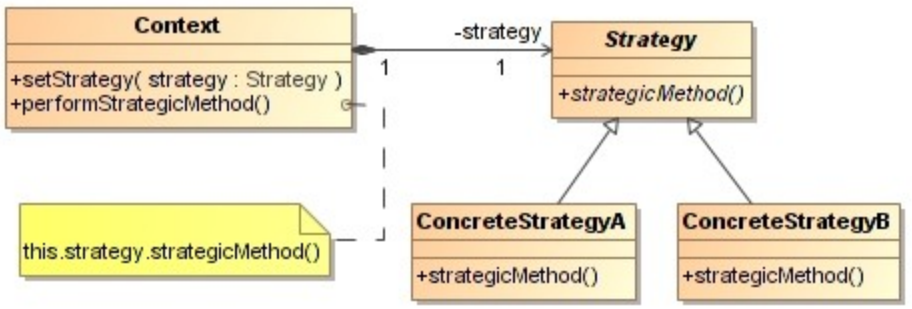
Clase ConcreteStateB:

```
Public class ConcreteStateB implements State
{
     Public void handle() {
     }
}
```

Strategy

Definir una familia de algoritmos, encapsular cada uno en una clase y usarlos de forma intercambiable. Permite intercambiar el algoritmo independientemente de los clientes que lo usen.

Se necesitan diferentes variantes de un algoritmo. Una clase define variantes de su comportamiento.



Ejemplo concreto de aplicación:

Supongamos un protagonista de un videojuego en el cual manejamos a un soldado que puede portar y utilizar varias armas distintas. La clase (o clases) que representan a nuestro soldado no deberían de preocuparse de los detalles de las armas que porta: debería bastar, por ejemplo, con un método de interfaz “atacar” que dispare el arma actual y otro método “recargar” que inserte munición en ésta (si se diera el caso).

En un momento dado, otro método “*cambiarArma*” podrá sustituir el objeto equipado por otro, manteniendo la interfaz intacta. Da igual que nuestro soldado porte un rifle, una pistola o un fusil: los detalles de cada estrategia estarán encapsulados dentro de cada una de las clases intercambiables que representan las armas.

Nuestra clase cliente (el soldado) únicamente debe preocuparse de las acciones comunes a todas ellas: atacar, recargar y cambiar de arma. Éste último método, de hecho, será el encargado de realizar la operación de “cambio de estrategia” que forma parte del patrón.

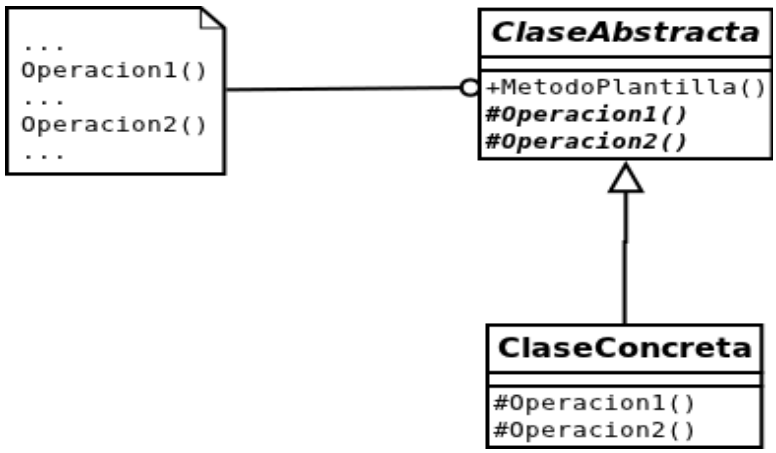
Entonces, el flujo para hacer un diagrama de secuencia sería el siguiente:

1. Creo una estrategia concreta
2. Creo un contexto, a quien le llega por parámetro dicha estrategia
3. Desde el contexto, ejecuto la operación estratégica

Template Method

Definir el esqueleto de un algoritmo de una operación mientras se difieren algunos pasos a la subclase.

Cuando una **porción** de un algoritmo a ser usado en una clase es dependiente del tipo de objeto de las subclases.



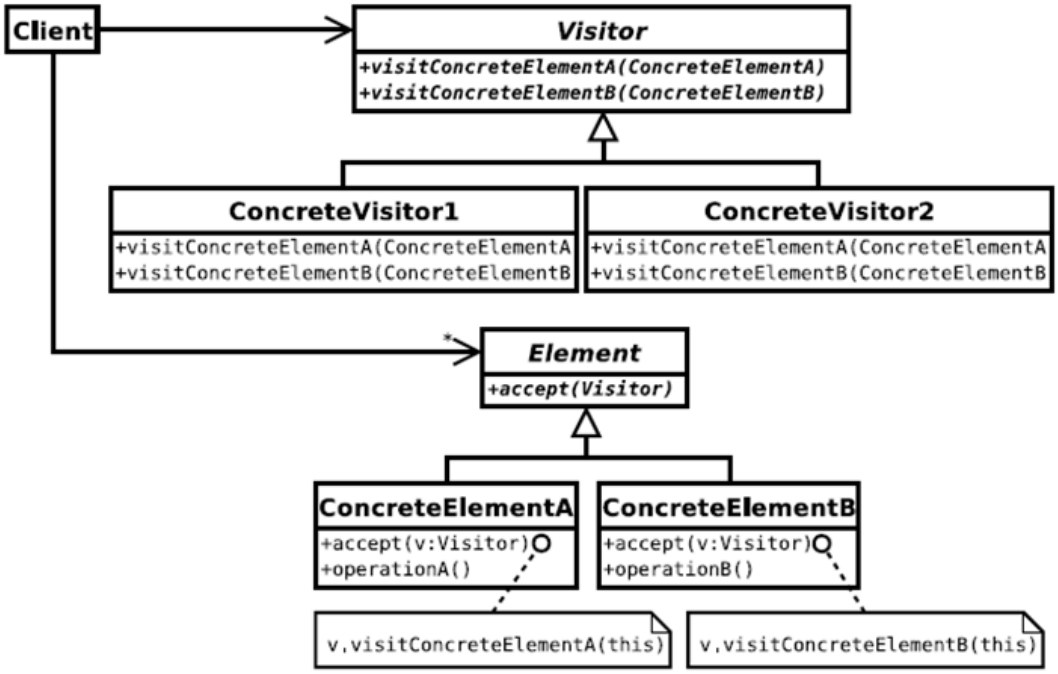
La diferencia con la forma común herencia y sobre escritura de los métodos abstractos es que la clase abstracta contiene un método denominado 'plantilla' que hace llamadas a los que han de ser implementados por las clases que hereden de ella.

La superclase definirá un método que contendrá el esqueleto de ese algoritmo común (método plantilla o template method) y delegará determinada responsabilidad en las clases hijas, mediante uno o varios métodos abstractos que deberán implementar.

Visitor

Permite añadir funcionalidades a una clase sin tener que modificarla, siendo usado para manejar algoritmos, relaciones y responsabilidades entre objetos.

Es un patrón de comportamiento, que permite definir una operación sobre objetos de una jerarquía de clases sin modificar las clases sobre las que opera. Representa una operación que se realiza sobre los elementos que conforman la estructura de un objeto.

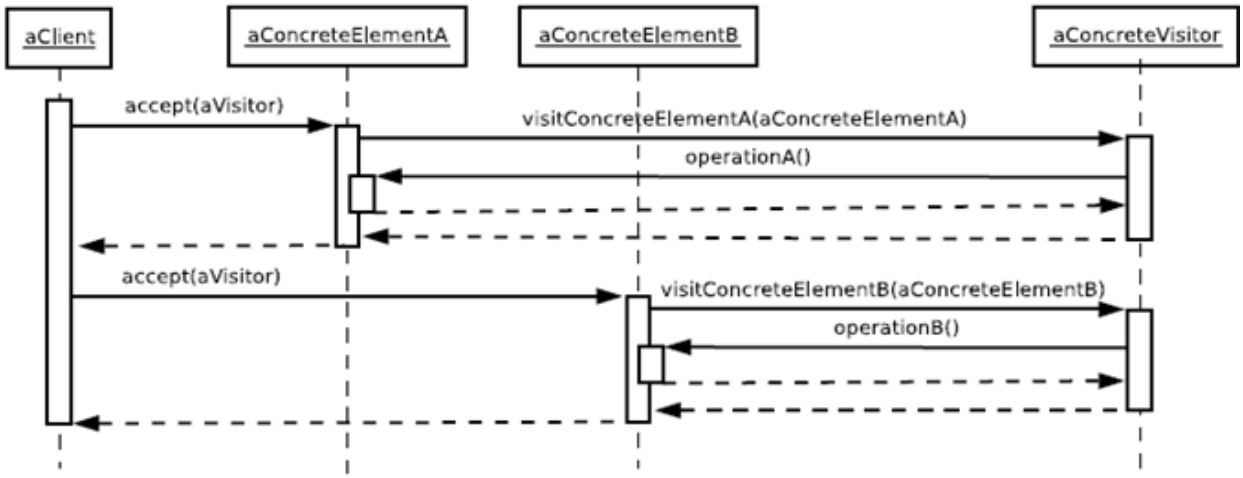


Donde:

- **Visitante** (Visitor): Declara una operación de visita para cada elemento concreto en la estructura de objetos, que incluye el propio objeto visitado
- **Visitante Concreto** (ConcreteVisitor1/2): Implementa las operaciones del visitante y acumula resultados como estado local
- **Elemento** (Element): Define una operación “Accept” que toma un visitante como argumento
- **Elemento Concreto** (ConcreteElementA/B): Implementa la operación “Accept”

La idea básica es que se tiene un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método aceptar (*accept()*) que recibe al objeto visitante (*visitor*) como argumento. El visitante es una interfaz que tiene un método *visit* diferente para cada clase elemento; por tanto, habrá implementaciones de la interfaz visitor de la

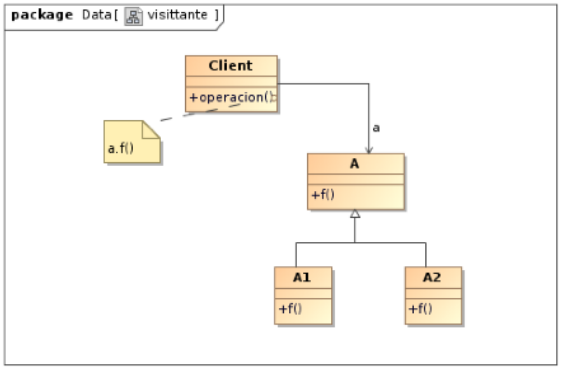
forma: *visitorClase1*, *visitorClase2*... *visitorClaseN*. El método *accept* de una clase elemento llama al método *visit* de su clase. Clases concretas de un visitante pueden entonces ser escritas para hacer una operación en particular.



Aplicación:

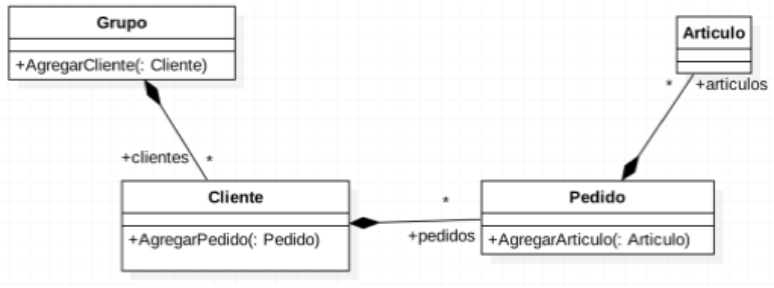
- Cuando varias clases de objetos con interfaces diferentes y se desean realizar operaciones que dependen de sus clases concretas. También cuando se necesitan diversas operaciones sobre objetos de una jerarquía y no se desea recargar las clases con estas operaciones.
- Cuando las clases de la jerarquía no cambian, pero se añaden con frecuencia operaciones a la estructura.
- Si la jerarquía cambia no es aplicable, ya que cada vez que agrega nuevas clases que deben ser visitadas, hay que añadir una operación “visita” abstracta a la clase abstracta del visitante, y debe agregar una aplicación de dicha categoría a cada Visitante concreto que se haya escrito.

Ejemplo donde sería útil:



Ejemplo concreto:

En el siguiente diagrama de clases se representa un *Grupo* de clientes, en el cual cada *Cliente* cuenta con uno o más pedidos, y cada *Pedido* tiene uno o más *Artículos*.



Se requiere crear un nuevo módulo de *Informes* en nuestra solución para poder realizar reportes sobre un grupo de clientes. Estos reportes necesitan analizar todos los datos relacionados con el cliente, por lo que todas las entidades involucradas en el diagrama Cliente, Pedido y Articulo deben poder ser visitados.