Resumen de Patrones de Diseño

Aclaración

Este apunte es un resumen de ciertos patrones publicados en el libro Patrones de Diseño, de Erich Gamma. El mismo debe ser utilizado como guía de lectura rápida una vez que se haya leído y comprendido el libro, es decir, no se recomienda en absoluto aprender patrones con el apunte sin haber leído previamente el libro ya que en él existen muchos ejemplos y casos de uso que acá no figuran.
Los patrones que figuran en el apunte no son todos los del libro, sino aquellos que se dieron en la cátedra Arquitectura de Software de la Universidad de Buenos Aires, Facultad de Ingeniería, en el primer cuatrimestre del 2005.

Patrones de Creación

Patrón Singleton

Categoría de Patrón: *Idiom*

Categoría de Problema: *Creación*

Propósito

Garantiza que una sola clase sólo tenga una instancia y proporciona un punto de acceso global a ella.

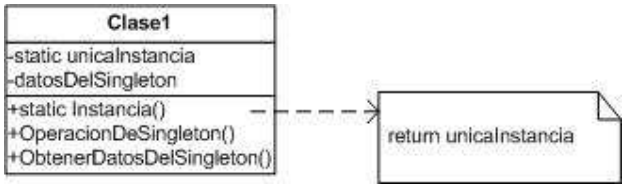
Motivación

Es importante que algunas clases tengan exactamente una instancia. ¿Cómo lo podemos asegurar y que ésta sea fácilmente accesible?. Una variable global no previene de crear múltiples instancias de objetos. Una solución mejor es hacer que sea la propia clase la responsable de su única instancia, quien debe garantizar que no se pueda crear ninguna otra (interceptando las peticiones para crear nuevos objetos) y proporcione un modo de acceder a ella.

Aplicabilidad

Usar Singleton cuando:
1) Deba haber exactamente una instancia de una clase y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
2) La única instancia debería ser extensible mediante herencia y los clientes deberían ser capaces de usar una instancia extendida sin modificar su código.

Estructura



Participantes

1) **Singleton:** Define una operación Instancia que permite que los clientes accedan a su única instancia. Instancia es una operación de clase. Puede ser responsable de crear su única instancia.

Colaboraciones

Los clientes acceden a la instancia de un Singleton exclusivamente a través de la operación Instancia de ésta.

Consecuencias

Ventajas

1) *Acceso controlado a la única instancia:* Encapsula su única instancia, puede tener un control estricto sobre como y cuando acceden a ella los clientes.
2) *Espacio de nombres reducido:* Es una mejora sobre las variables globales. Evita contaminar el espacio de nombres con variables globales que almacenan las instancias.
3) *Permite el refinamiento de operaciones y la representación:* Se puede crear una subclase de la clase Singleton, y es fácil configurar una aplicación con una instancia de esta clase extendida, incluso en tiempo de ejecución.
4) *Permite un numero variable de instancias:* Hace que sea fácil permitir mas de una instancia de la clase. Solo se necesitaría cambiar la operación que otorga acceso a la instancia del Singleton.

Implementación

```
Class Singleton {
public:
        static Singleton* Instancia();
protected:
        Singleton();
Private:
        Static Singleton* _instancia;
};
```

La correspondiente implementación es

```
Singleton* Singleton:: _instancia = 0;
Singleton* Singleton::Instancia () {
        if  (_instancia == 0) {
                _instancia = new Singleton;
}
return _instancia;
}
```

Los clientes acceden al Singleton exclusivamente a través de la función miembro Instancia. La variable _instancia se inicializa a 0, y la función miembro Instancia devuelve su valor, inicializándola con la única instancia en caso de que sea 0. Instancia usa inicialización perezosa; el valor que devuelve no se crea y se almacena hasta que se accede a él por primera vez.
Nótese que el constructor se declara como protegido. Un cliente que trate de crear una instancia de Singleton directamente obtendrá un error en tiempo de compilación. Esto garantiza que solo se puede crear una instancia.

Patrones relacionados

Muchos patrones pueden implementarse usando Singleton: Abstract Factory, Builder, Prototype, etc.

Patrón Factory Method

Categoría de Patrón: *Idiom*

Categoría de Problema: *Creación*

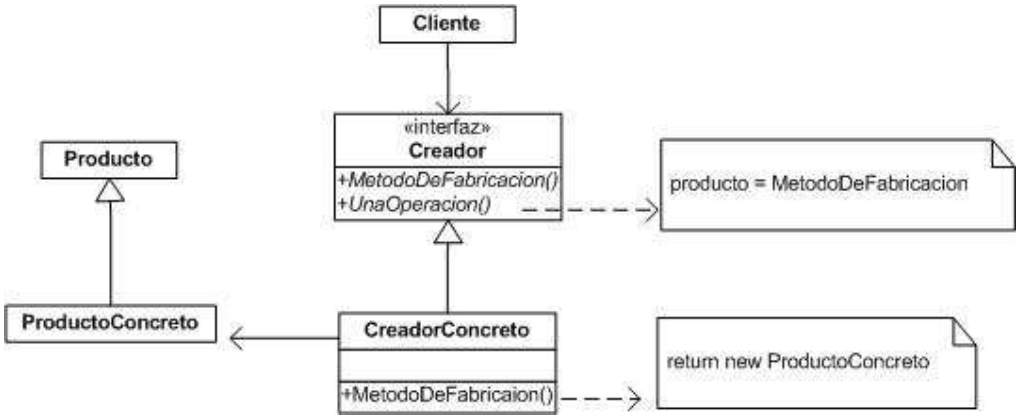
Propósito

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan que clases instanciar. Permite que una clase delegue en sus subclases la creación de objetos (Constructor Virtual).

Aplicabilidad

Usar Factory Method cuando:
1) Una clase no pueda prever la clase de objetos que debe crear.
2) Una clase quiere que sean sus subclases quienes especifiquen los objetos que esta crea.

Estructura



Participantes

1) **Producto:** Define la interfaz de los objetos que crea el método de fabricación.
2) **ProductoConcreto**: Implementa la interfaz Producto.
3) **Creador**: Declara el método de fabricación, el cual devuelve un objeto de tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelve un objeto ProductoConcreto. Puede llamar al método de fabricación para crear un objeto Producto.
4) **CreadorConcreto**: Redefine el método de fabricación para devolver una instancia de un ProductoConcreto.

Colaboraciones

El Creador se apoya en sus subclases para definir el método de fabricación de manera que éste devuelva una instancia del ProductoConcreto apropiado.

Consecuencias

Los métodos de fabricación eliminan la necesidad de ligar clases específicas de la aplicación a nuestro código, quien solo trata con la interfaz Producto.

1) *Proporciona enganches para las subclases:* Crear objetos con un método de fabricación es siempre más flexible que hacerlo directamente, les da a las subclases un punto de enganche para proveer una versión extendida de un objeto (Ventaja).
2) *Los clientes heredan de la clase Creador:* Un inconveniente potencial es que los clientes pueden tener que heredar de la clase Creador simplemente para crear un determinado objeto ProductoConcreto.

Implementación

Métodos de fabricación parametrizados: Permite crear varios tipos de productos. Recibe un parámetro que identifica el tipo de objeto a crear. Tiene la siguiente forma general, donde MiProducto y TuProducto son subclases de Producto:

```
class Creador {
public:
      virtual Producto* Crear (idProducto);
};

Producto* Creador::Crear (idproducto id){
      if  (id == MIO) return new MiProducto;
      if  (id == TUYO) return new TuProducto;
      // repetir para los productos restantes …

      return 0;
}
```

Redefiniendo un método de fabricación parametrizado se pueden extender o cambiar fácilmente y de manera selectiva los productos fabricados por un Creador. Ejemplo: una subclase de MiCreador.

```
Producto* MiCreador::Crear (idproducto id){
      if  (id == TUYO) return new MiProducto;
      if  (id == MIO) return new TuProducto;

      // NOTA: intercambiados TUYO y MIO …

      if  (id == SUYO) return new SuProducto;

      // Llamado cuando falla todo lo demás
      return Creador::Crear(id);
}
```

Nótese que lo ultimo que hace esta operación es llamar al Crear de la clase padre, debido a que MiCreador::Crear sólo trata TUYO, MIO y SUYO de forma diferente a la clase padre. No esta interesada en otras clases, extiende los tipos de productos creados y delega en su padre la responsabilidad de crear todos los productos excepto unos pocos.

Patrones relacionados

El patrón Abstract Factory suele implementarse con métodos de fabricación que también generalmente son llamados desde el interior de Template Method.
El patrón Prototype no necesita heredar de Creador. Sin embargo, suelen requerir una operación Inicializar en la clase Producto. El Creador usa Inicializar para inicializar el objeto.

Patrón Abstract Factory

Categoría de Patrón: *Diseño*

Categoría de Problema: *Creación*

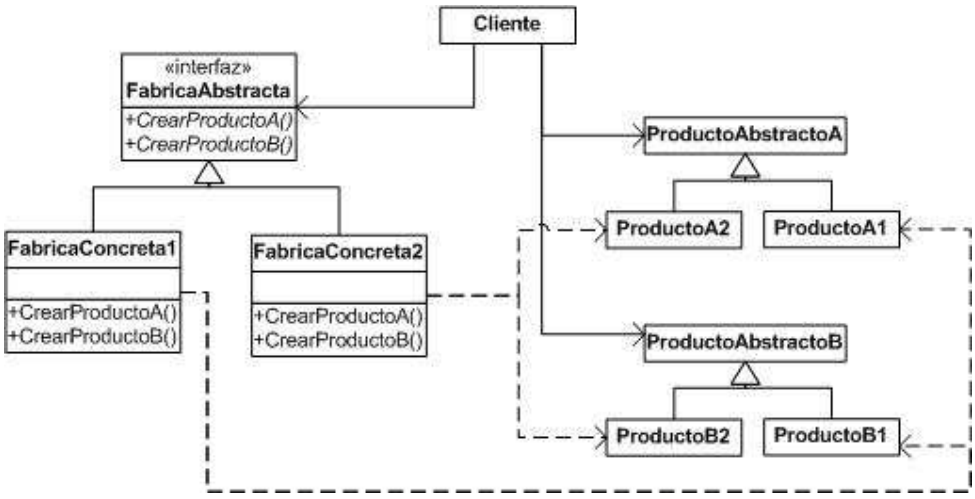
Propósito

Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

Aplicabilidad

Usar Abstract Factory cuando:
1) Un sistema debe ser independiente de cómo se crean, componen y representan sus productos.
2) Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.
3) Quiere proporcionar una biblioteca de clases de productos y solo quiere revelar sus interfaces, no sus implementaciones.

Estructura



Participantes

1) **FabricaAbstracta:** Declara una interfaz para operaciones que crean objetos producto abstractos.
2) **FabricaConcreta**: Implementa las operaciones para crear objetos producto concreto.
3) **ProductoAbstracto**: Declara una interfaz para un tipo de objeto producto.
4) **ProductoConcreto**: Define un objeto producto para que sea creado por la fábrica correspondiente. Implementa la interfaz ProductoAbstracto.
5) **Cliente**: Sólo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto.

Colaboraciones

Normalmente solo se crea una única instancia de una clase FabricaConcreta en tiempo de ejecución, quien crea objetos producto que tienen una determinada implementación. Para crear diferentes objetos producto, los clientes deben usar una fabrica concreta diferente. FabricaAbstracta delega la creación de objetos producto en su subclase FabricaConcreta.

Consecuencias

1) *Aísla las clases concretas:* Ayuda a controlar las clases de objetos que crea una aplicación, encapsula la responsabilidad y el proceso de creación de objetos producto, aísla a los clientes de las clases de implementación, manipulan las instancias a través de sus interfaces abstractas. Los nombres de las clases producto quedan aisladas en la implementación de la fábrica concreta; no aparecen en el código cliente (Ventaja).
2) *Facilita el intercambio de familias de productos:* La clase de una fábrica concreta solo aparece una vez en una aplicación, cuando se crea. Esto facilita cambiarla (como crea una familia completa de productos, cambia toda de una vez) (Ventaja).
3) *Promueve la consistencia entre productos:* Se diseñan objetos producto en una familia para trabajar juntos (Ventaja).
4) *Es difícil dar cabida a nuevos tipos de productos:* Ampliar las fábricas abstractas para producir nuevos tipos de productos no es fácil ya que la interfaz FabricaAbstracta fija el conjunto de productos que se pueden crear (implica cambiar la clase FabricaAbstracta y todas sus subclases) (Desventaja).

Patrones relacionados

Las clases FabricaAbstracta suelen implementarse con el patrón Factory Method, pero también con el patrón Prototype. Una fábrica concreta suele ser un Singleton.

Patrón Prototype

Categoría de Patrón: *Diseño*

Categoría de Problema: *Creación*

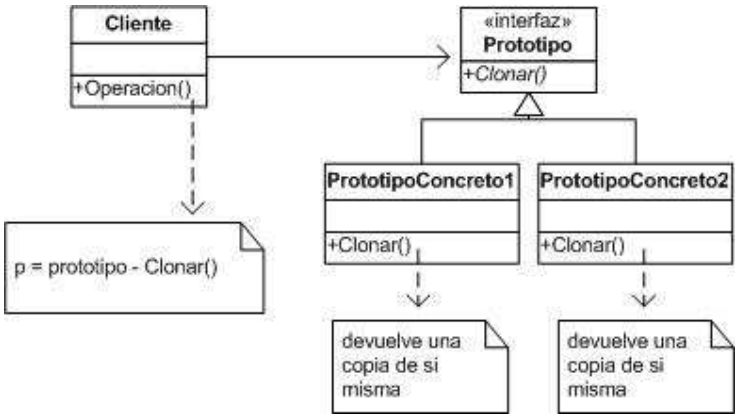
Propósito

Especifica los tipos de objetos a crear por medio de una instancia prototípica y crea nuevos objetos copiando dicho prototipo.

Aplicabilidad

Usar Prototype cuando:
1) Un sistema deba ser independiente de cómo se crean, componen y representan sus productos.
2) Las clases a instancias sean especificadas en tiempo de ejecución.
3) Se quiera evitar construir una jerarquía de fábricas paralela a la jerarquía de clases de productos.
4) Las instancias de una clase puedan tener uno de entre solo unos pocos estados diferentes. Puede ser más adecuado tener un numero equivalente de prototipos y clonarlos, en vez de crear manualmente instancias de la clase cada vez con el estado apropiado.

Estructura



Participantes

1) **Prototipo:** Declara una interfaz para clonarse.
2) **PrototipoConcreto**: Implementa una operación para clonarse.
3) **Cliente**: Crea un nuevo objeto pidiéndole a un prototipo que se clone.

Colaboraciones

Un cliente le pide a un prototipo que se clone.

Consecuencias

Parecidas a las de los patrones Abstract Factory y Builder; oculta al cliente las clases producto conreta, reduciendo así el numero de nombres que conocen los clientes.

1) *Añadir y eliminar productos en tiempo de ejecución:* Permite incorporar a un sistema una nueva clase concreta de producto simplemente registrando una instancia prototípica con el cliente (más flexible ya que un cliente puede instalar y eliminar prototipos en tiempo de ejecución) (Ventaja).
2) *Especificar nuevos objetos modificando valores:* Podemos definir nuevos tipos de objetos creando instancias de clases existentes y registrando esas instancias como prototipos de los objetos del cliente. Un cliente puede exhibir comportamiento nuevo delegando responsabilidad en su prototipo. Permite que los usuarios definan nuevas "clases" sin programación (clonar un prototipo es parecido a crear una instancia de una clase) (Ventaja).
3) *Reduce la herencia:* Permite clonar un prototipo en vez de decirle a un "Factory Method" que cree un nuevo objeto. (Ventaja).
4) *Configurar dinámicamente una aplicación con clases:* Una aplicación que quiere crear instancias de una clase cargada dinámicamente no podrá hacer referencia al constructor de esta estáticamente. En vez de eso, en tiempo de ejecución crea automáticamente una instancia de cada clase cada vez que es cargada y la registra con un gestor de prototipos. Después puede solicitar al gestor de prototipos instancias de nuevas clases que no fueron enlazadas con el programa originalmente (Ventaja).
5) *Implementar Clonar:* Cada subclase de Prototipo debe implementar la operación Clonar, lo cual puede ser difícil en muchos casos (Desventaja).

Implementación

Cuando el numero de prototipos de un sistema no es fijo (puede crearse y destruirse dinámicamente), se mantiene un registro de los prototipos disponibles. Los clientes no gestionaran ellos mismos los prototipos, sino que los guardaran y recuperaran del registro. Un cliente le pedirá al registro, que llamaremos **Gestor de Prototipos,** un prototipo antes de clonarlo.
Un gestor de prototipos es un almacén asociativo que devuelve el prototipo que concuerda con una determinada clave. Tiene operaciones para registrar un prototipo y para desregistrarlo.

Patrones relacionados

Prototype y Abstract Factory son patrones rivales en algunos aspectos, pero también pueden usarse juntos. Un Abstract Factory puede almacenar un conjunto de prototipos a partir de los cuales colar y devolver objetos producto.
Los patrones Composite y Decorator suelen beneficiarse también del Prototype.

Patrón Builder

Categoría de Patrón: *Diseño*

Categoría de Problema: *Creación*

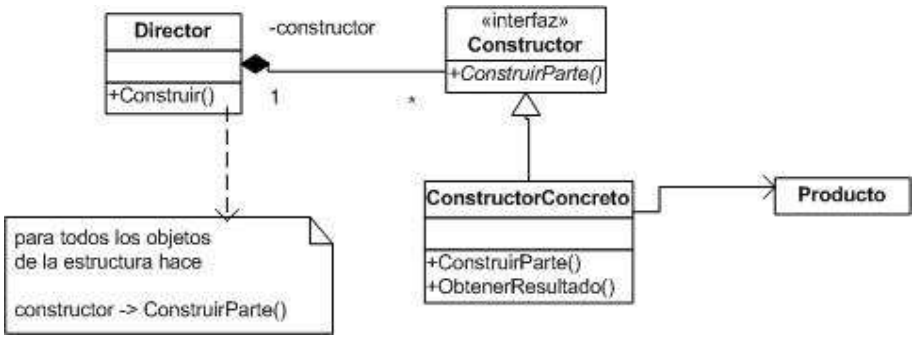
Propósito

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

Aplicabilidad

Usar Builder cuando:
1) El algoritmo para crear un objeto complejo debiera ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
2) El proceso de construcción debe permitir diferentes representaciones del objeto que esta siendo construido.

Estructura



Participantes

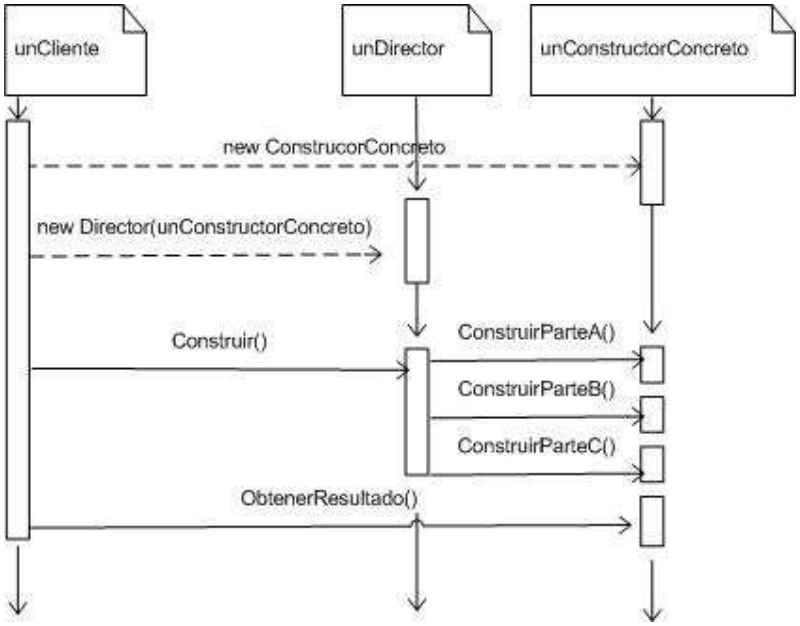
1) Constructor: Especifica una interfaz abstracta para crear las partes de un objeto Producto.
2) ConstructorConcreto: Implementa la interfaz Constructor para construir y ensamblar las partes del producto. Define la representación a crear. Proporciona una interfaz para devolver el producto.
3) Director: Construye un objeto usando la interfaz Constructor.
4) Producto: Representa el objeto complejo en construcción. El ConstructorConcreto construye la representación interna del producto y define el proceso de ensamblaje. Incluye las clases que definen sus partes constituyentes, incluyendo interfaces para ensamblar las partes en el resultado final.

Colaboraciones

1) El cliente crea el objeto Director y lo configura con el constructor deseado.

2) El Director notifica al constructor cada vez que hay que construir una parte de un producto.
3) El Constructor maneja las peticiones del director y las añade al producto.
4) El cliente obtiene el producto del constructor.

El siguiente diagrama de interacción ilustra como cooperan con un cliente el Constructor y el Director:



Consecuencias

1) *Permite variar la representación interna de un producto:* El objeto Constructor proporciona al Director una interfaz abstracta para construir el producto que oculte la representación, la estructura interna del producto y el modo en que este es ensamblado. Todo lo que hay que hacer para cambiar la representación interna del producto es definir un nuevo tipo de constructor (Ventaja).
2) *Aísla el código de construcción y representación:* Aumenta la modularidad al encapsular como se construyen y representan los objetos complejos. Los clientes no necesitan saber nada de las clases que definen la estructura interna del producto; dichas clases no aparecen en la interfaz del Constructor. Cada ConstructorConcreto contiene todo el código para crear y ensamblar un determinado tipo de producto. El código solo se escribe una vez; después, los diferentes Directores puede reutilizarlo para construir variantes de Producto a partir del mismo conjunto de partes (Ventaja).
3) *Proporciona un control más fino sobre el proceso de construcción:* Construye el producto paso a paso, bajo el control del director, quien solo obtiene el producto del constructor una vez que este esta terminado. La interfaz Constructor refleja el proceso de construcción del producto mas que otros patrones de creación (Ventaja).

Implementación

Los constructores construyen sus productos paso a paso, la interfaz de la clase Constructor debe ser lo suficientemente general como para permitir construir productos por parte de todos los tipos de constructores concretos. Modelo de proceso de construcción: Los resultados de las peticiones de construcción simplemente se van añadiendo al producto.

Patrones relacionados

El patrón Abstract Factory se parece a un Builder en que también puede construir objetos complejos. La principal diferencia es que el patrón Builder se centra en construir un objeto complejo paso a paso. El Abstract Factory hace hincapié en familias de objetos producto (simples o complejos). El Builder devuelve el producto como paso final, mientras que el Abstract Factory lo devuelve inmediatamente.
Muchas veces lo que construye el constructor es un Composite.

Conclusión

Hay dos formas habituales de parametrizar un sistema con las clases de objetos que crea:

1) Heredar de la clase que crea los objetos: Uso del patrón Factory Method. El principal inconveniente de este enfoque es que puede requerir crear una nueva subcalse simplemente para cambiar la clase del producto y dichos cambios pueden tener lugar en cascada.
2) Basarse en la composición de objetos: Consiste en definir un objeto que sea responsable de conocer la clase de objetos producto y hacer que sea un parámetro del sistema; patrones Abstract Factory, Prototype y Builder. Los tres implican crear un nuevo “objeto fábrica” cuya responsabilidad es crear objetos producto. El objeto fábrica del Abstract Factory produce objetos de varias clases, mientras que en el caso de Builder construye un producto complejo incrementalmente usando un protocolo de complejidad similar. El Prototype hace que su objeto fábrica construya un producto copiando un objeto prototípico (el objeto fábrica y el prototipo son el mismo objeto, ya que el prototipo es el responsable de devolver el producto).

Que patrón es mejor depende de muchos factores.
Los diseños que usan los patrones Abstract Factory, Prototype o Builder son todavía más flexibles que aquellos que usan el Factory Method, pero también son más complejos. Muchas veces los diseñadores empiezan usando el Factory Method y luego evolucionan hacia los otros patrones de creación a medida que descubren donde es necesaria mas flexibilidad.

Patrones Estructurales

Patrón Adapter

Categoría de Patrón: *Diseño*

Categoría de Problema: *Adaptación*

Problema

Adaptar para facilitar implementación.

Propósito

Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles.

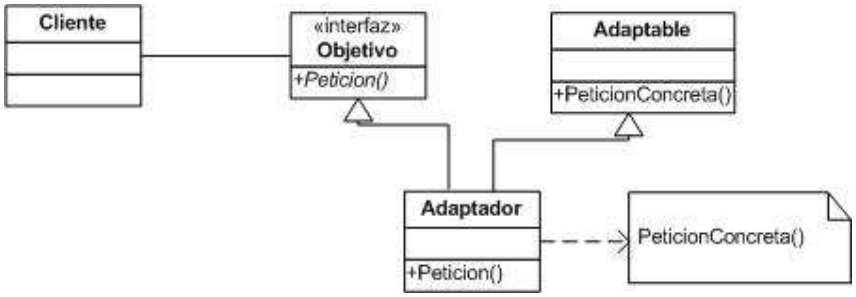
Aplicabilidad

Debería usarse cuando:

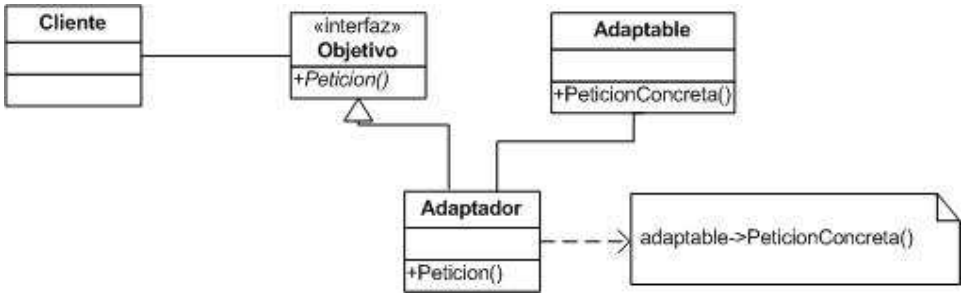
3) Quiere usar una clase existente y su interfaz no concuerda con la clase que necesita.
4) Quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas (que no tienen porque tener interfaces compatibles)
5) Solamente en el caso de un adaptador de objetos, cuando es necesario usar varias subclases existentes, pero no resulta practico adaptar su interfaz heredando de cada una de ellas. Un adaptador de objetos puede adaptar la interfaz de su clase padre.

Estructura

Un adaptador de CLASES usa herencia múltiple para adaptar una interfaz a otra:



Un adaptador de OBJETOS se basa en la composición de objetos



Participantes

2) **Objetivo**: Define la interfaz especifica del dominio que usa el Cliente.
3) **Cliente**: Colabora con objetos que se ajustan a la interfaz Objetivo.
4) **Adaptable**: Define una interfaz existente que necesita ser adaptada.
5) **Adaptador**: Adapta la interfaz de Adaptable a la interfaz Objetivo.

Colaboraciones

Los clientes llaman a operaciones de una instancia de Adaptador. A su vez, el adaptador llama a operaciones de Adaptable, que son las que satisfacen la petición.

Consecuencias

Adaptadores de clases y de objetos, ventajas e inconvenientes:
Un adaptador de clases:

1) Permita que el Adaptador redefina parte del comportamiento de Adaptable, por ser Adaptador una subclase de Adaptable. (Ventaja)
2) Adapta una clase Adaptable a Objetivo, pero se refiere a una clase Adaptable concreta (no nos servirá cuando queramos adaptar una clase y todas sus subclases). (Desventaja)

Un adaptador de objetos

1) Permite que un mismo Adaptador funcione con el Adaptable y todas sus subclases. El Adaptador también puede añadir funcionalidad a todos los adaptables a la vez. (Ventaja)
2) Hace que sea más difícil redefinir el comportamiento de Adaptable. Se necesitará crear una subclase de Adaptable y hacer que el Adaptador se refiera a la subclase en vez de a la clase Adaptable en sí. (Desventaja)

Cuestiones a tener en cuenta

1) Cuanta adaptación hace el Adaptador: Difieren en la cantidad de trabajo necesaria para adaptar Adaptable a la interfaz Objetivo (desde la simple conversión de interfaces hasta permitir un conjunto completamente nuevo de operaciones). Esto depende de lo parecida que sea la interfaz de Objetivo a la de Adaptable.
2) Adaptadores conectables: Una clase es mas reutilizable cuando minimizamos las asunciones que deben hacer otras clases para usarla.
3) Adaptadores Bidireccionales: Los adaptadores no son transparentes a todos los clientes. Un objeto adaptado ya no se ajusta a la interfaz Adaptable, por lo que no puede usarse tal cual en donde pudiera ir un objeto Adaptable. Los adaptadores bidireccionales pueden proporcionar esa transparencia. Resultan útiles cuando dos clientes distintos necesitan ver un objeto de distinta forma.

Patrones relacionados

1) El patrón Bridge tiene una estructura similar a un adaptador de objetos, pero con un propósito diferente: está pensado para separar una interfaz de su implementación, de manera que ambos puedan cambiar fácilmente y de forma independiente uno del otro, mientras que un adaptador esta pensado para cambiar la interfaz de un objeto existente.

2) El patrón Decorator decora otro objeto sin cambiar su interfaz. Un decorator es por tanto mas transparente a la aplicación que un adaptador. Como resultado, el patrón Decorator permite la composición recursiva, lo que no es posible con adaptadores puros.
3) El patrón Proxy define un representante o sustituto de otro objeto sin cambiar su interfaz.

Patrón Bridge

Categoría de Patrón: *Diseño*

Categoría de Problema: *Variación de Servicios*

Propósito

Desacopla una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.

Motivación

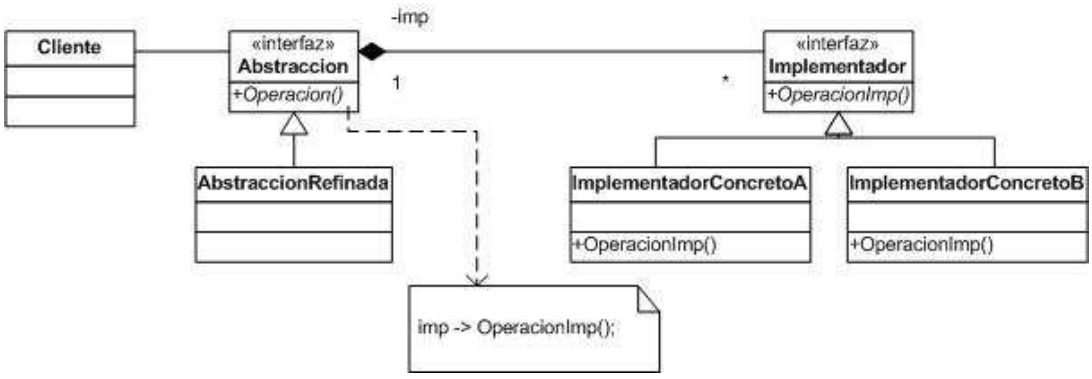
Cuando una abstracción puede tener varias implementaciones posibles, la forma más habitual de darles cabida es mediante la herencia. Una clase abstracta define la interfaz de la abstracción, y las subclases concretas la implementan de distintas formas. Pero este enfoque no siempre es lo bastante flexible. La herencia liga una implementación a la abstracción de forma permanente, lo que dificulta modificar, extender y reutilizar abstracciones e implementaciones de forma independiente.

Aplicabilidad

Use el patrón Bridge cuando:

1) Quiera evitar un enlace permanente entre una abstracción y su implementación (Ej. : cambiar la implementación en tiempo de ejecución).
2) Tanto las abstracciones como las implementaciones deberían ser extensibles mediante subclases (permite combinar las diferentes abstracciones y sus implementaciones, extendiéndolas independientemente).
3) Quiera compartir una implementación entre varios objetos (tal vez usando un contador de referencias) y este hecho deba permanecer oculto al cliente.

Estructura



Participantes

1) **Abstracción:** Define la interfaz de la abstracción. Mantiene una referencia a un objeto de tipo Implementador.
2) **AbstraccionRefinada:** Extiende la interfaz definida por Abstracción.

3) **Implementador:** Define la interfaz de las clases de implementación que no tiene por qué corresponderse exactamente con la de Abstracción; de hecho pueden ser muy distintas. Normalmente la interfaz Implementador solo proporciona operaciones primitivas y Abstracción define operaciones de mas alto nivel basadas en dichas primitivas.

Colaboraciones

Abstracción redirige las peticiones del cliente a su objeto Implementador.

Consecuencias

1) *Desacopla la interfaz y la implementación*: No une permanentemente una implementación a una interfaz, sino que la implementación puede configurarse y cambiar en tiempo de ejecución. Desacoplar Abstracción e Implementador también elimina de la implementación dependencias de tiempo de compilación. Cambiar una clase ya no requiere recompilar la clase Abstracción y sus clientes (Ventaja)
2) *Mejora la extensibilidad*: Podemos extender las jerarquías de Abstracción y de Implementador de forma independiente (Ventaja).
3) *Oculta detalles de implementación a los clientes:* Podemos aislar a los clientes de los detalles de implementación, como el comportamiento de objetos implementadotes (Ventaja).

Patrones relacionados

1) El patrón Abstract Factory puede crear y configurar un Bridge.
2) El patrón Adapter esta orientado a conseguir que trabajen juntas clases que no están relacionadas. El patrón Bridge, por otro lado, se usa al comenzar un diseño para permitir que abstracciones e implementaciones varíen independientemente unas de otras.

Patrón Composite

Categoría de Patrón: *Diseño*

Categoría de Problema: *Descomposición Estructural*

Propósito

Que el cliente vea a las partes y al todo como una misma cosa. Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Contexto

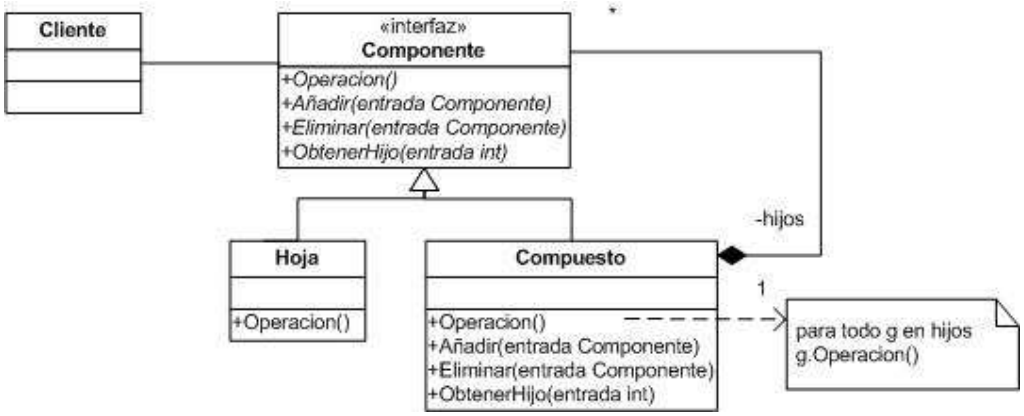
Donde necesitamos componer objetos que representan una jerarquía como suma de partes

Aplicabilidad

Use el patrón Composite cuando:

1) Quiera representar jerarquías de objetos parte-todo.
2) Quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

Estructura



Una estructura de objetos compuestos típica puede parecerse a:

Participantes

1) **Componente**: Declara la interfaz de los objetos de la composición. Implementa el comportamiento predeterminado de la interfaz que es común a todas las clases. Declara una interfaz para acceder a sus componentes hijos y gestionarlos. Opcionalmente, define una interfaz para acceder al padre de un componente en la estructura recursiva y, si es necesario, la implementa.
2) **Hoja:** Representa objetos hoja en la composición. Una hoja no tiene hijos. Define el comportamiento de los objetos primitivos de la composición.
3) **Compuesto:** Define el comportamiento de los componentes que tienen hijos. Almacena componentes hijos. Implementa las operaciones de la interfaz Componente relacionadas con los hijos.
4) **Cliente**: Manipula objetos en la composición a través de la interfaz Componente.

Colaboraciones

Los Clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

Consecuencias

1) *Define jerarquías de clases formadas por objetos primitivos y compuestos:* Los objetos primitivos pueden componerse en otros objetos mas complejos, que a su vez pueden ser compuestos, y así de manera recurrente. Allí donde el código espere un objeto primitivo, también podrá recibir un objeto compuesto (Ventaja)
2) *Simplifica el código del cliente:* Los clientes pueden tratar uniformemente las estructuras compuestas y a los objetos individuales. Normalmente no conocen si están usando una hoja o un componente compuesto. Esto simplifica el cliente, evita tener que escribir funciones con instrucciones if anidadas en las clases que definen la composición (Ventaja).
3) *Facilita añadir nuevos tipos de componentes:* Si se definen nuevas subclases Compuesto u Hoja, funcionarán automáticamente con las estructuras y el código cliente existentes. No hay que cambiar los clientes para nuevas clases Componente (Ventaja).
4) *Puede hacer que un diseño sea demasiado general*: Hace difícil restringir los componentes de un compuesto. A veces queremos que un compuesto solo tenga ciertos componentes. No podemos confiar en el sistema de tipos para que haga cumplir estas restricciones por nosotros, tendremos que usar comprobaciones en tiempo de ejecución.

Patrones relacionados

1) Muchas veces se usa el enlace al componente para implementar el patrón Chain of Responsability.
2) El patrón Decorator suele usarse junto con el Composite. Normalmente ambos tendrán una clase padre común. Por tanto, los decoradores tendrán que admitir la interfaz Componente con operaciones como Añadir, Eliminar y ObtenerHijo.
3) El patrón Flyweight permite compartir componentes, si bien en ese caso estos ya no pueden referirse a sus padres.
4) Se puede usar el patrón Iterator para recorrer las estructuras definidas por el patrón Composite.
5) El patrón Visitor, localiza operaciones y comportamiento que de otro modo estaría distribuido en varias clases Compuesto y Hoja.

Patrón Decorator

Categoría de Patrón: *Diseño*

Categoría de Problema: *Extensión de servicios*

Propósito

Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Motivación

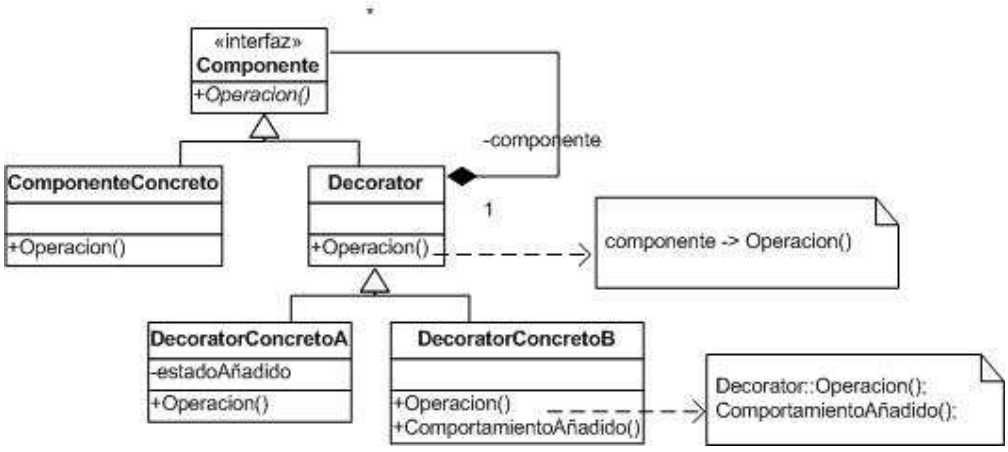
A veces queremos añadir responsabilidades a objetos individuales en vez de a toda una clase.
Un modo de hacerlo es a través de la herencia, pero esto es inflexible ya que se hace estáticamente, un cliente no puede controlar cómo ni cuándo decorar el componente.
Un enfoque más flexible es encerrar el componente en otro objeto, el "decorador", que se ajusta a la interfaz del componente que decora de manera que su presencia es transparente a sus clientes (el decorador reenvía las peticiones al componente, pudiendo realizar acciones adicionales antes o después del reenvío). Dicha transparencia permite anidar decoradores recursivamente, permitiendo así un numero ilimitado de responsabilidades añadidas.

Aplicabilidad

Usar Decorator:

1) Para añadir objetos individuales de forma dinámica y transparente, sin afectar a otros objetos.
2) Para responsabilidades que pueden ser retiradas.
3) Cuando la extensión mediante la herencia no es posible.

Estructura



Participantes

1) **Componente**: Define la interfaz para los objetos a los que se pueden añadir responsabilidades dinámicamente.
2) **ComponenteConcreto:** Define un objeto al que se pueden añadir responsabilidades adicionales.
3) **Decorador**: Mantiene una referencia a un objeto Componente y define una interfaz que se ajusta a la interfaz del Componente.
4) **DecoradorConcreto**: Añade responsabilidades al componente

Consecuencias

1) *Mas flexibilidad que la herencia estática*: La herencia requiere crear una nueva clase para cada responsabilidad adicional que se quiere agregar, dando lugar a muchas clases diferentes, incrementando la complejidad de un sistema. Por el contrario, con el patrón Decorador se puede añadir y quitar responsabilidades en tiempo de ejecución simplemente poniéndolas y quitándolas. Proporcionar diferentes clases Decorador para una determinada clase componente permite mezclar responsabilidades (Ventaja).
2) *Evita clases cargadas de funciones en la parte de arriba de la jerarquía:* En vez de tratar de permitir todas las funcionalidades inimaginables en una clase compleja y adaptable, podemos definir primero una clase simple y añadir luego funcionalidad incrementalmente con objetos Decorador. La funcionalidad puede obtenerse componiendo partes simples (una aplicación no necesita pagar por características que no usa) (Ventaja).
3) *Un decorador y su componente no son idénticos:* Un decorador se comporta como un revestimiento transparente, pero desde el punto de vista de la identidad de un objeto, un componente decorado no es idéntico al componente en sí. No deberíamos apoyarnos en la identidad de objetos (casteo?) cuando estamos usando decoradores (Desventaja)
4) *Muchos objetos pequeños:* Un diseño que usa el patrón Decorador suele dar como resultado sistemas formados por muchos objetos pequeños muy parecidos que son fáciles de adaptar por quienes los comprenden bien, pero pueden ser difíciles de aprender y depurar (Desventaja).

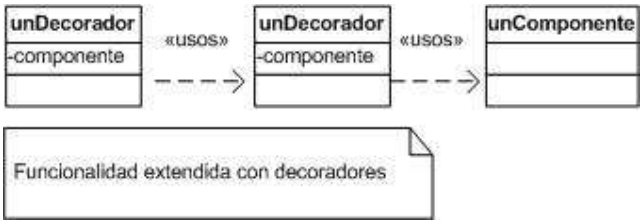
Implementación

Tener en cuenta estas 3 cuestiones

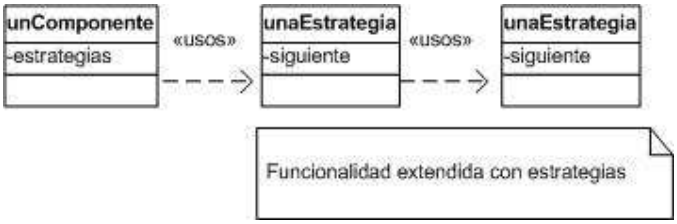
1) *Concordancia de interfaces:* La interfaz de un objeto decorador debe ajustarse a la interfaz del componente que decora (las clases DecoradorConcreto deben por lo tanto heredar de una clase común.
2) *Omisión de la clase abstracta Decorador:* No hay necesidad de definir una clase abstracta Decorador cuando solo necesitamos añadir una responsabilidad.
3) *Mantener ligereas las clases Componente:* Para garantizar una interfaz compatible, los componentes y los decoradores deben descender de una clase Componente común, quien debería centrarse en definir una interfaz, no en guardar datos. La definición de cómo se representan los datos debería delegarse en las subclases; de no ser así, la complejidad de la clase Componente puede hacer que los decoradores sean demasiados pesados como para usar un gran numero de ellos. Poner mucha funcionalidad en el Componente también incrementa la probabilidad de que las subclases concretas estén pagando por características que no necesitan.

Similitudes con el patrón Strategy

Se dice que el patrón Decorator le cambia la *PIEL* a un objeto ya que se lo puede pensar como un revestimiento de un objeto que cambia su comportamiento.
En cambio se dice que el Strategy le cambia las *TRIPAS* a un objeto, ya que cambia las interioridades del mismo.
El patrón Strategy (quien delega parte de la funcionalidad del Componente en un objeto Estrategia aparte, permitiendo alterar o extender la funcionalidad del Componente cambiando el objeto Estrategia), es una mejor elección en los casos en que la clase Componente sea muy pesada, lo que hace que el patrón Decorator sea demasiado costoso de aplicar.
Puesto que el patrón Decorator solo cambia la parte exterior de un objeto, el Componente no tiene que saber nada de sus decoradores (son transparentes).



Con estrategias, el Componente conoce sus posibles extensiones, por lo tanto tiene que referenciar y mantener las estrategias correspondientes.



El enfoque basado en estrategias puede requerir modificar el componente para permitir nuevas extensiones. Por otro lado, una estrategia puede tener su propia interfaz especializada, mientras que la interfaz de un decorador debe ajustarse a la del componente.

Código de Ejemplo

Suponemos que hay una clase Componente llamada ComponenteVisual:

```
class ComponenteVisual {
public:
	ComponenteVisual();

	virtual void Dibujar();
	virtual void CambiarTamaño();
	// ...
};
```

Una subclase de ComponenteVisual llamada Decorador, de la cual heredaremos para obtener diferentes decoraciones:

```
class Decorador : public ComponenteVisual {
public:
	Decorador (ComponenteVisual *);

	virtual void Dibujar();
```

```
        virtual void CambiarTamaño();
        // ...
private:
        ComponenteVisual * _componente;
};

void Decorador :: Dibujar () {
        _componente -> Dibujar();
}

void Decorador :: CambiarTamaño () {
        _componente -> CambiarTamaño ();
}
```

Las subclases de Decorator definen decoraciones concretas (ej: DecoradorBorde añade un borde a su componente):

```
class DecoradorBorde : public Decorator {
public:
        DecoradorBorde (ComponenteVisual*, int anchoBorde);
        virtual void Dibujar();
private:
        void DibujarBorde(int);
private:
        int _ancho;
};

void DecoradorBorde :: Dibujar () {
        Decorador::Dibujar();
        DibujarBorde(_ancho);
}
```

Patrones relacionados

1) *Adapter:* Un decorador se diferencia de un adaptador en que el decorador solo cambia las responsabilidades de un objeto, no su interfaz, mientras que el adaptador le da a un objeto una interfaz completamente nueva.
2) *Composite:* Podemos ver a un decorador como un compuesto degenerado que solo tiene un componente. No obstante, un decorador añade responsabilidades adicionales (no esta pensado para la agregación de objetos).
3) *Strategy:* Permite cambiar el servicio, las tripas de un objeto. El decorador permite cambiar el exterior, la piel, extendiendo su funcionalidad. Son 2 formas alternativas de modificar un objeto.

Patrón Facade

Categoría de Patrón: *Diseño*

Categoría de Problema: *Control de acceso*

Propósito

Proporcionar una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

Problema

Ordenar acceso a recursos (se empezó a usar cuando gran parte del código C fue reemplazado por C++, Ej. : MFC, API que ordenan el acceso a recursos vinculados con el sistema operativo. No agregan funcionalidad, son un punto de entrada, de control, delegan. Parecido a Mediator, solo que en este el conocimiento era de ida y vuelta ya que conoce a las clases y estas lo conocen a él. En cambio en el Facade las clases del subsistema no conocen a este.

Motivación

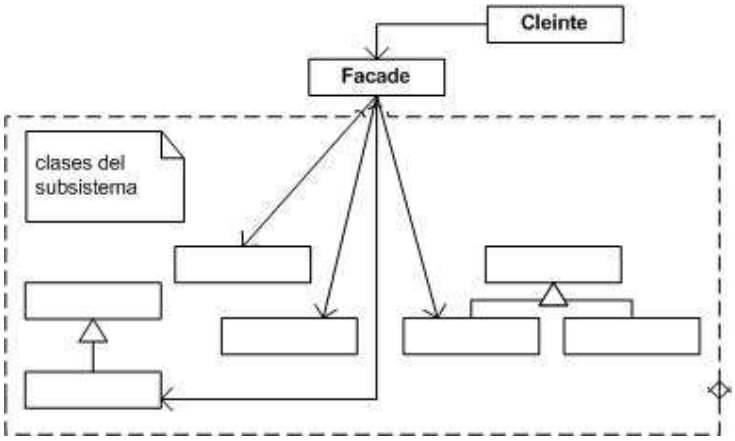
Estructurar un sistema en subsistemas ayuda a reducir la complejidad. Un típico objetivo de diseño es minimizar la comunicación y dependencias entre subsistemas. Un modo de lograr esto es introduciendo un objeto fachada que proporcione una interfaz única y simplificada para los servicios más generales del subsistema.

Aplicabilidad

Use el patrón Facade cuando:

1) Quiera proporcionar una interfaz simple para un subsistema complejo. Los subsistemas suelen volverse mas complicados a medida que van evolucionando. Una fachada puede proporcionar una vista simple del subsistema que resulta adecuada para la mayoría de los clientes y solo aquellos clientes que necesitan mas personalización necesitarán ir mas allá de la fachada.
2) Haya muchas dependencias entre los clientes y las clases que implementan una abstracción. Se introduce una fachada para desacoplar el subsistema de sus clientes y de otros subsistemas, promoviendo así la independencia entre subsistemas y portabilidad.
3) Queramos dividir en capas nuestros subsistemas. Se usa una fachada para definir un punto de entrada en cada nivel del subsistema. Si estos son dependientes, se pueden simplificar las dependencias para entre ellos haciendo que se comuniquen entre sí únicamente a través de sus fachadas.

Estructura



Participantes

1) **Fachada**: Sabe que clases del subsistema son las responsables ante una petición. Delega las peticiones de los clientes en los objetos apropiados del subsistema.
2) **Clases del subsistema**: Implementan la funcionalidad del subsistema. Realizan las labores encomendadas por el objeto Facade. No conocen a la fachada, es decir, no tienen referencias a ella.

Colaboraciones

Los clientes se comunican con el subsistema enviando peticiones al objeto Fachada, el cual las reenvía a los objetos apropiados del subsistema. Aunque son los objetos del subsistema los que realizan el trabajo real, la fachada puede tener que hacer algo de trabajo para pasar de su interfaz a la del subsistema.
Los clientes que usan la fachada no tienen que acceder directamente a los objetos del subsistema (pero podrían hacerlo si lo desean, si quieren una mayor personalización).

Consecuencias

Ventajas

1) Oculta a los clientes los componentes del subsistema, reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar.
2) Promueve un débil acoplamiento entre el subsistema y sus clientes (nos permite modificar los componentes de un subsistema sin que sus clientes se vean afectados). Las fachadas ayudan a estructurar en capas un sistema y las dependencias entre los objetos. Pueden eliminar dependencias complejas o circulares (consecuencia importante cuando el cliente y el subsistema se implementan por separado).

Implementación

Reducción del acoplamiento cliente-subsistema: El acoplamiento entre clientes y el subsistema puede verse reducido todavía mas haciendo que Facade sea una clase abstracta con subclases concretas para las diferentes implementaciones de un subsistema. De esa manera los clientes pueden comunicarse con el subsistema a través de la interfaz de una

clase abstracta Facade. Este acoplamiento abstracto evita que los clientes tengan que saber que implementación de un subsistema están usando.
Una alternativa a la herencia es configurar un objeto Facade con diferentes objetos del subsistema. Para personalizar la fachada basta con reemplazar uno o varios de tales objetos.

Patrones relacionados

1) El patrón Abstract Factory puede usarse para proporcionar una interfaz para crear el subsistema de objetos de forma independiente a otros subsistemas. También puede ser una alternativa a las fachadas para ocultar clases específicas de la plataforma.
2) El patrón Mediator es parecido al Facade en el sentido de que abstrae funcionalidad a partir de unas clases existentes. Sin embargo, el propósito del Mediatos es abstraer cualquier comunicación entre objetos similares, a menudo centralizando la funcionalidad que no pertenece a ninguno de ellos. Los colegas de un mediator solo se preocupan de comunicarse con el y no entre ellos directamente. Por el contrario, una fachada simplemente abstrae una interfaz para los objetos del subsistema, haciéndolos más fácil de usar; no define nueva funcionalidad y las clases del subsistema no saben de su existencia.
3) Normalmente solo necesita un objeto Facade, por lo que suelen implementarse como Singletons.

Patrón Proxy

Categoría de Patrón: *Diseño*

Categoría de Problema: *Control de acceso*

Propósito

Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.

Motivación

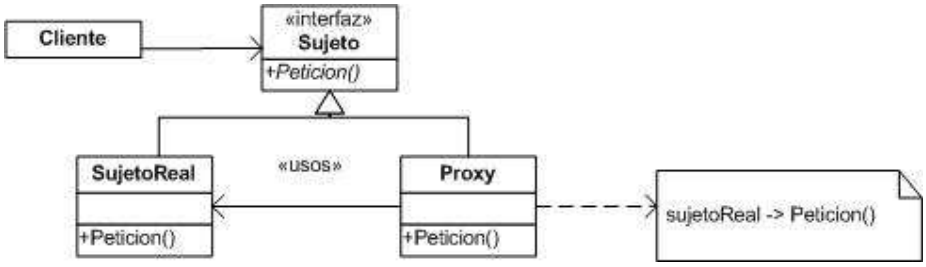
Una razón para controlar el acceso a un objeto es retrasar todo el coste de su creación e inicialización hasta que sea realmente necesario usarlo.

Aplicabilidad

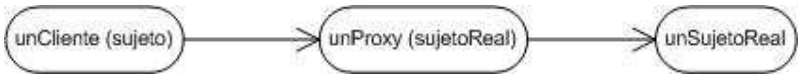
Este patrón es aplicable cada vez que hay necesidad de una referencia a un objeto mas sofisticada que un simple puntero:

1) **Proxy Remoto**: Proporciona un representante local de un objeto situado en otro espacio de direcciones.
2) **Proxy Virtual**: Crea objetos costosos por encargo (es decir, solo cuando se van a usar, no al principio de la aplicación).
3) **Proxy de Protección**: Controla el acceso al objeto original. Son útiles cuando los objetos debieran tener diferentes permisos de acceso.
4) **Referencia Inteligente**: Es un sustituto de un simple puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto, Ej:
   a) Contar el numero de referencias al objeto real.
   b) Cargar un objeto persistente cuando es referenciado por primera vez.
   c) Comprobar que se bloquea el objeto real antes de acceder a él para garantizar que no pueda ser modificado por ningún otro objeto.

Estructura



Este es un posible diagrama de objetos de una estructura de proxies en tiempo de ejecución.



Participantes

1) **Proxy:** Mantiene una referencia que le permite acceder al objeto real. Proporciona una interfaz idéntica a la del Sujeto, de manera que pueda ser sustituido por el SujetoReal. Controla el acceso al SujetoReal y puede ser responsable de su creación y borrado. Otras responsabilidades dependen del tipo de proxy:
   a) Los **Proxies Remotos** son responsables de codificar una petición y sus argumentos para enviar al SujetoReal que se encuentra en un espacio de direcciones diferente.
   b) Los **Proxies Virtuales** pueden guardar información adicional sobre el SujetoReal, por lo que pueden retardar el acceso al mismo.
   c) Los **Proxies de Protección** comprueban que el llamador tenga los permisos de acceso necesarios para realizar una petición.
2) **Sujeto:** Define la interfaz común para el SujetoReal y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que se espere un SujetoReal.
3) **SujetoReal**: Define el objeto real representado.

Colaboraciones

El Proxy redirige peticiones al SujetoReal cuando sea necesario, dependiendo del tipo de Proxy.

Consecuencias

El patrón Proxy introduce un nivel de indirección al acceder a un objeto que tiene muchos usos y ventajas dependiendo del tipo de Proxy:

1) Un Proxy Remoto puede ocultar el hecho de que un objeto reside en un espacio de direcciones diferente.
2) Un Proxy Virtual puede llevar a cabo optimizaciones tales como crear un objeto por encargo.
3) Tanto los Proxies de Protección como las Referencias Inteligentes permiten realizar tareas de mantenimiento adicionales cuando se accede a un objeto.

Los proxies no siempre tienen que conocer el tipo de SujetoReal. Si una clase Proxy puede tratar con su Sujeto sólo a través de una interfaz abstracta, entonces no hay necesidad de hacer una clase Proxy para cada clase de SujetoReal (el Proxy puede tratar de manera uniforme a todas las clases SujetoReal). Pero si los objetos Proxy van a crear instancias de SujetoReal (como los proxies virtuales) entonces tienen que conocer la clase concreta.

Patrones relacionados

1) El patrón Adapter proporciona una interfaz diferente para el objeto que adapta. Por el contrario, un Proxy tiene la misma interfaz que su sujeto. No obstante, un Proxy utilizado para protección de acceso podría rechazar una operación que el sujeto si realiza, de modo que su interfaz puede ser realmente un subconjunto de la del sujeto.
2) El patrón Decorator, si bien puede tener una implementación parecida a los proxies, tienen un propósito diferente. Un Decorator añade una o más responsabilidades a un objeto, mientras que un proxy controla el acceso a un objeto.

Conclusión

Existen similitudes entre los patrones estructurales, especialmente en sus participantes y colaboradores. Esto es debido a que se basan en un mismo pequeño conjunto de mecanismos del lenguaje para estructurar el código y los objetos: herencia simple y herencia múltiple para los patrones basados en clases, y composición de objetos para los patrones de objetos. Pero estas similitudes ocultan los diferentes propósitos de estos patrones.

Adapter vs. Bridge

Los patrones Adapter y Bridge tienen algunos atributos comunes. Ambos promueven la flexibilidad al proporcionar un nivel de indirección a otro objeto. Ambos implican reenviar peticiones a este objeto desde una interfaz distinta de la suya propia.
La diferencia fundamental entre estos patrones radica en su propósito. Adapter se centra en resolver incompatibilidades entre dos interfaces existentes. No presta atención a como se implementan dichas interfaces, ni tiene en cuenta como podrían evolucionar de forma independiente. Es un modo de lograr que dos clases diseñadas independientemente trabajen juntas sin tener que volver a implementar una u otra. Por otro lado, Bridge une una implementación con sus implementaciones (que pueden ser numerosas). Proporciona una interfaz estable a los clientes permitiendo, no obstante, que cambien las clases que la implementan. También permite incorporar nuevas implementaciones a medida que evoluciona el sistema.
Como resultado de estas diferencias, los patrones Adapter y Bridge suelen usarse en diferentes puntos del ciclo de vida del software. Un adaptador suele hacerse necesario cuando se descubre que deberían trabajar juntas dos clases incompatibles, generalmente para evitar duplicar código, y este acoplamiento no había sido previsto. Por el contrario, el usuario de un puente sabe de antemano que una abstracción debe tener varias implementaciones y que unas y otras pueden variar independientemente. El patrón Adapter hace que las cosas funcionen después de que han sido diseñadas; el Bridge lo hace antes. Cada uno resuelve un problema distinto.
Podemos ver un Facade como un adaptador para un conjunto de objetos. Pero esta implementación obvia el hecho de que una fachada define una nueva interfaz, mientras que un adaptador reutiliza una interfaz existente (hace que trabajen juntas dos interfaces existentes en vez de tener que definir una completamente nueva).

Composite vs. Decorator vs. Proxy

Los patrones Composite y Decorator tienen diagramas de estructura parecidos, ambos se basan en la composición recursiva para organizar un numero indeterminado de objetos, pero se usan para propósitos diferentes.
El patrón Decorator esta diseñado para permitir responsabilidades a objetos sin crear subclases, evita la explosión de subclases a la que puede dar lugar al intentar cubrir cada combinación de responsabilidades estáticamente. El patrón Composite tiene un propósito diferente, consiste en estructurar subclases para que se puedan tratar de manera uniforme muchos objetos relacionados y que múltiples objetos puedan ser tratados como uno solo, es decir, no se centra en la decoración sino en la representación.
Estos propósitos son distintos pero complementarios, por lo tanto, suelen usarse conjuntamente.
Otro patrón con una estructura similar al Decorator es el Proxy. Ambos describen como proporcionar un nivel de indirección a un objeto y las implementaciones mantienen una referencia a otro objeto, al cual reenvían las peticiones. Pero están pensados para propósitos diferentes.

Al igual que el Decorator, el patrón Proxy compone un objeto y proporciona una interfaz idéntica a los clientes. A diferencia del Decorator, el patrón Proxy no tiene que ver con asignar o quitar propiedades dinámicamente y tampoco esta diseñado para la composición recursiva. Su propósito es proporcionar un sustituto para un sujeto cuando no es conveniente o deseable acceder directamente a él, debido a, por ejemplo, residir en una máquina remota, tener acceso restringido o ser persistente.
En el patrón Proxy, el sujeto define la principal funcionalidad y el proxy proporciona (o rechaza) el acceso al mismo. En el Decorator, el componente proporciona solo parte de funcionalidad y uno o más decoradores hacen el resto. El patrón Decorator se encarga de aquellas situaciones en las que no se puede determinar toda la funcionalidad de un objeto en tiempo de compilación, o al menos no resulta conveniente hacerlo. Ese no es el caso del Proxy, ya que este se centra en una relación (entre el proxy y su sujeto) que puede expresarse estáticamente. Pero esto no significa que estos patrones no puedan combinarse.

Patrones de Comportamiento

Patrón Chain of Responsibility

Categoría de Patrón: *Diseño*

Categoría de Problema: *Organización del trabajo*

Propósito

Evita acoplar el emisor de una petición a su receptor, dando a mas de un objeto la posibilidad de responder a la petición. Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

Motivación

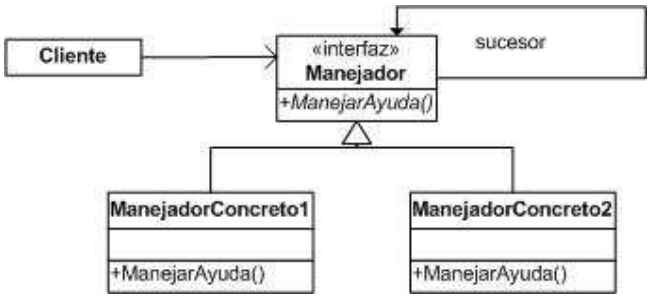
La idea de este patrón es desacoplar a los emisores y a los receptores dándole a varios objetos la posibilidad de tratar una petición, que se pasa a través de una cadena de objetos hasta que es procesada por alguno de ellos.
Para reenviar la petición a lo largo de la cadena, garantizando que los receptores permanezcan implícitos, cada objeto de la cadena comparte una interfaz común para procesar peticiones y para acceder a su sucesor en la cadena.

Aplicabilidad

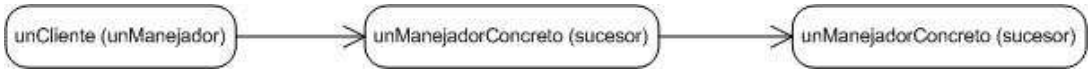
Usar Chain of Responsability cuando:

6) Hay mas de un objeto que puede manejar una petición y el manejador no se conoce a priori, sino que debería determinarse automáticamente.
7) Se quiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.
8) El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

Estructura



Una estructura de objetos típica podría parecerse a ésta:



Participantes

6) **Manejador**: Define una interfaz para tratar las peticiones. Opcionalmente, implementa el enlace al sucesor.
7) **ManejadorConcreto**: Trata las peticiones de las que es responsable. Puede acceder a su sucesor. Si el ManejadorConcreto puede manejar la petición, lo hace; en caso contrario la reenvía a su sucesor.
8) **Cliente**: Inicializa la petición a un objeto ManejadorConcreto de la cadena.

Colaboraciones

Cuando un cliente envía una petición, ésta se propaga a través de la cadena hasta que un objeto ManejadorConcreto se hace responsable de procesarla.

Consecuencias

4) *Reduce el acoplamiento:* Libera a un objeto de tener que saber que otro objeto maneja una petición. Ni el emisor ni el receptor se conocen y tampoco tienen que conocer la estructura de la cadena. Simplifica las interconexiones entre objetos. En vez de que los objetos mantengan referencias a todos los posibles receptores, solo tienen una única referencia a su sucesor (Ventaja).
5) *Añade flexibilidad para asignar responsabilidades a objetos:* Se pueden añadir o cambiar responsabilidades para tratar una petición modificando la cadena en tiempo de ejecución (Ventaja).
6) *No se garantiza la recepción:* Dado que las peticiones no tienen un receptor explícito, no hay garantía de que sean manejadas (la petición puede alcanzar el final de la cadena sin haber sido procesada). Una petición también puede quedar sin tratar cuando la cadena no está configurada correctamente (Desventaja).

Patrones relacionados

3) Este patrón se suele aplicar conjuntamente con el patrón Composite, en el que los padres de los componentes pueden actuar como sucesores.

Patrón Command

Categoría de Patrón: *Diseño*

Categoría de Problema: *Organización del trabajo*

Propósito

Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones y poder deshacer las operaciones.

Contexto

Quiero ver como se comunican las instancias de un conjunto de clases

Problema

Desacoplar las instancias de los objetos que hacen “request” de los que prestan el servicio (response)

Motivación

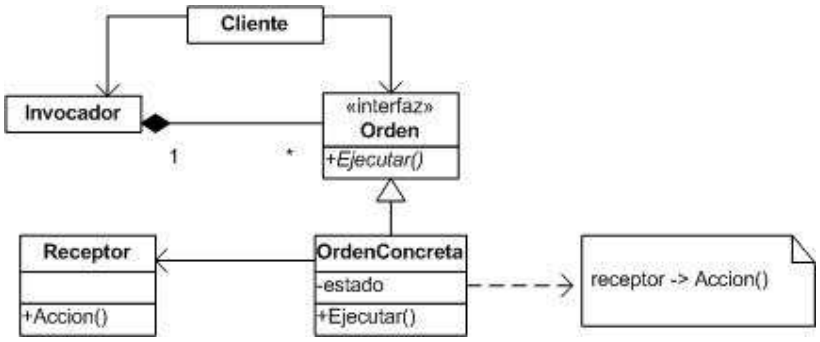
A veces es necesario enviar peticiones a objetos sin saber nada acerca de la operación solicitada o de quien es el receptor de la petición.

Aplicabilidad

Usar Command cuando se quiera:

1) Parametrizar objetos con una acción a realizar. En un lenguaje procedural se puede expresar dicha parametrización con una función callback (registrada en algún sitio para que sea llamada mas tarde). Los objetos Orden son un sustituto orientado a objetos para las funciones callback.
2) Especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo. Un objeto Orden puede tener un tiempo de vida independiente de la petición original.
3) Permitir deshacer. La operación Ejecutar de Orden puede guardar un estado y la operación en una lista “historial”. Debe añadirse una operación Deshacer que anule los efectos de una llamada anterior a Ejecutar. Se pueden lograr niveles ilimitados recorriendo el historial, llamando respectivamente a Deshacer y Ejecutar.
4) Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de una caída del sistema. Aumentando la interfaz de Orden con operaciones para cargar y guardar se puede mantener un registro persistente de los cambios.
5) Ofrecer un modo de modelar transacciones (encapsular un conjunto de cambios sobre unos datos). Las órdenes tienen una interfaz común, permitiendo así invocar a todas las transacciones del mismo modo.

Estructura



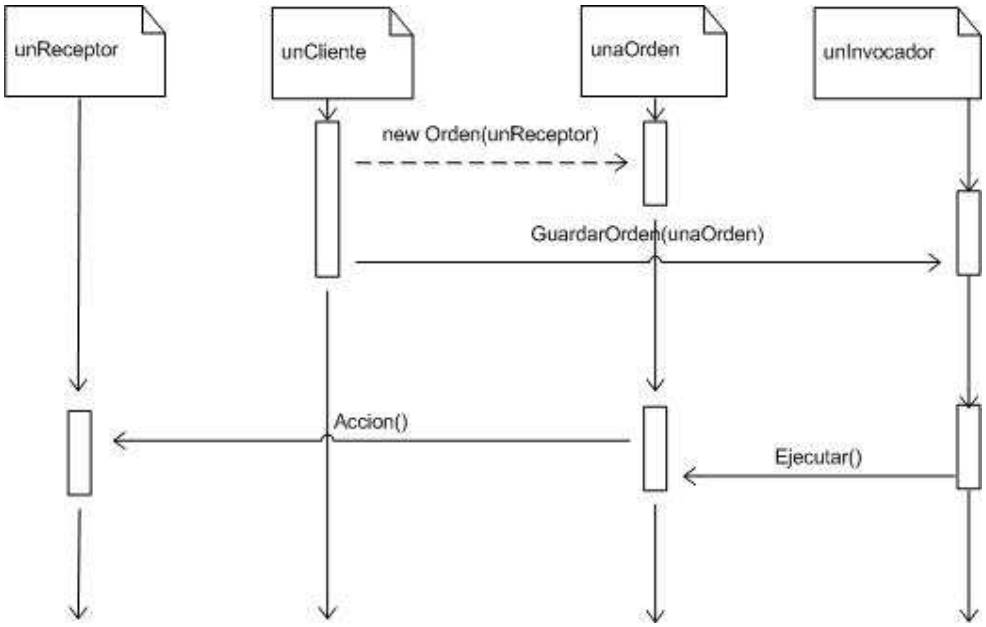
Participantes

1) **Orden**: Declara una interfaz para ejecutar una operación.
2) **OrdenConcreta**: Define un enlace entre un objeto Receptor y una acción. Implementa Ejecutar invocando la correspondiente operación u operaciones del Receptor.
3) **Cliente**: Crea un objeto OrdenConcreta y establece su receptor.
4) **Invocador**: Le pide a la orden que ejecute la petición.
5) **Receptor**: Sabe como llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede actuar como Receptor.

Colaboraciones

1) El cliente crea un objeto OrdenConcreta y especifica su receptor.
2) Un objeto Invocador almacena el objeto OrdenConcreta.
3) El Invocador envía una petición llamando a Ejecutar sobre la orden. Cuando las ordenes se pueden deshacer, OrdenConcreta guarda el estado para deshacer la orden antes de llamar a Ejecutar.
4) El objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición.

El siguiente diagrama muestra las interacciones entre los objetos.



Consecuencias

Ventajas

1) Orden desacopla el objeto que invoca la operación de aquel que sabe como realizarla.
2) Las ordenes son objetos de primera clase, pueden ser manipulados y extendidos.
3) Se pueden ensamblar ordenes en una orden compuesta (generalmente se usa el patrón Composite).
4) Es fácil añadir nuevas ordenes ya que no hay que cambiar las clases existentes.

Patrones relacionados

1) El patrón Composite se puede usar para ordenes compuestas.
2) El patrón Memento puede mantener el estado que necesitan las ordenes para anular sus efectos.
3) El patrón Prototype puede usarse si una orden debe ser copiada antes de ser guardada en el historial.

Patrón Iterator

Categoría de Patrón: *Diseño*

Categoría de Problema: *Control de Acceso*

Propósito

Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna (una especie de cursor).

Motivación

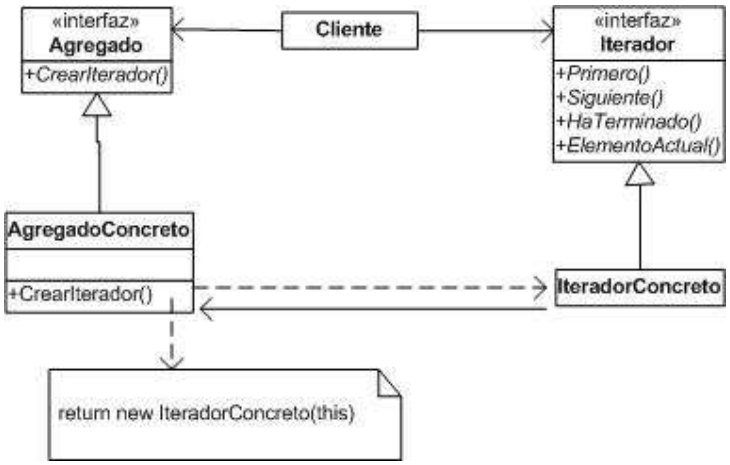
Un objeto lista debería darnos una forma de acceder a sus elementos sin exponer su estructura interna, queremos recorrerla de diferentes formas, pero no plagarla con operaciones para diferentes recorridos. Por otro lado, también puede necesitarse hacer mas de un recorrido simultáneamente sobre la lista.
El patrón Iterator nos permite hacer todo esto. La idea clave es tomar la responsabilidad de acceder y recorrer el objeto lista y poner dicha responsabilidad en un objeto Iterador que define una interfaz para acceder a los elementos de la lista y es responsable de saber cual es el elemento actual; es decir, sabe que elementos ya han sido recorridos.

Aplicabilidad

Usar Iterator:

1) Para acceder al contenido de un objeto agregado sin exponer su representación interna.
2) Para permitir varios recorridos sobre objetos agregados.
3) Para proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (es decir, para permitir la iteración polimórfica).

Estructura



Participantes

1) Iterador: Define una interfaz para recorrer los elementos y acceder a ellos.

2) **IteradorConcreto**: Implementa la interfaz de Iterador. Mantiene la posición actual en el recorrido del agregado.
3) **Agregado**: Define una interfaz para crear un objeto Iterador.
4) **AgregadoConcreto**: Implementa la interfaz de creación de Iterador para devolver una instancia del IteratorConcreto apropiado.

Colaboraciones

Un IteratorConcreto sabe cual es el objeto actual del agregado y puede calcular el objeto siguiente en el recorrido.

Consecuencias

Ventajas

5) Permite variaciones en el recorrido de un agregado: Los agregados complejos pueden recorrerse de muchas formas (ej.: recorrer árboles en en-orden o pre-orden). Los iteradores facilitan cambiar el algoritmo de recorrido, basta con sustituir la instancia de iterador por otra diferente. También se pueden definir subclases de Iterador para permitir nuevos recorridos.
6) Los iteradores simplifican la interfaz Agregado: La interfaz de recorrido de Iterador elimina la necesidad de una interfaz parecida en Agregado, simplificando así la interfaz del agregado.
7) Se puede hacer mas de un recorrido a la vez sobre un agregado: Un iterador mantiene su propio estado del recorrido. Por tanto es posible estar realzando mas de un recorrido al mismo tiempo.

Patrones relacionados

a. El patrón Composite suele usar iteradores para las estructuras recursivas como los compuestos.
b. El patrón Factory Method es usado por los iteradores polimórficos para crear instancias de las subclases apropiadas de Iterator.
c. El patrón Memento suele usarse conjuntamente con el patrón Iterator, quien usa un Memento para representar el estado de una iteración. El iterador almacena el memento internamente.

Patrón Mediator

Categoría de Patrón: *Diseño*

Categoría de Problema: *Organización del trabajo*

Propósito

Define un objeto que encapsula como interactúan una serie de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente y permite variar la interacción entre ellos de forma independiente.

Motivación

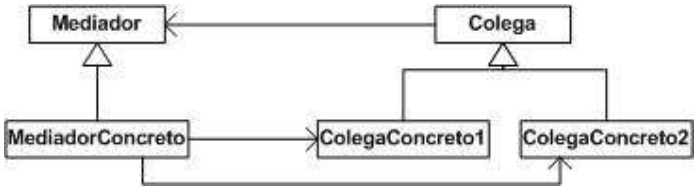
Los diseñadores orientados a objetos promueven la distribución de comportamiento entre objetos. Esto puede dar a lugar una estructura de objetos con muchas conexiones entre ellos; en el peor de los casos, cada objeto acaba por conocer a todos los demás.
Aunque dividir un sistema en muchos objetos suele mejorar la reutilización, la proliferación de interconexiones tiende a reducir esta de nuevo. Tener muchas interconexiones hace que sea menos probable que un objeto pueda funcionar sin la ayuda de otros. Mas aún, puede ser difícil cambiar el comportamiento del sistema de manera significativa, ya que el mismo se encuentra distribuido en muchos objetos. Como resultado, podemos vernos forzados a definir muchas subclases para personalizar el comportamiento del sistema.
Estos problemas pueden ser evitados encapsulando el comportamiento colectivo en un objeto aparte llamado Mediador, responsable de controlar y coordinar las interacciones entre un grupo de objetos, evitando que los mismos se refieran unos a otros explícitamente. Los objetos solo conocen al Mediador, reduciendo así el numero de interconexiones.

Aplicabilidad

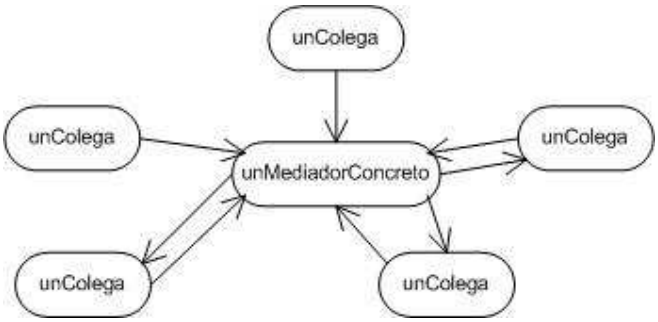
Usar Mediator cuando:

1) Un conjunto de objetos se comunican de forma bien definida, pero compleja. Las interdependencias resultantes no están estructuradas y son difíciles de comprender.
2) Es difícil reutilizar un objeto, ya que este se refiere a otros muchos objetos con los que se comunica.
3) Un comportamiento que esta distribuido entre varias clases debería poder ser adaptado sin necesidad de una gran cantidad de subclases.

Estructura



Una estructura de objetos típica



Participantes

1) **Mediador**: Define una interfaz para comunicarse con sus objetos Colega.
2) **MediadorConcreto**: Implementa el comportamiento cooperativo coordinando objetos Colega. Conoce a sus colegas.
3) **Clases Colega**: Cada clase conoce a su objeto Mediador y se comunica a este cada vez que, de no existir el Mediador, se hubiera comunicado con otro Colega.

Colaboraciones

Los Colegas envían y reciben peticiones a través de un Mediador. Este implementa el comportamiento cooperativo encaminando estas peticiones a los Colegas apropiados.

Consecuencias

1) *Reduce la herencia:* Un Mediador localiza el comportamiento que de otra manera estaría distribuido en varios objetos. Para cambiarlo solo es necesario crear una subclase del Mediador; las clases Colega pueden ser reutilizadas tal cual (Ventaja).
2) *Desacopla a los Colegas:* Un Mediador promueve un bajo acoplamiento entre Colegas. Las clases Colega pueden usarse y modificarse de forma independiente (Ventaja).
3) *Simplifica los protocolos de los objetos:* Un Mediador sustituye interacciones muchos-a-muchos por interacciones uno-a-muchos entre el objeto mediador y sus Colegas, que son más fáciles de comprender, mantener y extender (Ventaja).
4) *Abstrae como cooperan los objetos:* Hacer de la mediación un concepto independiente y encapsularla en un objeto permite centrarse en como interactúan los objetos en vez de en su comportamiento individual, ayudando a clarificar como interactúan los objetos de un sistema (Ventaja).
5) *Centraliza el control:* El patrón Mediator cambia complejidad de interacción por complejidad en el mediador, pudiendo hacerse más complejo que cualquier Colega individual, difícil de mantener (Desventaja).

Implementación

1) Omitir la clase abstracta Mediador: No es necesario definirla cuando los colegas solo trabajan con un Mediador.
2) Comunicación Colega-Mediador: Los Colegas tienen que comunicarse con su mediador cuando tiene lugar un evento de interés. Un enfoque es implementar el Mediador usando el patrón Observer. Las clases Colega envían notificaciones al mediador cada vez que cambia su estado y este responde propagando los efectos del cambio a otros Colegas

Patrones relacionados

1) El patrón Facade difiere del Mediator en que abstrae un subsistema de objetos para proporcionar una interfaz más conveniente. Su protocolo es unidireccional (los objetos Facade hacen peticiones al subsistema pero no a la inversa). Por el contrario, Mediator permite un comportamiento cooperativo que no es proporcionado por los objetos Colegas y el protocolo es multidireccional.
2) El patrón Observer se usa para que los Colegas se comuniquen con el Mediador puede usarse si una orden debe ser copiada antes de ser guardada en el historial.

Patrón Observer

Propósito

Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de el.

Motivación

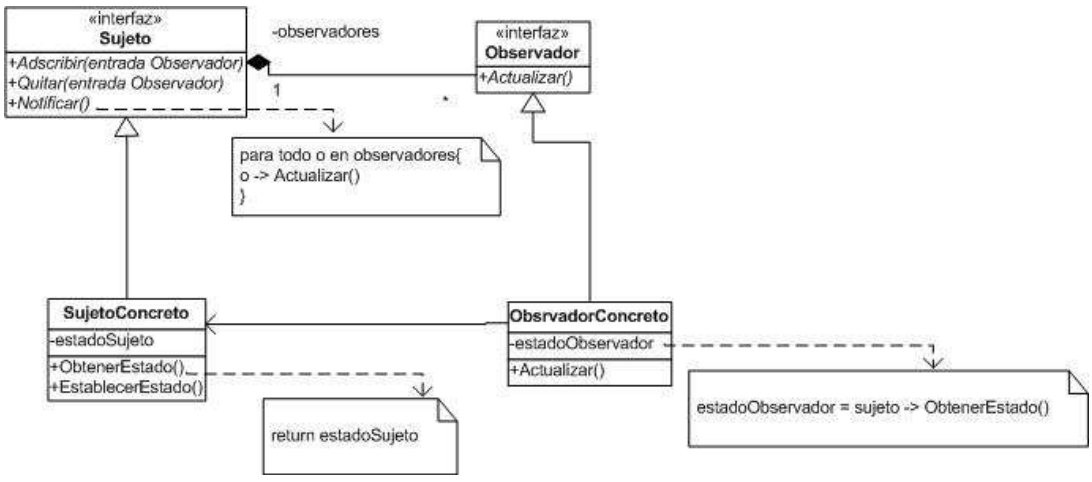
Un efecto lateral habitual de dividir un sistema en una colección de clases cooperantes es la necesidad de mantener una consistencia entre objetos relacionados, pero sin hacer a las clases fuertemente acopladas ya que eso reduciría su reutilización.
El patrón Observer describe como establecer estas relaciones. Los principales objetos de este patrón son el sujeto y el observador. Un sujeto puede tener cualquier numero de observadores dependientes de él. Cada vez que el sujeto cambia su estado se notifica a todos sus observadores. En respuesta, cada observador consultara al sujeto para sincronizar su estado con el estado de este.
Este tipo de interacción también se conoce como publicar-suscribir. El sujeto es quien publica las notificaciones. Envía estas sin tener que conocer quienes son sus observadores. Pueden suscribirse un numero indeterminado de observadores para recibir notificaciones.

Aplicabilidad

Usar Observer cuando:

1) Una abstracción tiene dos aspectos y uno depende del otro. Encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
2) Cuando un cambio en un objeto requiere cambiar otros y no sabemos cuantos objetos necesitan cambiarse.
3) Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quienes son dichos objetos (no queremos que estos objetos estén fuertemente acoplados).

Estructura



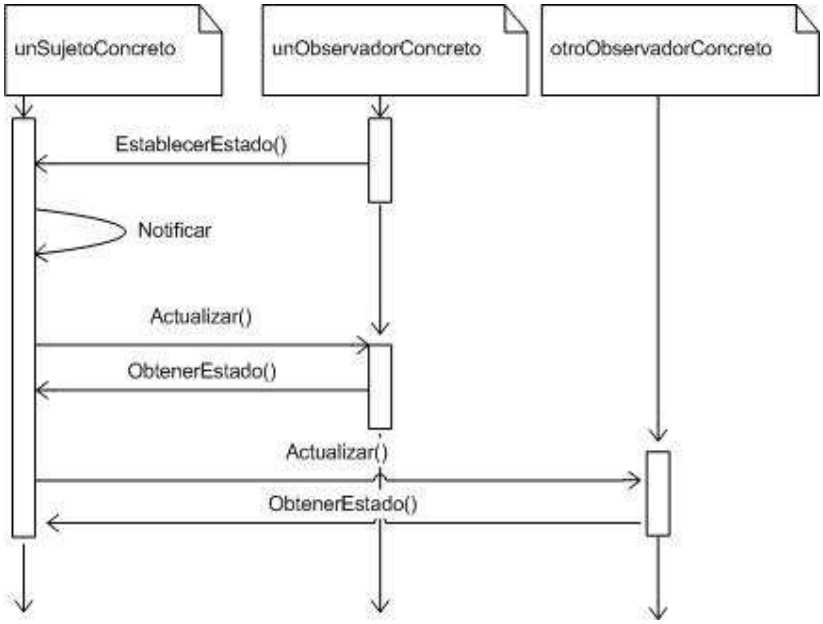
Participantes

1) **Sujeto**: Conoce a sus observadores. Un sujeto puede ser observado por cualquier numero de objetos Observador. Proporciona una interfaz para asignar y quitar objetos Observador.
2) **Observador**: Define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto.
3) **SujetoConcreto**: Almacena el estado de interés para los objetos ObservadorConcreto. Envía una notificación a sus observadores cuando cambia su estado.
4) **ObservadorConcreto**: Mantiene una referencia a un SujetoConcreto. Guarda un estado que debería ser consistente con el del sujeto. Implementa la interfaz de actualización del Observador para mantener su estado consistente con el del sujeto.

Colaboraciones

1) SujetoConcreto notifica a sus observadores cada vez que se produce un cambio que pudiera hacer que el estado de estos fuera inconsistente con el suyo.
2) Después de ser informado de un cambio en el SujetoConcreto, un objeto ObservadorConcreto puede pedirle al sujeto más información. ObservadorConcreto usa esta información para sincronizar su estado con el del sujeto.

El siguiente diagrama de interacción muestra las colaboraciones:



El objeto Observador que inicializa la petición de cambio pospone su actualización hasta que obtiene una notificación del sujeto. Notificar no siempre es llamado por el sujeto. Puede ser llamado por el Observador o por un tipo de objeto completamente diferente.

Consecuencias

El patrón Observer permite modificar los sujetos y observadores de forma independiente. Es posible reutilizar objetos sin reutilizar sus observadores y viceversa, permitiendo añadir observadores sin modificar el sujeto u otros observadores.

1) *Acoplamiento abstracto entre Sujeto y Observador:* Todo lo que un sujeto sabe es que tiene una lista de observadores que se ajusta a la interfaz simple de la clase abstracta Observador. El sujeto no conoce la clase concreta de ningún observador. Por lo tanto el acoplamiento entre sujetos y observadores es mínimo, pueden pertenecer a diferentes capas de abstracción de un sistema (Ventaja).
2) *Capacidad de comunicación mediante difusión:* A diferencia de una petición ordinaria, la notificación enviada por un sujeto no necesita especificar su receptor, se envía automáticamente a todos los objetos interesados que se hayan suscripto a ella. Al sujeto no le importa cuantos objetos interesados haya; su única responsabilidad es notificar a sus observadores (libertad de añadir y quitar observadores en cualquier momento) (Ventaja).
3) *Actualizaciones inesperadas:* Dado que los observadores no saben de la presencia de los otros, pueden no saber el coste último de cambiar el sujeto. Una operación aparentemente inofensiva sobre el sujeto puede dar lugar a una serie de actualizaciones en cascada de los observadores y sus objetos dependientes (Desventaja).

Usos conocidos

El patrón Observer aparece en el Modelo Vista Controlador (MVC). La clase Modelo desempeña el papel del Sujeto, mientras que Vista es la clase de los Observadores.

Patrón State

Categoría de Patrón: *Diseño*

Categoría de Problema: *Variación de servicios*

Propósito

Permite que un objeto modifique su comportamiento cada vez que cambie su estad interno. Parecerá que cambia la clase del objeto.

Problema

Las instancias se comportan de distinta manera en función del estado.

Motivación

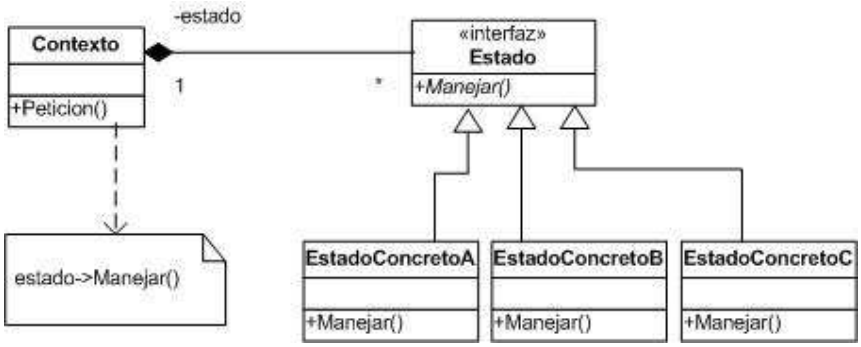
La idea clave es introducir una clase abstracta “Estado” que representa todos los estados declara una interfaz común para todas las clases que representan diferentes estados operacionales. Las subclases implementan comportamiento específico de cada estado.
La clase “Contexto” mantiene una instancia de una subclase de “Estado” que representa el estado actual, y delega todas las peticiones dependientes del estado a este objeto de estado, es decir, usa esta instancia para realizar operaciones que dependen del estado del “Contexto”.
Cada vez que cambia el estado del contexto, cambia la instancia del objeto estado que este usa.

Aplicabilidad

Usar cuando:

1) El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.
2) Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Muchas veces son varias las operaciones que contienen esta misma estructura condicional. El patrón State pone cada rama de la condición en una clase aparte. Permite tratar al estado del objeto como un objeto que puede variar independientemente de otros objetos.

Estructura



Participantes

1) **Contexto**: Define la interfaz de interés para los clientes. Mantiene una instancia de una subclase de EstadoConcreto que define el estado actual.
2) **Estado**: Define una interfaz para encapsular el comportamiento asociado con un determinado estado del Contexto.
3) **Subclases de EstadoConcreto:** Cada una implementa un comportamiento asociado con un estado del Contexto.

Colaboraciones

Contexto delega las peticiones que dependen del estado en el objeto EstadoConcreto actual.
Un contexto puede pasarse a sí mismo como parámetro para que el objeto Estado maneje la petición. Esto permite al objeto Estado acceder al contexto si fuera necesario.
Contexto es la interfaz principal para los clientes. Los clientes pueden configurar un contexto con objetos Estado. Una vez que esta configurado el contexto, sus clientes ya no tienen que tratar con los objetos Estado directamente.
Cualquiera de las subclases de Contexto o de EstadoConcreto pueden decidir que estado sigue a otro y bajo que circunstancias.

Consecuencias

3) Sitúa en un objeto todo el comportamiento asociado con un determinado estado. Como todo el código dependiente del estado reside en una subclase de Estado, pueden añadirse fácilmente nuevos estados y transiciones definiendo nuevas subclases. (Ventaja).
4) Evita el problema de tener sentencias condicionales repartidas por toda la implementación de Contexto, pero de esta manera incrementa el número de clases y es menos compacto que una única clase. Dicha distribución es realmente buena si hay muchos estados, que de otro modo necesitarían grandes sentencias condicionales (hay que tratar de evitarlas, ya que tienden a hacer el código menos explicito, difícil de modificar y extender) (Ventaja).
5) Ofrece un modo mejor de estructurar el código dependiente del estado. La lógica que determina las transiciones entre estados se reparte entre las subclases de Estado. Al encapsular cada transición y acción en una clase estamos elevando la idea de un estado de ejecución a objetos de estado en toda regla (Ventaja).
6) En caso de que los objetos Estado no tengan variables, entonces varios contextos pueden compartir un mismo objeto Estado (Ventaja)

Implementación

4) ¿Quién define las transiciones entre estados? El patrón no lo especifica. Si estos criterios son fijos, pueden implementarse enteramente en el Contexto, pero es generalmente más flexible y conveniente que sean las propias subclases de Estado quienes especifiquen su estado sucesor y cuando llevar a cabo la transición (requiere agregar una interfaz al Contexto que permita a los objetos Estado asignar el estado actual del Contexto). Descentralizar de esta forma la lógica facilita modificar o extender dicha lógica definiendo nuevas subclases de Estado, la desventaja es que una subclase de Estado conocerá al menos a otra, lo que introduce dependencias de implementación entre subclases.
5) Alternativa basada en tablas: Por cada estado, una tabla hace corresponder cada posible entrada con un estado sucesor. Este enfoque convierte código en una tabla de búsqueda. El patrón modela el comportamiento específico del estado, mientras que el enfoque basado en una tabla se centra en definir las transiciones de estado.

Ventaja: Se pueden cambiar los criterios de transición modificando datos en vez del código del programa.

Desventajas

a. Una tabla de búsqueda es normalmente menos eficiente que una llamada a una función.
b. Situar la lógica de transición en un formato tabular uniforme hace que los criterios de transición sean más difíciles de comprender.
c. Suele dificultar añadir acciones que acompañen a las transiciones de estado.

Similitudes con otros patrones

Si bien estructuralmente son casi idénticos el patrón State con el patrón Strategy, los fines para los que fueron creados no lo son y ahí radica su diferencia.
El Strategy define familias de algoritmos para cambiar el comportamiento dinámicamente de la clase que los utiliza, pero es esta clase o el cliente el que especifica que algoritmo usar. Además, por lo general, los algoritmos persiguen el mismo fin, y la diferencia entre ellos radicaría en la forma en que se implementan para obtener un resultado diferente. Ejemplo, para un contexto que se dedica a ordenar, podría tener una estrategia que lo haga con el algoritmo de “Selección”, otra que lo haga con el de “Burbujeo” y otra que lo haga por el de “Inserción”.
El State modifica dinámicamente el comportamiento de una clase, en función de un cierto “estado” (de acuerdo al estado que tenga varía la tarea que voy a realizar). Además, la lógica de transición de estados reside generalmente en las subclases concretas de Estado, por lo que es transparente para el cliente la utilización de los mismos.

Ejemplos

Se puede usar el patrón State para una clase que modele la conexión TCP de una red, donde los distintos estados podrían ser: Establecida, Escuchando, Cerrada, etc.

Patrones Relacionados

Los objetos estados muchas veces son Singletons

Patrón Strategy

Categoría de Patrón: *Diseño*

Categoría de Problema: *Variación de servicios*

Propósito

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Problema

Posibilidad de cambiar el comportamiento en forma dinámica.

Motivación

Si existen muchos algoritmos, codificarlos en las clases que los usan no resulta una buena práctica por:

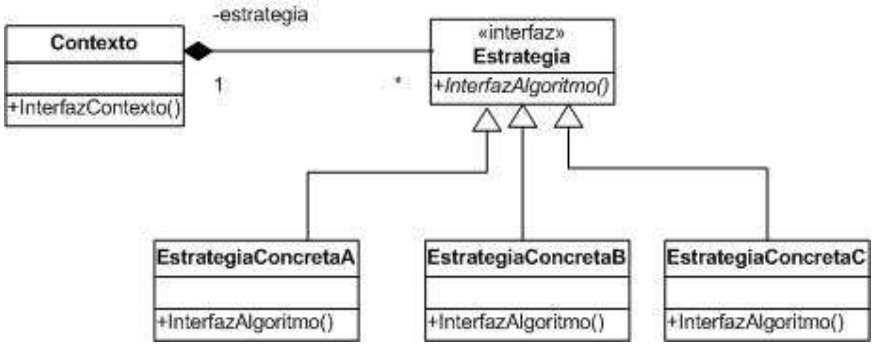
1) Los clientes se vuelven más complejos, más grandes y más difíciles de mantener si incluyen dicho código.
2) Los distintos algoritmos serán apropiados en distintos momentos, no hay que permitir múltiples algoritmos si no los vamos a usar todos.

Aplicabilidad

Usar cuando:

1) Muchas clases relacionadas difieren solo en su comportamiento (permiten configurar una clase con un determinado comportamiento de entre muchos posibles)
2) Se necesitan distintas variantes de un algoritmo
3) Un algoritmo usa datos que los clientes no deberían conocer (para evitar exponer estructuras de datos complejas y dependientes del algoritmo)
4) Una clase define muchos comportamientos y estos se representan como múltiples sentencias condicionales en sus operaciones. En vez de esto, usamos una clase estrategia para cada comportamiento.

Estructura



Participantes

1) **Estrategia:** Declara una interfaz común a todos los algoritmos permitidos. El Contexto usa esta interfaz para llamar al algoritmo definido por una EstrategiaConcreta.
2) **EstrategiaConcreta**: Implementa el algoritmo usando la interfaz Estrategia
3) **Contexto**: Se configura con un objeto EstrategiaConcreta. Mantiene una referencia a un objeto Estrategia. Puede definir una interfaz que permite a la Estrategia acceder a sus datos.

Colaboraciones

Un Contexto puede pasar a la Estrategia todos los datos requeridos por el algoritmo cada vez que se llama a este (menor acoplamiento entre el contexto y las estrategias, pero podría estar pasando datos de mas, que alguna Estrategia no necesitaría).
Otra alternativa es que el Contexto se pase a sí mismo como argumento de las operaciones de Estrategia (mayor acoplamiento entre clases).
Los clientes normalmente crean un objeto EstrategiaConcreta y se lo pasan al Contexto. Los clientes interactúan exclusivamente con el Contexto.

Consecuencias

1) *Es una alternativa a la herencia*: Encapsular el algoritmo en clases Estrategia separadas nos permite variar el algoritmo independientemente de su contexto, haciéndolo más fácil de cambiar, comprender y extender. (Ventaja)
2) *Las estrategias eliminan las sentencias condicionales:* Un código con muchas sentencias condicionales suele indicar la necesidad de aplicar este patrón. (Ventaja)
3) *Una elección de implementación:* Proporcionan distintas implementaciones de un mismo comportamiento. (Ventaja)
4) *Los clientes deben conocer las distintas estrategias antes de seleccionar la adecuada:* Debería usarse solo cuando la variación de comportamiento sea relevante a los clientes. (Desventaja)
5) *Costes de comunicación entre Estrategia y Contexto:* La interfaz Estrategia es compartida por todas las clases EstrategiaConcreta, siendo probable que algunos objetos EstrategiaConcreta no usen toda la información que reciben, por lo que el Contexto estaría creando e inicializando parámetros que nunca se usarían (Desventaja)
6) Mayor numero de objetos (Desventaja)

Patrón Template Method

Categoría de Patrón: *Idiom*

Categoría de Problema: *Variación de servicios*

Propósito

Define en una operación el esqueleto de un algoritmo, delegando en las subclases alguno de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

Problema

Permite definir un algoritmo y redefinir alguno de sus pasos

Motivación

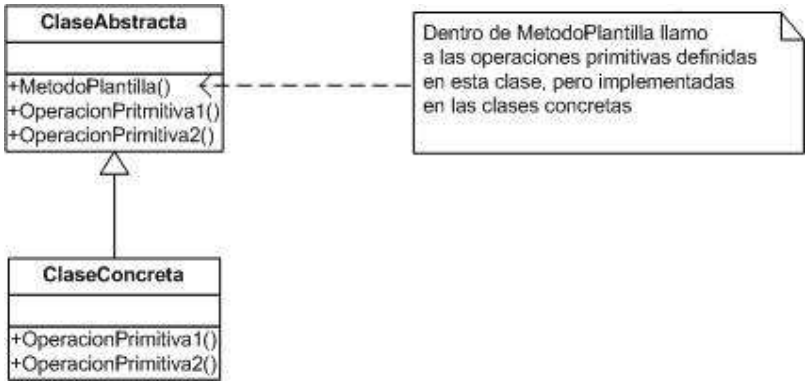
Define un algoritmo en termino de operaciones abstractas que las subclases deben redefinir para proporcionar un determinado comportamiento.

Aplicabilidad

Debería usarse

1) Para implementar las partes de un algoritmo que no cambia y dejar que sean las subclases quienes implementen el comportamiento que puede variar.
2) Cuando el comportamiento repetido de varias subclases debería factorizarse y ser localizado en una clase común para evitar el código duplicado. Refactorizar para generalizar: identificamos las diferencias en el código existente y a continuación separamos dichas diferencias en nuevas operaciones, luego sustituimos el código que cambia por un metodo que llama a una de estas nuevas operaciones.
3) Para controlar las extensiones de las subclases. Podemos definir un método plantilla que llame a operaciones “de enganche” (véase Consecuencias) en determinados puntos, permitiendo así las extensiones solo en esos puntos.

Estructura



Participantes

1) **ClaseAbstracta:** Define operaciones primitivas abstractas que son definidas por las subclases para implementar los pasos de un algoritmo. Implementa un método plantilla que define el esqueleto de un algoritmo. El método plantilla llama a las operaciones primitivas así como a operaciones definidas ClaseAbstracta o a las de otros objetos.
2) **ClaseConcreta:** Implementa las operaciones primitivas para realizar los pasos del algoritmo específicos de las subclases.

Colaboraciones

ClaseConcreta se basa en ClaseAbstracta para implementar los pasos de un algoritmo que no cambia.

Consecuencias

Los métodos de plantilla deben especificar que operaciones son de enganche (que *pueden* ser redefinidas) y cuales son operaciones abstractas (que *deben* ser redefinidas). Los escritores de las subclases deben saber que operaciones están diseñadas para ser redefinidas.
Una subclase puede extender el comportamiento de una operación de una clase padre redefiniendo la operación y llamando explícitamente a la operación del padre:

```
void ClaseDerivada::Operación(){
      // ClaseDerivada extiende el comportamiento
      ClasePadre::Operacion();
}
```

Pero es fácil olvidarse de llamar a la operación heredada. Podemos transformar esta operación en un método plantilla que le dé control al padre sobre como este puede ser extendido por las subclases (llamar a una operación de enganche desde un método plantilla en la clase padre y que las subclases redefinan esta operación de enganche)

```
void ClasePadre::Operación(){
      // Comportamiento de ClasePadre
      OperacionDeEnganche();
}
```

OperacionDeEnganche no hace nada en ClasePadre

```
void ClasePadre::OperacionDeEnganche(){}
```

Las subclases redefinen OperacionDeEnganche para extender su comportamiento

```
void ClaseDerivada::OperacionDeEnganche(){
      // Extensión de la clase derivada
}
```

Ejemplo

Frameworks.

Patrones Relacionados

Los métodos de fabricación se llaman muchas veces desde métodos plantilla.
El patrón Strategy usa delegación para variar el algoritmo completo. El patrón Template Method usa la herencia para modificar una parte de un algoritmo.

Patrón Visitor

Categoría de Patrón: *Diseño*

Categoría de Problema: *Extensión de servicios*

Propósito

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

Motivación

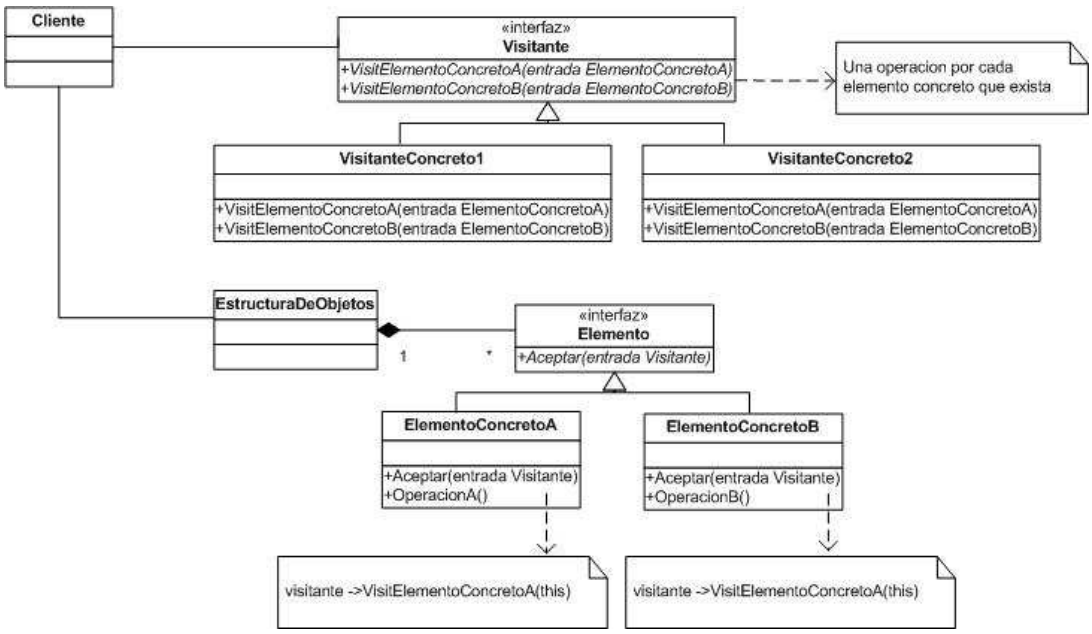
Sumar funcionalidades asociadas a clases ya definidas (sin modificar estas clases).

Aplicabilidad

Usar cuando:

1) Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta.
2) Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos, y queremos evitar "contaminar" sus clases con dichas operaciones. Permite mantener juntas operaciones relacionadas definiéndolas en una clase, ej.: cuando la estructura de objetos es compartida por varias aplicaciones, permite poner operaciones solo en aquellas aplicaciones que las necesiten.

Estructura

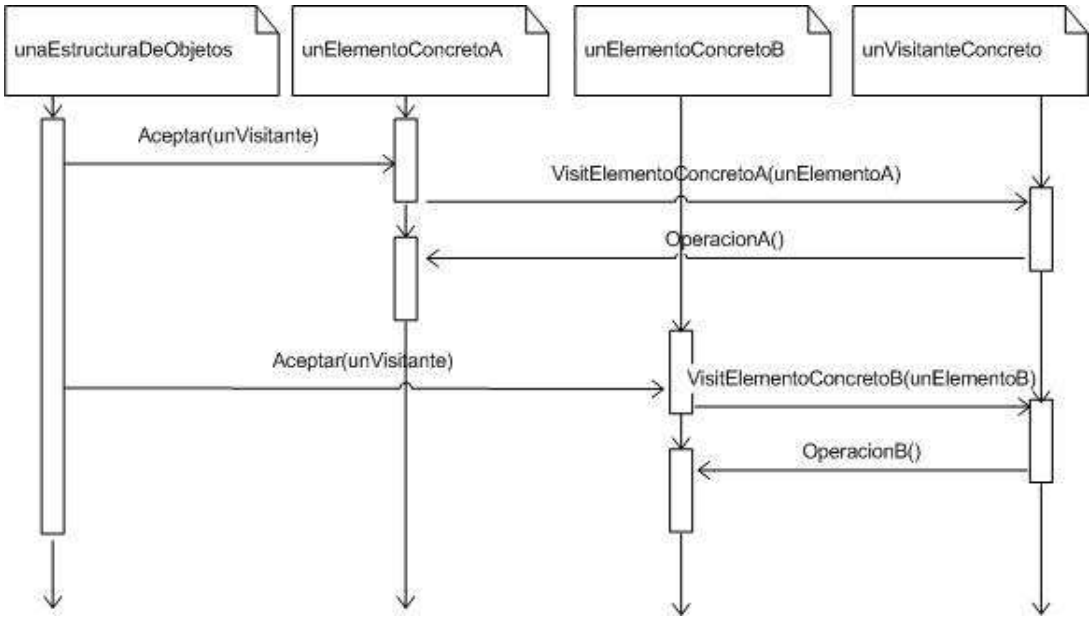


Participantes

1) **Visitante**: Declara una operación Visitar para cada clase de operación ElementoConcreto de la estructura de objetos. El nombre (ej.: VisitElementoConcretoA) de la operación identifica a la clase que envía la petición Visitar al visitante (permite al visitante determinar la clase concreta de elemento que esta siendo visitada y acceder al elemento directamente a través de su interfaz particular.
2) **VisitanteConcreto:** Implementa cada operación declarada por Visitante. Cada operación implementa un fragmento del algoritmo que definido para la clase correspondiente de la estructura. VisitanteConcreto proporciona el contexto para el algoritmo y guarda su estado local.
3) **Elemento**: Define una operación Aceptar que toma un visitante como argumento.
4) **ElementoConcreto**: Implementa una operación Aceptar que toma un visitante como argumento.
5) **EstructuraDeObjetos**: Puede enumerar sus elementos. Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos.

Colaboraciones

Un cliente que usa el patrón Visitor debe crear un objeto VisitanteConcreto y a continuación recorrer la estructura, visitando cada objeto con el visitante.
Cada vez que se visita a un elemento, este llama a la operación del Visitante que se corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario.



Consecuencias

1) *El visitante facilita añadir nuevas operaciones*: Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante. Si, por el contrario,

extendiésemos la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación *(Ventaja).*

2) *Un visitante agrupa operaciones relacionadas y separa las que no lo están:* Las partes de comportamiento no relacionadas se dividen en sus propias subclases del visitante. Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes. Cualquier estructura de datos específica de un algoritmo puede estar oculta en el visitante *(Ventaja).*
3) *Acumular el estado:* Los visitantes pueden acumular el estado a medida que van visitando cada elemento de la estructura de objetos. Sin un visitante, este estado se pasaría como argumentos extra a las operaciones que realizan el recorrido, o quizá como variables globales *(Ventaja).*
4) *Es difícil añadir nuevas clases de ElementoConcreto:* Cada ElementoConcreto nuevo da lugar a una nueva operación abstracta del Visitante y a su correspondiente implementación en cada clase VisitanteConcreto, por lo que, la cuestión fundamental a considerar la hora de aplicar el patrón es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen la estructura. Si la jerarquía de clases de Elemento es estable pero estamos continuamente añadiendo operaciones o cambiando algoritmos, el patrón Visitor nos ayudará a controlar dichos cambios *(Desventaja)*.
5) *Romper la encapsulación:* El enfoque del patrón asume que la interfaz de ElementoConcreto es lo bastante potente como para que los visitantes hagan su trabajo. Obliga a proporcionar operaciones públicas que accedan al estado interno de un elemento, lo que puede comprometer su encapsulación *(Desventaja)*

Conclusión

El patrón Visitor nos permite añadir operaciones a clases sin modificarlas.
La clave del patrón es que la operación que se ejecuta depende tanto del tipo del visitante como del tipo del elemento visitado. En vez de enlazar operaciones estáticamente en la interfaz de Elemento, podemos fusionar operaciones en un Visitante y usar Aceptar para hacer el enlace en tiempo de ejecución. Extender la interfaz de Elemento consiste en definir una nueva subclase de Visitante en vez de muchas subclases de Elemento.

Patrones relacionados

1) *Composite:* Los visitantes pueden usarse para aplicar una operación sobre una estructura de objetos definidos por el Composite.
2) *Interpreter:* Se puede aplicar el patrón Visitor para llevar a cabo la interpretación.

Conclusión

Encapsular lo que varía

Encapsular aquello que puede variar es el tema de muchos patrones de comportamiento:

1) Un objeto Estrategia encapsula un algoritmo.
2) Un objeto Estado encapsula un comportamiento dependiente del estado.
3) Un objeto Mediador encapsula el protocolo entre objetos.
4) Un objeto Iterador encapsula el modo en que se accede y se recorren los componentes de un objeto agregado.

Pero no todos los patrones de comportamiento de un objeto dividen así la funcionalidad.
El patrón Chain of Responsability describe el modo de comunicación entre un numero indefinido de objetos. Otros patrones usan objetos que se pasan como argumentos.
El Mediator y el Observer son patrones rivales. La diferencia entre ellos es que el patrón Observer distribuye la comunicación introduciendo objetos Observador y Sujeto, mientras que un objeto Mediador encapsula la comunicación entre otros objetos.

Desacoplar Emisores y Receptores

Cuando los objetos que colaboran se refieren unos a otros explícitamente, se vuelven dependientes unos de otros y eso puede tener un impacto adverso sobre la división en capas y la reutilización de un sistema. Los patrones Command, Observer, Mediator y Chain of Responsability tratan el problema de cómo desacoplar emisores y receptores, cada uno con sus ventajas e inconvenientes:

1) El patrón Command permite el desacoplamiento usando un objeto Orden que define un enlace entre un emisor y un receptor. El objeto Orden proporciona una interfaz simple para emitir la petición (operación Ejecutar). Definir la conexión emisor-receptor en un objeto aparte permite que el emisor funcione con diferentes receptores. Gracias a mantener al emisor desacoplado de los receptores es más fácil reutilizar los emisores.
2) El patrón Observer desacopla a los emisores (sujetos) de los receptores (observadores) definiendo una interfaz para indicar cambios en los sujetos. Define un enlace más débil que Command entre el emisor y el receptor, ya que un sujeto puede tener múltiples observadores, cuyo numero puede variar en tiempo de ejecución. Las interfaces Sujeto y Observador están diseñadas para posibles cambios en la comunicación. Por tanto, el patrón Observer es mejor para desacoplar objetos cuando hay dependencias de datos entre ellos.
3) El patrón Mediator desacopla los objetos haciendo que se refieran unos a otros indirectamente, a través del objeto Mediador, quien encamina peticiones entre objetos Colega y centraliza su comunicación. En consecuencia, los colegas solo pueden hablar entre sí a través de la interfaz del Mediador. El patrón Mediador puede reducir la herencia en un sistema, al centralizar el comportamiento de comunicación en una clase en vez de distribuirlo entre las subclases.
4) El patrón Chain of Responsability desacopla al emisor del receptor pasando la petición a lo largo de una cadena de receptores potenciales. La Cadena de Responsabilidad es un buen modo de desacoplar el emisor y el receptor en caso de que la cadena ya forme parte de la estructura del sistema y haya uno o varios objetos capaces de manejar la petición (la cadena puede cambiarse o ampliarse fácilmente).

Resumen de Patrones

Patrones de Creación

Patrón	Propósito
Singleton	Garantiza que una sola clase sólo tenga una instancia y proporciona un punto de acceso global a ella
Factory Method	Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan que clases instanciar. Permite que una clase delegue en sus subclases la creación de objetos (Constructor Virtual)
Abstract Factory	Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas
Prototype	Especifica los tipos de objetos a crear por medio de una instancia prototípica y crea nuevos objetos copiando dicho prototipo
Builder	Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas

Patrones Estructurales

Patrón	Propósito
Adapter	Adaptar para facilitar implementación
Bridge	Desacopla una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.
Composite	Que el cliente vea a las partes y al todo como una misma cosa. Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
Decorator	Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
Facade	Proporcionar una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
Proxy	Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.

Patrones de Comportamiento

Patrón	Propósito
Chain of Responsibility	Evita acoplar el emisor de una petición a su receptor, dando a mas de un objeto la posibilidad de responder a la petición. Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.
Command	Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones y poder deshacer las operaciones.
Iterator	Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna (una especie de cursor).
Mediator	Define un objeto que encapsula como interactúan una serie de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente y permite variar la interacción entre ellos de forma independiente.
Observer	Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de el.
State	Permite que un objeto modifique su comportamiento cada vez que cambie su estad interno. Parecerá que cambia la clase del objeto.
Strategy	Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
Template Method	Define en una operación el esqueleto de un algoritmo, delegando en las subclases alguno de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
Visitor	Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.