

Patrones de Diseño en Programación Orientada a Objetos

Una introducción a una selección de
patrones

Jorge Puente Peinador

14/10/2011

En este documento se describen varios de los patrones de diseño utilizados en POO. El contenido del documento está en gran parte tomado de la página web www.sourcemaking.com y del libro GoF, más algunas aportaciones del autor del documento

Fuente

El contenido de este documento está basado principalmente en el contenido sobre Patrones de Diseño, en programación orientada a objetos, de la página Web SourceMaking:

http://sourcemaking.com/design_patterns

En gran parte es una traducción del contenido de dicha Web, pero al texto se le han añadido algunos comentarios, aclaraciones, y figuras donde se ha considerado necesario.

En la página web original se pueden encontrar además ejemplos en código fuente, en diferentes lenguajes de programación orientados a objeto, en los que se aplica de una manera más o menos clara cada patrón de diseño.

Más patrones

En este documento sólo se recogen algunos de los patrones de diseño de cada categoría, en la página Web de SourceMaking se puede encontrar información similar del resto de patrones.

La fuente bibliográfica de referencia es el libro denominado GoF (por Gang of Four, la banda de los cuatro, haciendo referencia a sus cuatro autores):

Patrones de diseño "Elementos de software orientado a objetos reutilizable". Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. Ed. Addison Wesley Publishers, 2002.

Patrones de Diseño

En ingeniería de software, un patrón de diseño es una solución repetible a un problema que se presenta habitualmente en el diseño de software. Un patrón de diseño no es un diseño completamente finalizado que pueda ser transformado directamente en código. Es una descripción o plantilla *de cómo solucionar un problema* que puede ser usada en situaciones muy distintas.

Usos de los Patrones de Diseño

Los patrones de diseño facilitan el proceso de desarrollo de software, proporcionando paradigmas de desarrollo probados y testeados. Un diseño de software eficaz requiere tomar en consideración aspectos que pueden hacerse visibles después de la implementación. La reutilización de patrones de diseño ayuda a prevenir problemas sutiles que pueden causar problemas mayores y para mejorar la legibilidad del código por programadores y arquitectos familiarizados con los patrones.

A menudo, los desarrolladores sólo entienden cómo aplicar determinadas técnicas de diseño de software a ciertos problemas. Estas técnicas son difíciles de aplicar a una variedad más amplia de problemas. Los patrones de diseño proporcionan soluciones generales, documentadas y en un formato que no está necesariamente ligado a un problema concreto.

Además, los patrones permiten a los desarrolladores comunicarse entre ellos utilizando nomenclaturas de sobra conocidas para las interacciones entre elementos software. Los

patrones de diseño de uso habitual pueden mejorarse con el tiempo, logrando ser más robustos que los diseños ad-hoc.

Clasificación de los patrones de diseño

Patrones de Diseño Creacionales

Estos patrones de diseño se centran en la instanciación de clases. Este patrón puede dividirse a su vez en patrones de *creación de clases* y patrones de *creación de objetos*. Mientras que los patrones de *creación de clases* usan de manera efectiva la herencia para el proceso de instanciación, los patrones de creación de objetos usan la delegación¹ como método efectivo para realizar su trabajo.

Abstract Factory

Crea una instancia de varias familias de clases.

Builder

Separa el proceso de construcción de un objeto de su representación.

Factory Method

Crea una instancia de varias clases derivadas

Object Pool

Evita una costosa adquisición y liberación de recursos mediante el reciclado de objetos que ya no están en uso.

Prototype

Una instancia completamente inicializada que debe ser copiada o clonada.

Singleton

Una clase para la cual sólo puede existir una instancia.

Patrones de Diseño Estructurales

Estos patrones de diseño se centran en la composición de Clases y Objetos. Los patrones estructurales de *creación de clases* usan la herencia para componer interfaces. Los *patrones de objeto* estructurales definen formas de componer objetos para obtener nuevas funcionalidades.

Adapter

Adapta interfaces de diferentes clases.

Bridge

Separa la interfaz de un objeto de su implementación.

Composite

Una estructura de árbol de objetos simples y compuestos.

¹ Por delegación entendemos que una acción no la realizamos en el propio objeto, sino que utilizamos el método de un segundo objeto para realizarla, es decir, delegamos en este segundo objeto la responsabilidad de ejecutar esa acción.

Decorator

Añade responsabilidades a objetos de forma dinámica.

Facade

Una sola clase que representa un subsistema entero.

Flyweight

Una instancia de *grano fino* utilizada para un intercambio eficiente.

Private Class Data

Restringe el acceso de observadores/modificadores.

Proxy

Un objeto que representa a un objeto distinto.

Patrones de Diseño de Comportamiento

Estos patrones de diseño se centran en clases para la comunicación entre objetos. Los patrones de comportamiento son aquellos patrones que más concretamente están referidos a la comunicación entre objetos.

Chain of responsibility

Una forma de pasar una petición entre una serie de objetos encadenados.

Command

Encapsula una petición de una orden como un objeto.

Interpreter

Una forma de incluir elementos del lenguaje en un programa.

Iterator

Accede secuencialmente a los elementos de una colección.

Mediator

Define un mecanismo de comunicación simplificada entre objetos.

Memento

Captura y restaura el estado interno de un objeto.

Null Object

Diseñado para actuar como un valor por defecto para un objeto.

Observer

Una forma de notificar cambios a varias clases.

State

Modifica el comportamiento de un objeto cuando su estado cambia.

Strategy

Encapsula un algoritmo dentro de una clase.

Template method

Aplaza los pasos exactos de un algoritmo a una subclase.

Visitor

Define una nueva operación en una clase sin modificarla.

Abstract Factory – Patrón de Diseño Creacional

Motivación

Proporcionar una interfaz para crear familias de objetos relacionados o interdependientes sin especificar sus clases concretas.

Una jerarquía que encapsula: muchas posibles “plataformas”, y la construcción de una serie de productos.

Cuando el operador **new** se considera peligroso.

Problema

Si una aplicación tiene que ser portable, necesita encapsular las dependencias referidas a la plataforma. Estas “plataformas” pueden incluir: el sistema de ventanas, el sistema operativo, base de datos, etc. Demasiado a menudo, esta encapsulación no se diseña adecuadamente de antemano, como consecuencia en el código aparecen como setas un montón de directivas condicionales de compilación (sentencias del tipo `#ifdef`) referidas a las distintas plataformas en las que se puede compilar el programa.

Argumentación

Proporcionan un procedimiento indirecto para abstraer la creación de familias de objetos interdependientes o relacionados entre sí, sin especificar las clases concretas a instanciar. El objeto “factoría” tiene la responsabilidad de proporcionar servicios de creación para todas las posibles plataformas. Los clientes nunca crearán directamente objetos para una plataforma concreta, en su lugar pedirán a la factoría que haga esa labor por ellos.

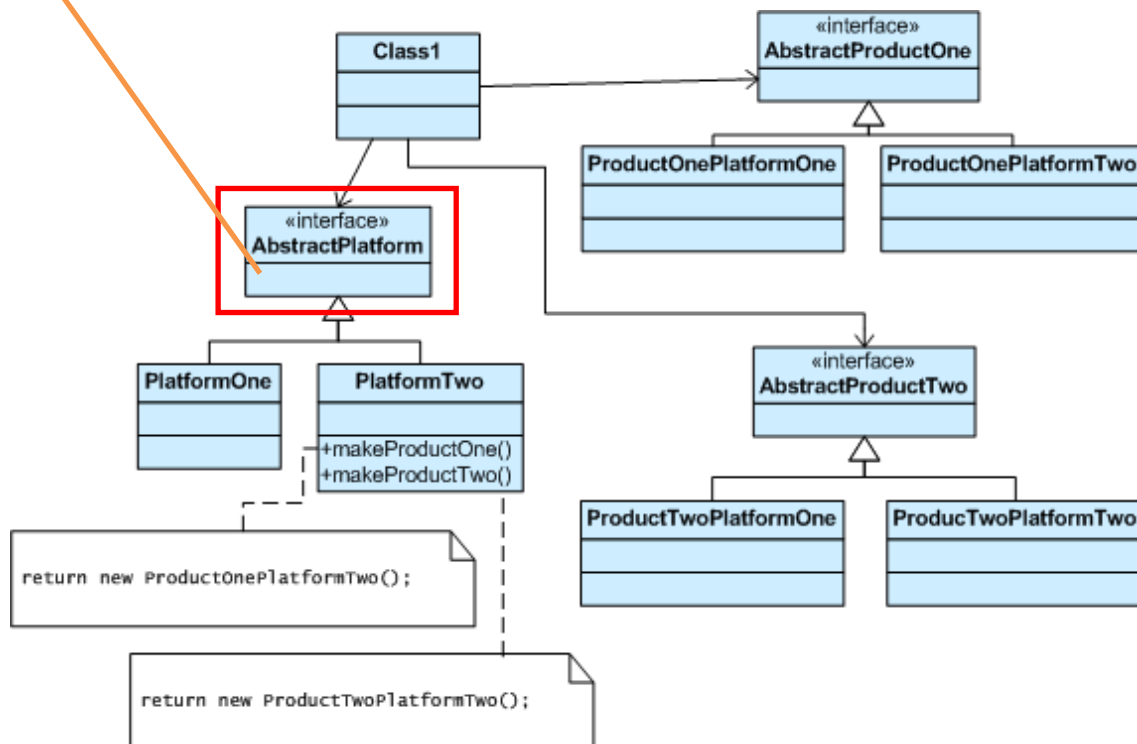
Este mecanismo permite intercambiar familias de productos fácilmente ya que la clase del objeto factoría sólo aparece una vez en el código – en el punto donde es instanciado. La aplicación puede reemplazar totalmente la familia de productos a generar con sólo instanciar un objeto diferente de la *abstract factory*.

Debido a que el servicio proporcionado por el objeto factoría está tan centralizado, se le suele implementar como un **Singleton**.

Estructura

La Abstract Factory define un *Método Fábrica* por producto. Cada *Método Fábrica* encapsula el operador **new** y la clase específica adecuada para la plataforma destino. Cada “plataforma” es por tanto modelada como una clase derivada de Factory.

AbstractProductOne +makeProductOne()
AbstractProductTwo +makeProductTwo()



En el ejemplo un objeto de la clase Class1 necesita crear dos objetos: uno de tipo ProductOne y otro de tipo ProductTwo. Para ello en primer lugar se define la “Abstract Factory” específica de la plataforma con la que quiere trabajar (en el ejemplo la PlataformaTwo) a través de una interface AbstractPlatform. Una vez creada se invoca al método “makeProductOne()” o “makeProductTwo()” para que devuelva los objetos que serán asignados a atributos de tipo “AbstractProductOne” y “AbstractProductTwo” respectivamente. Si queremos cambiar la plataforma, el único código que hay que cambiar es el que crea la “AbstractFactory” en esta ocasión como un “PlataformOne” en lugar de “PlataformTwo”.

Reglas generales

- Algunas veces los patrones creacionales compiten entre sí: hay caso en que tanto Prototype como Abstract Factory pueden ser utilizadas de manera provechosa. En otras ocasiones son complementarias: Abstract Factory puede almacenar un conjunto de Prototypes de los cuales clonar y retornar objetos producto, Builder puede usar uno de estos patrones para implementar que componentes son construidos. Abstract Factory, Builder y Prototype pueden usar Singleton en sus implementaciones.
- Abstract Factory, Builder y Prototype definen una fábrica de objetos que es responsable de conocer y crear la clase de objetos producto, y los convierte en un parámetro del sistema. Abstract Factory tiene el objeto factoría que produce objetos de varias clases. Builder tiene el objeto factoría para construir un producto complejo de forma incremental utilizando un procedimiento complejo. Prototype tiene el objeto factoría (o prototipo) para construir un producto mediante una copia de un objeto prototipo.
- La implementación de los métodos de fabricación de productos se pueden definir usando Prototype.

- Abstract Factory puede utilizarse como una alternativa a Facade para ocultar clases específicas de una plataforma.
- Builder se centra en construir un objeto complejo paso a paso. Abstract Factory hace hincapié en una familia de productos (que pueden ser simples o complejos). Builder retorna el producto como paso final, pero en el caso de Abstract Factory el product generado se devuelve en cuanto es creado.
- A menudo, los diseñadores comienzan un diseño utilizando Factory Method (que es menos complicado, más configurable, con múltiples subclases) y evolucionan hacia un Abstract Factory, Prototype, o Builder (más flexible y más complejo) según va necesitando más flexibilidad el diseñador.

Factory Method – Patrón de Diseño Creacional

Motivación

- Define una interfaz para la creación de un objeto, pero permite a sus subclasses decidir qué clase instanciar. El Método Factoría permite a una clase delegar la instanciación en sus subclasses.
- Definiendo un constructor “virtual”.
- El operador “new” se considera peligroso.

Problema

Un framework necesita estandarizar el modelo arquitectónico para una serie de aplicaciones, pero permitiendo a las aplicaciones individuales definir su propio dominio de objetos y proporcionar el procedimiento para su instanciación.

Argumentación

El Método Factoría es el equivalente en la creación de objeto al Método Template en la implementación de un algoritmo. Una superclase especifica todo el comportamiento estándar y genérico (utilizando “contenedores” virtuales puros para los pasos asociados a su creación), y delegando los detalles de la creación a las subclasses que se proporcionen por el cliente.

El Método Factoría modifica un diseño haciéndolo más personalizable y sólo un poco más complicado. Otros patrones de diseño requieren nuevas clases, mientras que el Método Factoría sólo requiere una nueva operación.

Generalmente se usa el Método Factoría como un estándar para crear objetos, pero no es necesario si: la clase que se ha instanciado nunca cambia², o si la instanciación se lleva a cabo en una operación que las subclasses pueden fácilmente sobrescribir (como por ejemplo el constructor).³

El Método Factoría es similar a Factoría Abstracta pero sin el énfasis en las familias⁴.

Los Métodos Factoría son habitualmente definidas en un framework arquitectónico, y a continuación implementadas por los usuarios del framework.

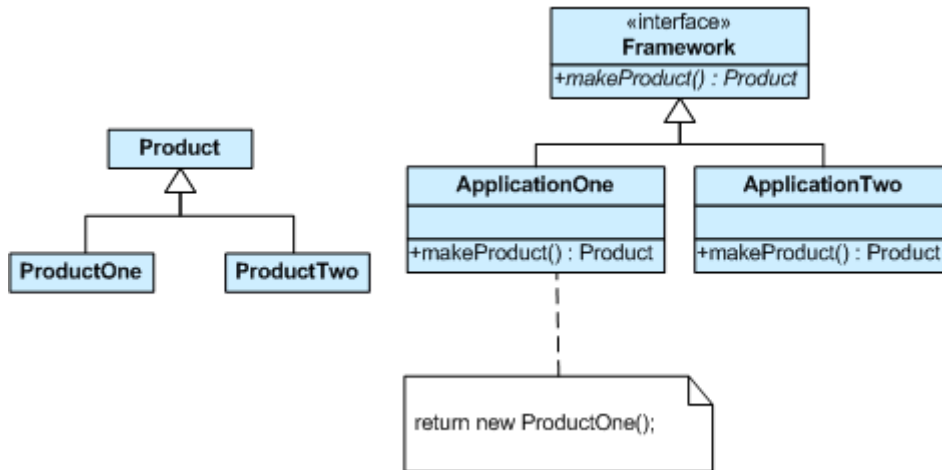
² Si nunca cambia no hay razón para no utilizar directamente el new. Un caso muy particular sería que el objeto fuera de la misma clase pero el método de construcción, es decir el constructor a utilizar, fuera significativamente distinto.

³ Si la construcción del producto se hace de forma centralizada en el constructor, no hay razón para crear una subclase y sobrescribir dicha operación con los nuevos news. La verdadera utilidad de este patrón es cuando los “news” se producen de forma repartida por todo el código de la clase. En este caso heredar y reescribir ya no es una operación trivial y se manifiesta la utilidad de este patrón, al permitirnos reescribir un solo método para reemplazar a todos los news de otra manera dispersos por la clase.

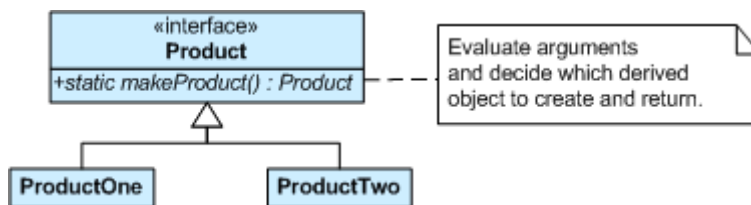
⁴ Ya que aquí creamos un único producto, no una familia. Pero por lo demás estamos separando la manipulación del producto abstracto, del procedimiento que se utiliza para crear el producto concreto (que es el que finalmente se está viendo manipulado).

Estructura

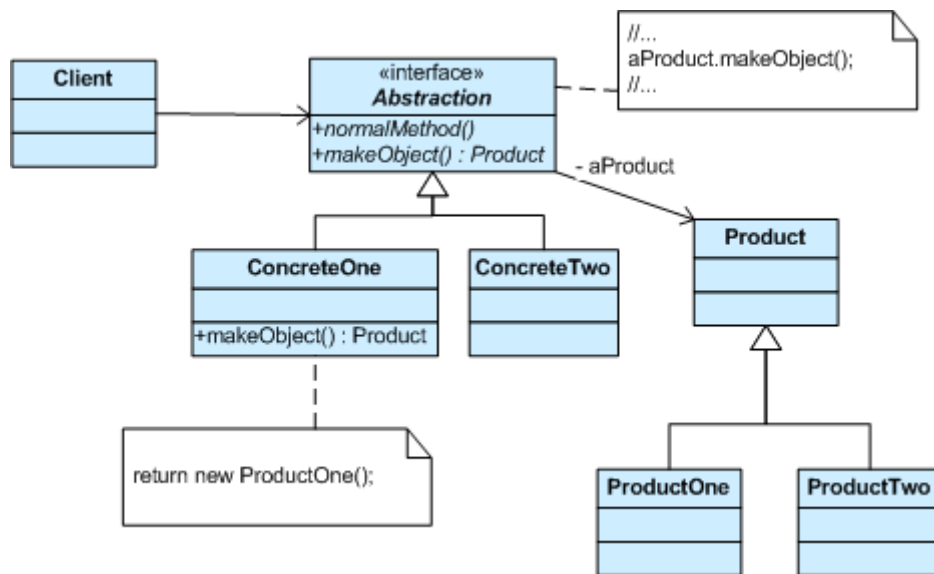
La implementación de Método Factoría tal y como se describe en GoF (ver figura debajo) se solapa en gran medida con la Factoría Abstracta. Por esta razón, la presentación en este capítulo se centra en una aproximación que se ha hecho más popular desde entonces.



Una definición cada vez más popular de un método factoría es: un método `static` de una clase que devuelve un objeto del tipo de esa clase. Pero al contrario que el constructor, el objeto realmente devuelto puede ser una instancia de una subclase. Al contrario que un constructor, un objeto ya existente puede ser reutilizado, en lugar de crear un nuevo objeto. Al contrario que un constructor, los métodos factoría pueden tener nombres diferentes y más descriptivos (e.g. `Color.make_RGB_color(float red, float Green, float blue)` y `Color.make_HSB_color(float hue, float saturation, float brightness)`)



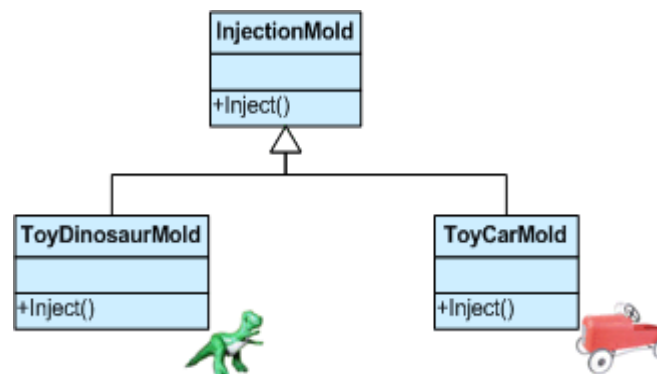
El cliente está totalmente desacoplado de los detalles de implementación de las clases derivadas. Ahora ya podemos crear productos de manera polimórfica.



Ejemplo

Factory Method define una interfaz para crear objetos, pero permite que sus subclases decidan que clase concreta instanciar. El proceso de fabricación por inyección de plástico en moldes es un ejemplo de este patrón.

Los fabricantes de juguetes de plástico parten de pequeñas bolitas de material plástico, las funden e inyectan en moldes con las formas deseadas. La clase de juguete (coche, muñeco de acción, etc.) se determina en función del molde.



Lista de comprobaciones

1. Si tienes una jerarquía de clases que se aprovecha del polimorfismo, considera la posibilidad de añadir una “creación polimórfica” definiendo un método factoría estático en la clase base.
2. Diseña los argumentos del método factoría. ¿Cuáles son las cualidades o características necesarias y suficientes para identificar correctamente la clase derivada a instanciar?
3. Considera la posibilidad de definir una “colección de objetos” que permita que los objetos puedan reutilizarse en lugar de crearse desde cero (estaríamos aplicando el patrón Prototype combinado con Factory Method).

4. Considera la posibilidad de hacer que todos los constructores del producto sean privados o protegidos (para que el cliente no pueda hacer new's directamente).

Reglas de oro

- Las clases Abstract Factory son a menudo implementadas mediante métodos factoría, pero también pueden ser implementados usando Prototype.
- Factory Methods suelen ser utilizados dentro de (el patrón) Template Methods.
- Factory Methods: la creación se realiza por herencia, en Prototype: la creación se realiza por delegación.
- A menudo, los diseñadores comienzan un diseño utilizando Factory Method (que es menos complicado, más configurable, con múltiples subclases) y evolucionan hacia un Abstract Factory, Prototype, o Builder (más flexible y más complejo) según va necesitando más flexibilidad el diseñador.
- Prototype no necesita de subclases, pero necesita una operación de inicialización. Factory Method requiere de subclases, pero no necesita una operación de inicialización.
- La ventaja de los métodos factoría es que pueden devolver la misma instancia múltiples veces, o pueden devolver una subclase en lugar de un objeto de exactamente el mismo tipo.
- Algunos defensores de Factory Method recomiendan que, como una cuestión de diseño del lenguaje (o al menos de estilo al programa) absolutamente todos los constructores deberían ser privados o protegidos. Ya que debería ser el único responsable de la creación de nuevos objetos producto, o de su reutilización.
- El operador new se considera peligroso. Hay una diferencia entre solicitar un objeto y crear uno nuevo. El operador new siempre crea un objeto, y no permite encapsular el proceso de creación de objetos. Un Factory Method sí que encapsula la creación, y permite que un objeto sea solicitado sin que ello signifique la creación de un objeto nuevo cada vez.

Singleton – Patrón de Diseño Creacional

Propósito

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

Problema

La aplicación necesita una, y sólo una, instancia de un objeto. También cuando es necesaria una inicialización perezosa y un acceso global a ella.

Argumentación

Permite que sea la clase de la instancia única la responsable de la creación, inicialización, acceso y aplicación. Se declara la instancia como un campo estático privado de la clase. Proporciona un método estático público que encapsula todo el código de inicialización, y proporciona acceso a la instancia.

EL cliente llama a la función observadora (usando el nombre de la clase y el método de acceso a la instancia interna estática) cada vez que se necesite acceder al objeto singleton.

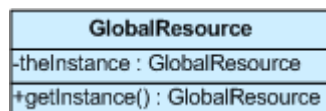
Sólo se debería considerar Singleton si se cumplen los tres criterios siguientes:

- No hay un candidato claro para que se le asigne el objeto singleton
- Es deseable una inicialización perezosa
- Es necesario un acceso global

Estructura



Hace que la clase de una instancia singleton sea responsable de acceder e inicializar (la primera vez) la instancia. La instancia en cuestión es un campo estático privado. La función observadora es un método estático público.



Lista de Comprobaciones

1. Define un atributo privado static en la clase singleton.
2. Define un observador static público en la clase.
3. Realiza una “inicialización perezosa” (creación en el primer uso) en la función de acceso.
4. Define todos los constructores como protected o private.
5. Los clientes pueden usar solamente el observado para manipular el Singleton.

Reglas de oro

- Abstract Factory, Builder y Prototype pueden usar Singleton en su implementación.
- Los objetos Facade usan a menudo Singletons debido a que sólo un objeto Facade se necesita.
- Los objetos State son muy a menudo Singletons.
- La ventaja de Singleton sobre las variables globales es que puedes estar absolutamente seguro del número de instancias cuando usas Singleton, y, tú puedes cambiar de idea y gestionar cualquier número de instancias.
- El patrón de diseño Singleton es uno de los patrones más mal utilizados. Singleton están diseñados para ser usados cuando una clase debe tener exactamente una sola instancia, ni más ni menos. Los diseñadores frecuentemente usan Singletons en un fallido intento de reemplazar variables globales. Un Singleton es, por intención y propósito, una variable global. Singleton no elimina las variables globales, simplemente las renombra.
- ¿Cuándo Singleton no es necesario? Respuesta corta: la mayor parte del tiempo. Respuesta larga: cuando es más fácil pasar un objeto como referencia a los objetos que lo necesitan, en lugar de acceder globalmente a él. El problema real con los Singletons es que proporcionan una excusa muy fácil para no pensar en de qué forma escoger la visibilidad de un objeto.
- No se debería utilizar como “disculpa” para mantener un conjunto de variables globales que por dejadez no se han sabido ubicar correctamente, con lo que la solución rápida ha sido convertirlas en campos/clases estáticas a modo de Singleton.

Adapter – Patrón de Diseño Estructural

Propósito

- Convertir la interfaz de una clase en la interfaz de la que esperan encontrarse los clientes. Adapter permite que colaboren clases que de otra forma no podrían debido a utilizar interfaces diferentes.
- Empaquete una clase ya definida con una nueva interfaz.
- Consigue integrar un componente viejo en un sistema nuevo.

Problema

Cuando tenemos un componente ya desarrollado con una funcionalidad que nos gustaría reutilizar, pero que su “visión del mundo” no es compatible con la filosofía de la arquitectura del sistema que estamos desarrollando.

Debate

La reutilización (de software) ha sido siempre un tema escabroso y complejo. Una de las razones ha sido la problemática de diseñar algo nuevo, mientras reutilizamos algo antiguo. Siempre hay algo que no encaja del todo bien entre un diseño previo y el nuevo que tenemos entre manos. Ya pueden ser sus dimensiones físicas o desajustes. Puede ser la temporización o la sincronización. Puede ser por presunciones desafortunadas o por estándares que chocan entre sí.

Es como el problema de insertar un enchufe de 220 voltios en un conector de pared americano, ambos responden a estándares que facilitan su utilización en sus demarcaciones geográficas, pero para conectar es necesario algún tipo de adaptador o intermediario.



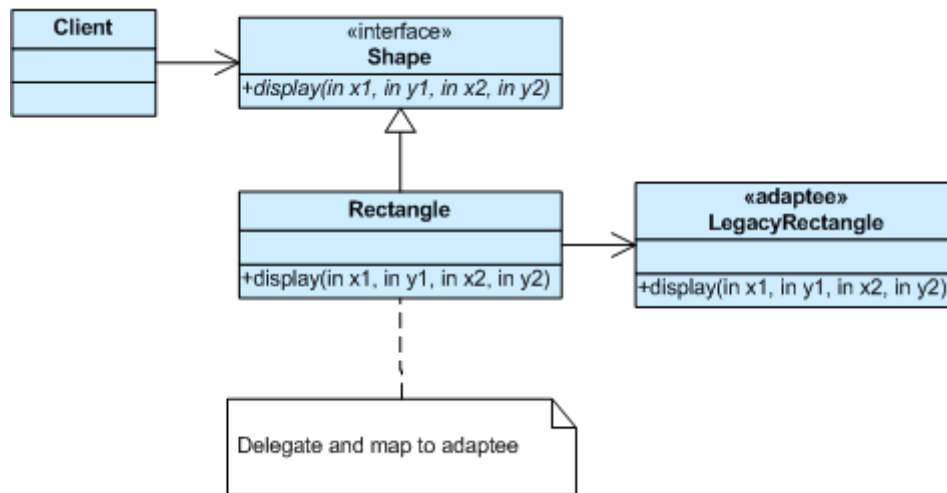
El patrón Adapter trata de crear una abstracción intermedia que traduzca, o mapee, el componente viejo en el sistema nueva. Los clientes invocan métodos de del objeto Adapter que éste redirige a llamadas correctas del componente original. Esta estrategia puede ser implementada tanto con herencia como con agregación.

Adapter funciona como un envoltorio o modificador de una clase ya existente. Proporciona una vista diferente, o reinterpretada, de una clase.

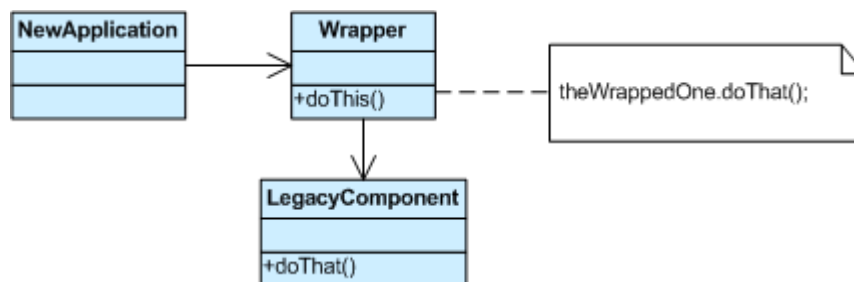
Estructura

En el siguiente ejemplo, podemos ver como un componente Rectangle dispone de un método display() que espera recibir como parámetros, el punto superior izquierdo: x,y; y el ancho y alto: w y h. Pero el cliente quiere pasarle “la posición x e y de la esquina superior izquierda” y

“la posición x e y de la esquina inferior derecha”. Podemos reconciliar a ambas parte por medio de un nivel adicional de direccionamiento – esto es, un objeto Adapter.

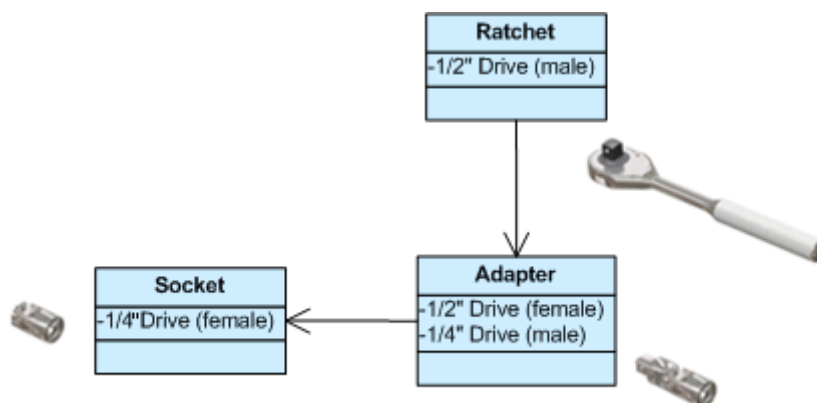


El Adapter también puede ser interpretado como un “envoltorio”.



Ejemplo

El patrón Adapter permite que clases, de otra forma incompatibles, puedan colaborar, convirtiendo para ello la interfaz de una clase en la interfaz que esperan los clientes. Una llave de carraca multicabezal nos proporciona un ejemplo de Adapter. La llave lleva un conector que permite adaptar cabezales de tuerca de distinto diámetro. Hay conectores de $\frac{1}{2}$ " y de $\frac{1}{4}$ ", que no son directamente compatibles. Para adaptar un ancho al otro se utiliza una pieza adaptadora que por un extremo es un macho $\frac{1}{4}$ " y por el otro hembra de $\frac{1}{2}$ ".



Lista de Comprobaciones

1. Identificar los actores: el/los componente(s) que desean acceder al componente antiguo (i.e. el cliente), y el componente que necesita ser adaptado (i.e. el adaptable).
2. Identificar la interfaz que requiere el cliente.
3. Diseñar una clase “envoltorio” que pueda adapte el acceso del componente al cliente.
4. Opción A: La clase adaptador/envoltorio “es una” instancia de la clase adaptada.
5. Opción B: La clase adaptador/envoltorio “redirecciona” la interfaz del cliente a la interfaz del componente adaptado.
6. El cliente utiliza esta nueva interfaz para acceder de forma transparente al componente.

Reglas de Oro

- Adapter consigue que las cosas funcionen después de haber sido diseñadas; Bridge consigue que colaboren antes de ser diseñadas.
- Bridge se diseña de antemano para permitir que la abstracción y la implementación pueden variar de forma independiente. Adapter se aplica para conseguir que clases sin relación alguna colaboren.
- Adapter proporciona una interfaz diferente a su sujeto. Proxy proporciona la misma interfaz. Decorator proporciona una interfaz mejorada (ampliada).
- Adapter trata de cambiar la interfaz de un objeto que ya existe. Decorator mejora otro objeto sin cambiar su interfaz. Decorator es por tanto más transparente a la aplicación que un Adapter. Como consecuencia, Decorator permite la composición recursiva, lo que resulta imposible en los Adapter puros.
- Facade define un nuevo interfaz, mientras que Adapter reutiliza uno antiguo. Recordemos que Adapter consigue que dos interfaces ya existentes colaboren entre sí, sin necesidad de definir un interfaz totalmente nuevo.

Facade – Patrón de Diseño Estructural

Propósito

Proporciona una interfaz unificada a un conjunto de interfaces de un subsistema. Facade define un interfaz de alto-nivel que facilita el uso del subsistema.

Empaqueta un complejo subsistema con una interfaz simple.

Problema

Un segmento de la comunidad cliente necesita una interfaz simplificada para acceder a todas las funcionalidades de un subsistema complejo

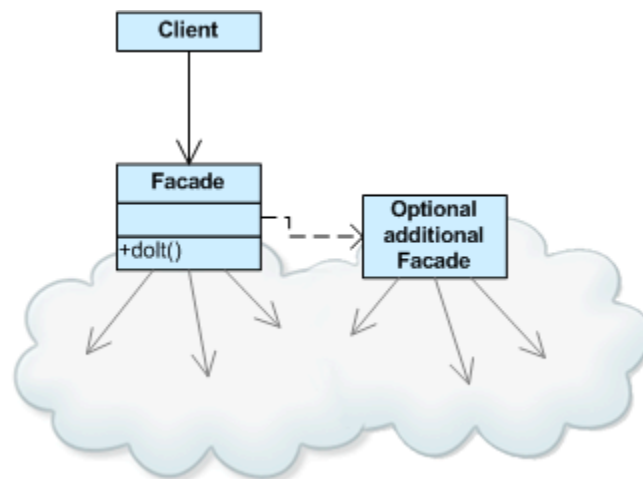
Debate

Facade resuelve la encapsulación de un subsistema complejo por medio de un simple objeto interfaz. Lo que reduce la curva de aprendizaje necesaria para aprovechar adecuadamente el subsistema. También promueve el desacople del subsistema de sus posiblemente muchos clientes. Por otro lado, si Facade es el único punto de acceso al subsistema, estaremos limitando la flexibilidad y las características que pueden llegar a demandar clientes con requisitos particulares o muy exigentes.

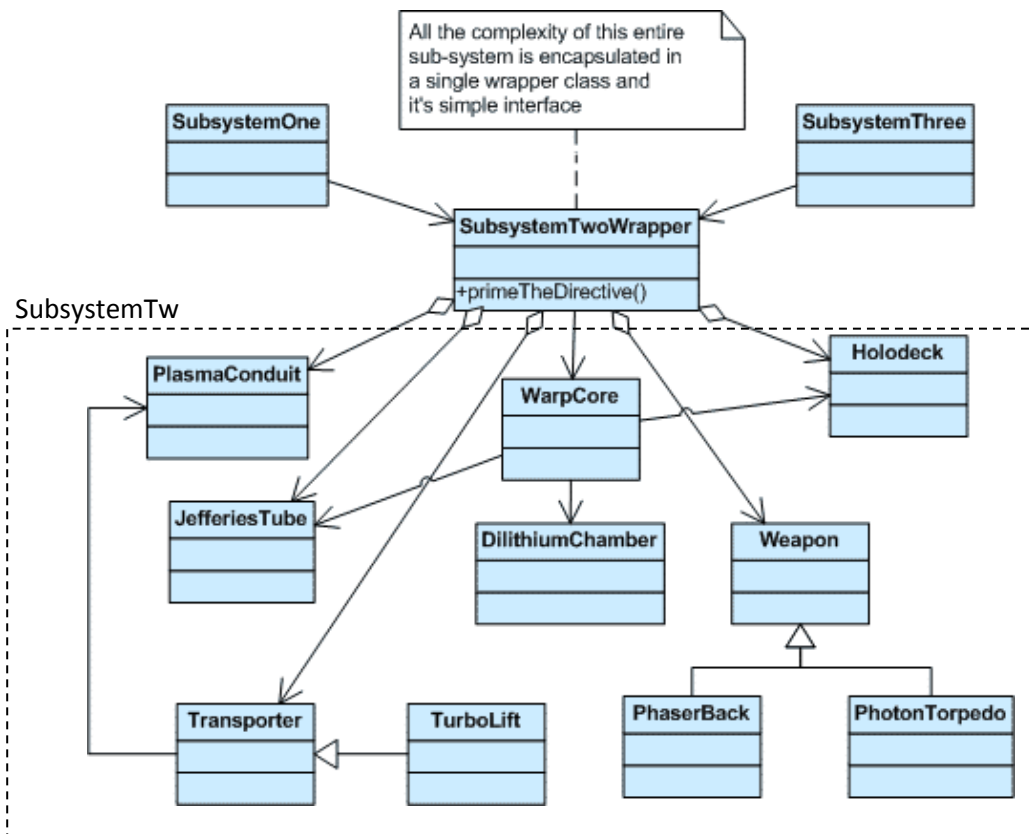
El objeto Facade debería ser un simple intermediario para facilitar el acceso. No debería considerarse en ningún caso como objeto “todopoderoso”.

Estructura

Facade se comporta como una visión simplista y accesible a un software potencialmente complejo, difícil o tedioso de manejar.

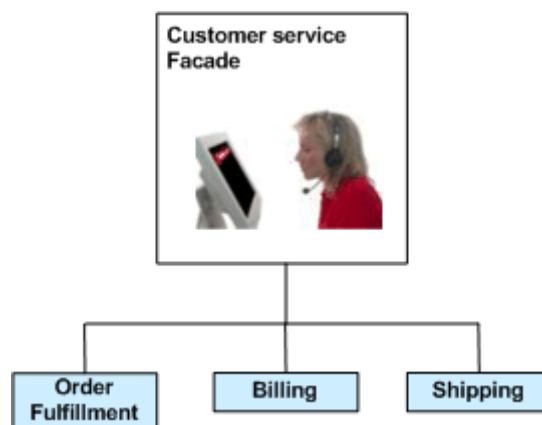


Así en el siguiente diagrama SubsystemOne y SubsystemThree no interaccionan directamente con los componentes internos de SubSystemTwo. En lugar de eso utilizan el SubsystemTwoWrapper “facade” (es decir, un nivel de abstracción mayor).



Ejemplo

Facade define una interfaz unificada y de alto nivel a un subsistema que facilita su utilización. Los clientes encuentran un Facade cuando hacen pedidos de un catálogo. El cliente llama a un número y habla con un representante del servicio al cliente. El representante actúa como un Facade, proporcionando una interfaz con el departamento de ventas, el departamento de facturación, y el departamento de distribución de pedidos.



Lista de Comprobaciones

1. Identificar un interfaz unificado y más simple de acceder al subsistema o componente.
2. Diseñar una clase contenedor que encapsule al subsistema.
3. La fachada/encapsulado captura la complejidad y colaboraciones entre los componentes del subsistema, y delega las peticiones de los clientes en los métodos apropiados de cada uno de ellos.
4. El cliente utiliza (se conecta únicamente) al objeto Facade.

5. Valoraremos el beneficio de definir objetos Facade adicionales.

Reglas de Oro

Facade define un nuevo interfaz, mientras que Adapter utiliza un interfaz previamente definido. Debemos recordar que Adapter logra que dos interfaces previamente definidos colaboren entre sí en oposición a crear un interfaz nuevo.

Mientras que Flyweight muestra cómo hacer montones de pequeños objetos, Facade muestra cómo lograr que un simple objeto represente un subsistema completo.

Mediator es similar a Facade en el sentido de que abstrae la funcionalidad de clases ya existentes. Mediator abstrae/centraliza las comunicaciones entre objetos colaboradores. Por norma general añade valor (funcionalidades), y es conocido por todos los objetos colaboradores. En contraste, Facade define un simple interfaz al subsistema, no añade nuevas funcionalidades, y no es conocido por las clases del subsistema.

Abstract Factory puede ser utilizado como una alternativa a Facade para ocultar las clases específicas de una plataforma.

Los objetos Facade son a menudo Singletons ya que sólo se necesita un objeto Facade.

Adapter y Facade realizan ambos empaquetados; pero son de distinto tipo. El objeto de Facade es crear un nuevo interfaz simple y unificado. Mientras que Adapter trata de diseñar un interfaz adaptado a uno ya existente. Mientras que habitualmente Facade empaqueta múltiples objetos y Adapter empaque un solo objeto; Facade también puede servir de front-end a un único objeto complejo y Adapter podría adaptar distintos objetos para lograr su colaboración.

Composite – Patrón de Diseño Estructural

Propósito

- Componer objetos en estructuras en forma de árbol para representar jerarquías de tipo parte-todo. Composite permite a los clientes trabajar con objetos individuales y con composiciones de objetos de forma uniforme.
- Composición recursiva
- “Directorios que contienen entradas, cada una de las cuales puede ser un directorio”
- Una jerarquía “es un” de forma ascendente, y 1-a-muchos “tiene un” de forma descendente.

Problema

Una aplicación necesita manipular una colección jerárquica de objetos “primitivos” y “compuestos”. El procesamiento de un objeto primitivo es distinto del de un objeto compuesto. Nos fuerza a identificar el “tipo” de cada objeto para poder decidir cuál es el procesamiento adecuado a aplicar.

Debate

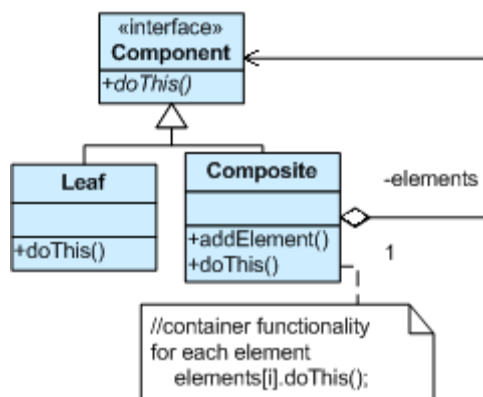
Define una clase base abstracta (Component) que especifica el comportamiento que van a necesitar todos los objetos, de forma uniforme, tanto para los primitivos como los compuestos. Los objetos primitivos (Primitive) y compuestos (Composite) se modelan como subclases de la clase base abstracta. Cada objeto compuesto gestiona a sus “hijos” a través únicamente de la clase base abstracta.

Este patrón debería utilizarse cuando tenemos “composiciones que contienen componentes, cada uno de los cuales puede ser una composición”.

Los métodos de gestión de nodos hijos (por ejemplo addChild(), removeChild()) normalmente deberían estar definidas en la clase Composite. Desafortunadamente, al tener como objetivo el tratar objetos primitivos y compuestos de igual forma, requiere que estos métodos se muevan a la clase abstracta Component. Sobre este tema, en la sección de “Opinión” se plantean aspectos enfrentados de “seguridad” y “transparencia”.

Estructura


Tenemos objetos compuestos (Composite) que contienen componentes (Component), que a su vez pueden ser objetos compuestos.



Menús que contienen ítems de menú, cada uno de los cuales pueden ser otro (sub)menú.

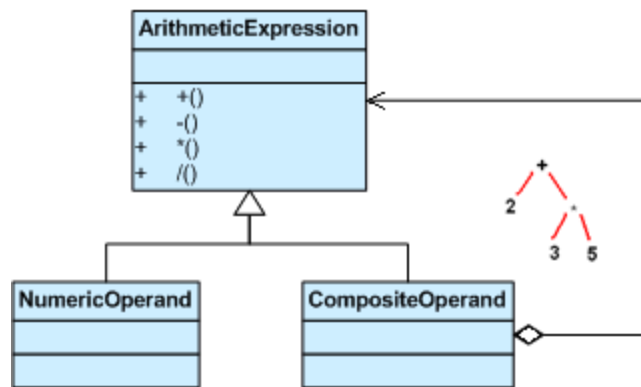
En el diseño de una interfaz de usuario gráfica (GUI) los gestores de organización de los controles en filas-columnas, donde cada uno de dichos controles puede ser otro gestor de controles en filas-columnas.

Contenedores que contienen Elementos, cada uno de los cuales puede ser un contenedor.

	
<p>Matrioskas como contenedor y contenido</p> <pre>graph TD sites[sites] --> 11heavens[11heavens.com] sites --> all[all] all --> modules[modules] all --> themes[themes] themes --> barlow[barlow] themes --> garland_ext[garland_ext] garland_ext --> accordion[accordion] garland_ext --> vertical_tabs[vertical_tabs] vertical_tabs --> garland_ext_info[garland_ext.info] vertical_tabs --> template_php[template.php] vertical_tabs --> theme_settings_php[theme-settings.php] themes --> README[README.txt] all --> default[default]</pre> <p>Árbol de carpetas y archivos, donde las carpetas pueden ser contenedores pero a su vez están contenidos</p>	<p>Matrioskas en su versión más geek</p>

Ejemplo

Composite “compone” objetos en estructuras de árbol y permite a los clientes trabajar con objetos individuales y compuestos de forma uniforme. Aunque el ejemplo es abstracto, las expresiones aritméticas son Composites. Una expresión aritmética consiste en un operando, un operador (+ - * /), y otro operando. Cada operando puede ser un número, u otra expresión aritmética. Por tanto $2 + 3$ y $(2 + 3) + (4 * 6)$ son dos expresiones válidas.



Lista de Comprobaciones

1. Asegúrate de que tu problema se centra en representar relaciones jerárquicas del tipo “parte-todo”.
2. Considera la siguiente regla: “*contenedores* que contienen *contenidos*, cada uno de los cuales puede ser un contenedor.” Por ejemplo, “Ensamblados que contienen componentes, cada uno de los cuales puede ser un componente”. Divide los conceptos del dominio de tu problema en *clases contenedoras* y *clases contenidas*.
3. Crea la interfaz “menor denominador común” que permita manejar por igual tus contenedores y tus contenidos. Esta interfaz debería especificar el comportamiento que se debe ejercer uniformemente tanto sobre objetos contenidos, como sobre objetos contenedores.
4. Tanto las clases contenedoras como contenidas declaran una relación “es un” en su interfaz.
5. Todas las clases contenedores declaran además una relación “tiene un” uno-a-muchos en su interfaz.
6. Las clases contenedoras aprovechan el polimorfismo para delegar las acciones en sus objetos contenidos.
7. Los métodos de gestión de nodos hijos (por ejemplo `addChild()`, `removeChild()`) normalmente deberían estar definidas en la clase Composite. Desafortunadamente, al tener como objetivo el tratar objetos primitivos y compuestos de igual forma, requiere que estos métodos se muevan a la clase abstracta Component. En el libro GoF se plantean los criterios para llegar a un compromiso entre “seguridad” y “transparencia” en el diseño.

Reglas de Oro

- Composite y Decorator tienen diagramas de estructura similares, reflejando el hecho de que ambos se basan en la composición recursiva para organizar un número ilimitado de objetos.
- Una estructura Composite puede recorrerse con un Iterator. Visitor puede aplicar una operación sobre una estructura Composite. Composite podría usar Chain of Responsibility para permitir a los componentes acceder a propiedades globales a través de sus progenitores (vía herencia). También podría usar Decorator para sobrescribir estas propiedades en partes de la composición. Se podría usar Observer para enlazar la estructura de un objeto a otro y State para permitir un componente cambiar su comportamiento según cambie su estado.

- Composite puede permitirte componer un Mediator a partir de piezas más pequeñas por medio de la composición recursiva.
- Decorator está diseñado para permitirte añadir responsabilidades a objetos sin necesidad de subclases. Composite se centra **en la representación**. Sus objetivos son distintos pero complementarios. Por esta razón, Composite y Decorator son utilizados en combinación con frecuencia.
- Flyweight con frecuencia se combina con Composite para implementar nodos hojas compartidos.

Opiniones

Las opiniones planteadas en la web **sourcemaking.com** nos describen conclusiones a las que han llegado profesionales de la informática como resultado de la aplicación práctica de este patrón. El enlace es: http://sourcemaking.com/design_patterns/composite (sección Opinions)

Particularmente interesante es la referida a donde ubicar la interfaz de las operaciones de gestión de los nodos hijos ¿en la clase componente base? ¿O en la subclase componente compuesto? Veamos la opinión de los autores:

“Mis clases Component no saben que la subclase Composite existe. Así que no proporcionan ninguna ayuda para navegar por los Composites, ni ofrecen ayuda para alterar los contenidos (nodos hijos) de un Composite. Esto es debido a que yo prefiero que la clase base (y todas sus clases derivadas) puedan ser reutilizadas en contextos donde no se requieran Composites. Cuando dado un puntero a una clase base (en lenguajes como C++, una referencia en el caso de Java o C#), si yo necesito saber exactamente si es un objeto Composite o si es Leaf (hoja), yo utilizaré un **dynamic_cast** (en el caso de C++, en el caso de Java el operador **instanceof**) para resolverlo. En aquellos casos donde utilizar un dynamic_cast resulte muy costoso se utilizaría un (patrón) Visitor.”

Command – Patrón de Diseño de Comportamiento

Propósito

- Encapsular las solicitudes a un objeto, lo que permite parametrizar clientes con diferentes solicitudes, encolar o registrar las peticiones, y permitir “deshacer” las operaciones.
- Promocionar “la invocación de un método en un objeto” al estatus de objeto.
- Una forma orientada a objeto de implementar un call-back⁵.

Problema

Es necesario emitir peticiones a objetos sin saber nada acerca de la operación que queremos invocar ni sobre el receptor de la petición.

Debate

Command desacopla el objeto que realiza la petición de una operación de aquel que sabe cómo ejecutarla. Para lograr esta separación, el diseñador crea una clase abstracta base que mapea un receptor (un objeto) con una acción (un puntero a una función miembro). La clase base contiene un método `execute()` que simplemente invoca la acción en el receptor.

Todos los clientes de los objetos Command tratan cada objeto como una “caja negra” ya que simplemente invocan al método virtual `execute()` del objeto cada vez que el cliente requiere el “servicio” del objeto.

Una clase Command define un subconjunto de los siguientes elementos: un objeto, un método que debe ser aplicado al objeto, y los argumentos a pasarle cuando el método es aplicado. El método “execute” de Command es el responsable de combinar todas estas piezas.

Secuencias de objetos Command pueden ser ensambladas para formar acciones compuestas (o macros).

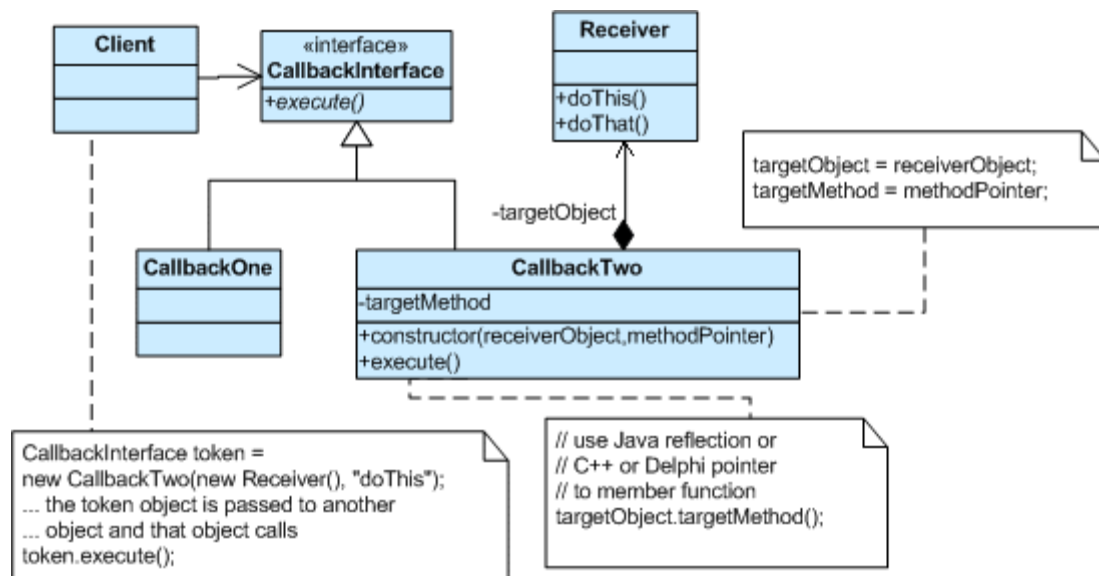
Estructura

El cliente que crea el objeto Command no es el mismo cliente que lo ejecuta. Esta separación proporciona flexibilidad en el tiempo y orden de ejecución de los Commands.

Representar las ordenes (commands) mediante objetos significa que pueden ser pasados, expuestos, compartidos, cargados en una tabla, y en definitiva ser manipulados como cualquier otro objeto.⁶

⁵ Se denomina “callback” (retrollamada) a un método que es llamado automáticamente como respuesta, generalmente, a un evento. Ejemplo: En el diseño de interfaz gráficos, los componentes (como por ejemplo un botón) definen una serie de eventos (como por ejemplo hacer clic en el botón) a los que se puede asociar un método. Este método se invoca automáticamente cuando se produce el evento.

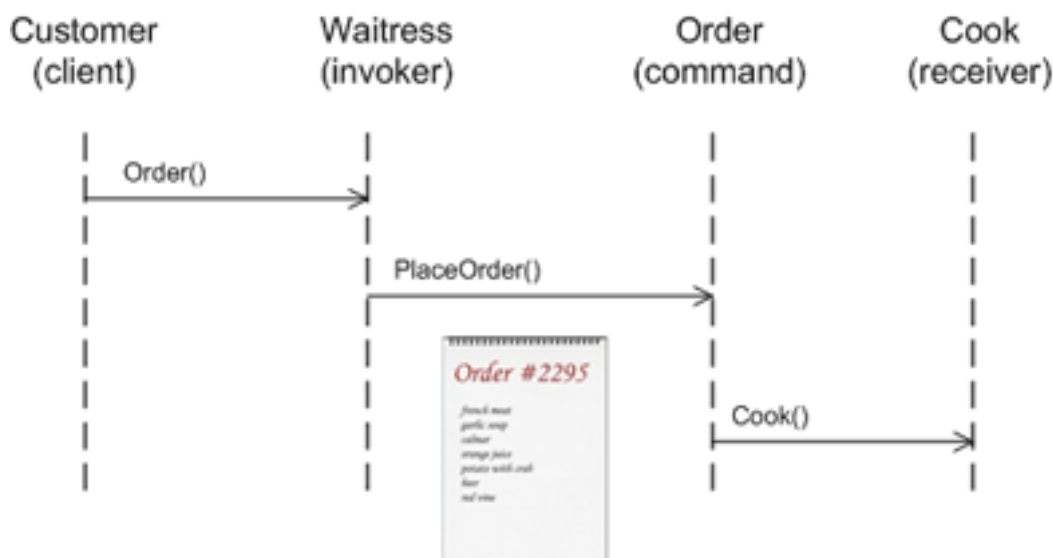
⁶ Aquí lo que nos vienen a decir es que el código (la llamada a un método con sus argumentos) se puede modelar como datos (un objeto con unas propiedades y operaciones), para luego finalmente volver a interpretarlo como código (es, decir que finalmente podemos ejecutar el método original que modelamos como objeto). Es muy similar a lo que pasa en la programación funcional con las funciones. Las funciones se manipulan como datos, se pueden meter en una lista, pasar como argumento o ser



Los objetos Command pueden ser interpretados como “tokens” que son creados por un cliente que conoce que necesita que se haga, y pasados a otro cliente que tiene los recursos para llevarlo a cabo.

Ejemplo

El patrón Command permite encapsular en un objeto la petición a un método, de esta forma permite podemos parametrizar los clientes con diferentes peticiones. El “pedido” en una cena es un ejemplo de patrón Command. El camarero o camarera toma un pedido u “orden” (command) del cliente y encapsula ese pedido escribiéndolo en una libreta. La hoja del pedido es entonces encolado para ser cocinado. Es importante señalar que la libreta de pedidos que utiliza el camarero no depende del menú, y por tanto permite generar órdenes de cocina de ítems muy diferentes.



resultado de una función; pero finalmente ese “metadato” puede volver a utilizarse como función y obtener el resultado de su evaluación al aplicarse a unos argumentos concretos.

Lista de Comprobaciones

1. Define un interfaz Command con un prototipo de método similar a `execute()`.
2. Crea una o más clases derivadas que encapsulen algún subconjunto de los siguientes: un objeto “receptor”, un método a invocar, los argumentos a pasarle.
3. Instancia un objeto Command para cada llamada pospuesta.
4. Pasa el objeto Command de su creador (es decir, el transmisor) al que finalmente invoca la acción (es decir, el receptor).
5. El receptor decide cuando ejecutar la acción, es decir, cuando llamar a `execute()`.

Reglas de oro

- Chain of Responsibility, Command, Mediator y Observer, se encargan de desacoplar el transmisor y receptor, pero con diferentes acuerdos. Command normalmente especifica una conexión transmisor-receptor con una clase.
- Chain of Responsibility puede usar Command para representar peticiones como objetos.
- Command y Memento actúan como tokens mágicos que deben ser pasados de unos a otros e invocados más tarde. En Command, el token representa una petición; en Memento, representa el estado interno de un objeto en un instante particular. El polimorfismo es importante en Command, pero no en Memento ya que su interfaz es tan mínima que un Memento sólo puede pasarse como un valor.
- Command puede utilizar Memento para mantener el estado requerido para una operación de “deshacer” (undo).
- Una macro formada por la unión de varios Commands pueden implementarse con Composite.
- Un Command que deba ser copiada antes de colocarse en un lista history (o registro) actúa como un Prototype.
- Dos aspectos importantes del patrón Command: separación de la interfaz (el invocador está aislado del receptor), separación en el tiempo (almacena una llamada a un método lista para ser ejecutada, pero que se iniciará más tarde).

State – Patrón de Diseño de Comportamiento

Propósito

- Permitir a un objeto alterar su comportamiento cuando su estado interno cambia. El objeto dará la sensación de ser de “otra clase”.
- Una máquina de estados orientada a objeto.
- Encapsulación + envoltorio polimórfico + colaboración.

Problema

El comportamiento de un objeto monolítico depende de su estado (state), de forma que su comportamiento cambia en tiempo de ejecución dependiendo de ese estado. O, una aplicación que está caracterizada por una gran cantidad de sentencias de control (if-the-else) que configuran el flujo de control del programa en función del estado de la aplicación.

Debate

El patrón State es una solución al problema de cómo lograr que el comportamiento dependa del estado del programa.

- Define una clase “contexto” que ofrece una interfaz sencilla al resto del mundo.
- Define una clase base abstracta State.
- Define un comportamiento dependiente del estado en la clase apropiada derivada de State.
- Mantiene un puntero/referencia al “estado” actual en la clase “contexto”.
- Para cambiar el estado de la *máquina de estados* cambiamos la referencia “estado” actual.

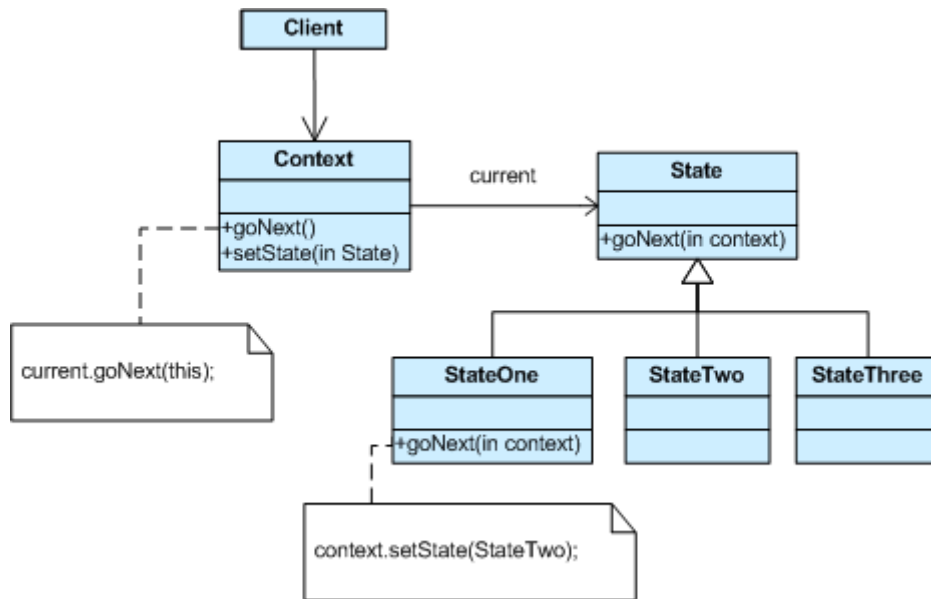
El patrón State no decide donde se producen las transiciones de estado. Las alternativas son dos: el objeto “contexto”, o cada una de las clases derivadas de State. La ventaja de esta última opción es la facilidad de añadir nuevas clases derivadas de State.

Una aproximación basada en una tabla de transiciones para diseñar una máquina de estados ofrece una buena solución al problema de especificar las transiciones de estado., pero es difícil de añadir acciones que acompañen dichas transiciones. La aproximación basada en patrones utiliza código (en lugar de estructuras de datos) para especificar transiciones de estado, pero a cambio ofrece una buena solución a la hora de ejecutar acciones con cada transición.

Estructura

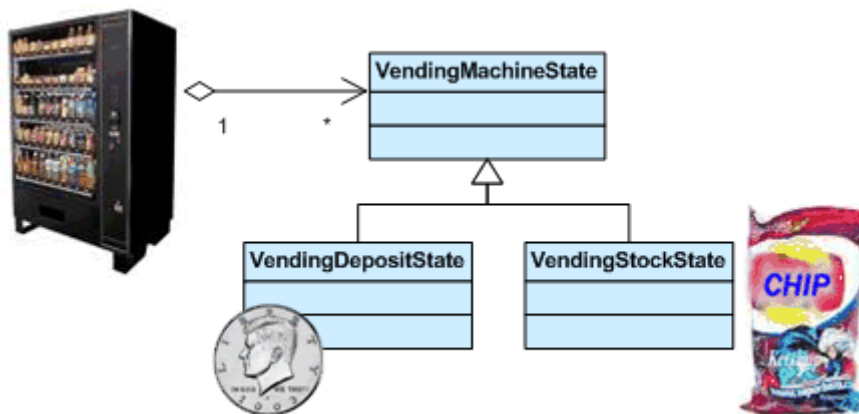
La interfaz de máquina de estados está encapsulada en la clase *envoltorio* (Context). La interfaz de la jerarquía de objetos *envueltos* (State) es una réplica de la del envoltorio excepto por un argumento adicional. Este argumento extra permite a la clase envuelta derivada, llamar de nuevo a la clase envoltorio si es necesario. La complejidad de definir las transiciones, que recaería en la clase envoltorio, ahora está perfectamente repartida y encapsulada en una jerarquía polimórfica en la que delega el objeto envoltorio⁷.

⁷ Es decir, que el objeto Contexto en lugar de tener una tabla de datos con la especificación de cada posible estado, y cada posible transición a otros estados, delega en su “estado actual”. Que es un objeto



Ejemplo

El patrón State permite a un objeto cambiar su comportamiento cuando su estado interno cambia. Este patrón puede ser observado en una máquina expendedora. Estas máquinas tienen un estado basado en: su inventario, la cantidad de dinero depositada, la disponibilidad de cambio, el ítem seleccionado, etc. Cuando el dinero es introducido y se ha hecho la selección, la máquina entregará el producto y no dará cambio, entregará el producto y el cambio, no entregará el producto debido a que no hay suficiente dinero para el cambio, o no entregará el producto debido a que se ha agotado el producto.



Lista de comprobaciones

1. Identificar una clase ya existente, o crear una nueva clase, que sirva como “máquina de estados” desde el punto de vista del cliente. Esta clase actuará como *Contexto*.
2. Crear una clase base State que replica los métodos de la interfaz de la máquina de estados. Cada método incorpora un parámetro adicional: una instancia de la clase

que sólo conoce las posibles transiciones desde él a un estado diferente. Así que las transiciones son conocidas sólo por el “estado origen de la transición”. De esta manera, el objeto Contexto puede delegar en los objetos Estado toda acción relativa a los cambios de estado. Y los clientes sólo tienen que trabajar con un objeto Contexto que ofrece una interfaz muy simple.

Contexto. La clase base especificará cualquier comportamiento “por defecto” que resulte útil.

3. Crear una clase derivada de State por cada posible estado. Estas clases derivadas sólo sobrescriben los métodos que necesiten ser reescritos.
4. La clase Contexto mantiene un objeto State “actual”.
5. Todas las peticiones de clientes al objeto Contexto son simplemente reenviadas al objeto State actual, y se incorpora a la llamada la referencia `this` al objeto Contexto.
6. Los métodos de State cambian el estado “actual” en el objeto contexto de acuerdo con su lógica definida.

Reglas de oro

- Los objetos State son a menudo Singleton.
- Flyweight establece cuándo y cómo pueden compartirse los objetos State.
- Interpreter puede usar State para definir sus contextos de análisis.
- Strategy tiene 2 diferentes implementaciones, la primera es similar a State. La diferencia está en los momentos en que se establecen las dependencias (Strategy es un patrón de un enlace único, mientras que State es más dinámico).
- Las estructuras de State y Bridge son idénticas (excepto que Bridge admite jerarquías de clases “envoltorio”, mientras que State permite sólo una). Los dos patrones usan la misma estructura para resolver problemas distintos: State permite que el comportamiento de un objeto cambie junto con su estado, mientras que el objetivo de Bridge es desacoplar una abstracción de su implementación de forma que las dos puedan variar independientemente.
- La implementación del patrón State se fundamenta en el patrón Strategy. La diferencia entre State y Strategy es su objetivo. Con Strategy, la elección del algoritmo es bastante estable. Con State, un cambio en el estado del objeto “contexto” le fuerza a seleccionar uno entre los objetos Strategy de su “paleta”.

Observer – Patrón de Diseño de Comportamiento

Propósito

- Definir una relación de dependencia 1 a n entre objetos, de forma que cuando un objeto cambia su estado, todos los objetos dependientes se les notifica y actualiza automáticamente.
- Encapsula los componentes fijos “a ser observados” como una única abstracción “Sujeto” (Subject), y los componentes variables (u opcionales o elementos de la interfaz de usuario) como una jerarquía de “Observador” (Observer).
- La “Vista” del patrón Modelo-Vista-Controlador.

Problema

Un enorme diseño monolítico no escala bien⁸ según se van añadiendo más requisitos de monitorización o representación gráfica del sistema.

Debate

Definir un objeto que es el “guardián” del modelo de datos y/o la lógica de negocio (el “Sujeto”). Delegar toda la funcionalidad de “representación” (view) en objetos Observer desacoplados y distintos. Los Observer se registran ellos mismos en el Subject en el momento en que son creados.

Siempre que el Subject cambie, se encarga de propagar que ha cambiado a todos los Observers registrados, y cada Observer solicitará al Subject el subconjunto del estado actual de Subject que es responsable de monitorizar.

Esto permite que el número y “tipo” de objetos “vista” pueda ser configurado dinámicamente, en lugar de estar especificado estáticamente en tiempo de compilación.

El protocolo descrito arriba especifica un modelo de interacción denominado “pull” (extraer). En lugar de ser el Subject el responsable de “pushing” (enviar) lo que ha cambiado a todos los Observers, cada Observer es responsable de “extraer” su particular “ventana de interés” del Subject. El modelo “push” complica su reutilización, mientras que el modelo “pull” es menos eficiente.

Los aspectos que se discuten, pero que se dejan a discreción del diseñador, incluyen: implementar la compresión de eventos (enviar un único mensaje broadcast⁹ con todos los cambios consecutivos que se han producido), tener un solo Observer monitorizando múltiples Subjects, y asegurando que un Subject notifica a sus Observers **cuando está a punto de finalizar**.

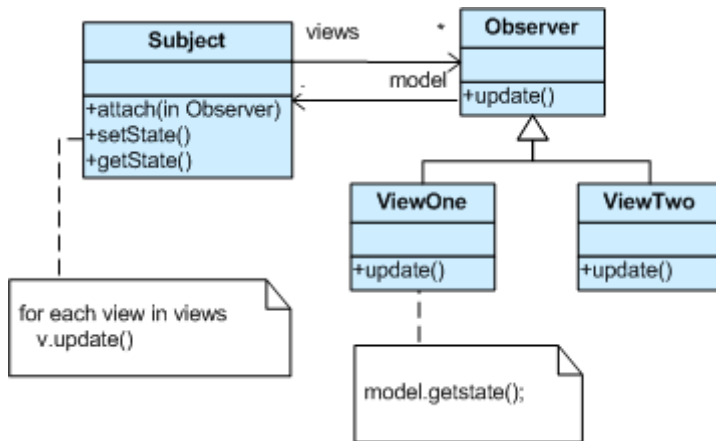
El patrón Observer captura gran parte de la potencia de la arquitectura Modelo-Vista-Controlador que ha sido una parte importante de la comunidad Smalltalk¹⁰ durante años.

⁸ Que no escala bien un sistema, se refiere a que cuanto más grande o complejo menos eficiente es.

⁹ Se traduce por “difusión”: un mensaje difundido a “todos”, en este caso, los observadores.

¹⁰ Un lenguaje orientado a objetos emblemáticos del paradigma de la P.O.O.

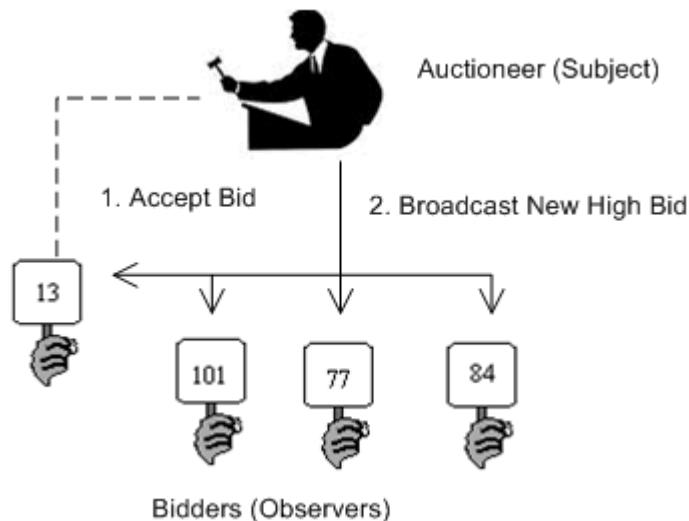
Estructura



Subject representa la entidad a ser observada. Observer representa la abstracción que puede ser variable (en tipo concreto y en número). El Subject indica a los objetos Observer cuando deben hacer su trabajo de “revisar los cambios”. Cada Observer puede entonces consultar al Subject como necesite¹¹.

Ejemplo

El Observer define una relación uno-a-muchos de forma que en el momento en que un objeto cambia se notifique y actualice a todos los demás automáticamente. Algunas subastas demuestran este patrón. Cada pujador posee una paleta numerada que usa para indicar que puja. El subastador (Auctioneer) empieza la subasta, y “observa” cuando una paleta se levanta y acepta la puja. La aceptación de una puja cambia el precio de la subasta que es notificado a todos los pujadores¹² en forma de una nueva subasta.



Lista de Comprobaciones

1. Diferenciar entre la funcionalidad independiente y la funcionalidad dependiente.
2. Modelar la funcionalidad independiente como la abstracción “sujeto”.

¹¹ Para identificar que ha cambiado y actuar en consecuencia.

¹² En este ejemplo vemos como los observadores no sólo consultan el estado del sujeto, sino que al aceptar una puja están modificando al objeto observado. Lo que en este caso lanza la difusión del nuevo valor del objeto subastado.

3. Modelar la funcionalidad dependiente como una jerarquía de “observadores”.
4. El Sujeto se acopla con los Observadores sólo a través de su clase base.
5. El cliente configura el número y tipo de Observadores.
6. Los Observadores se registran ellos mismo en el Sujeto.
7. El Sujeto difunde los eventos a todos los Observadores registrados.
8. El Sujeto puede “enviar” la información a los Observadores, o, los Observadores pueden extraer la información que necesitan del Sujeto¹³.

Reglas de oro

- Chain of Responsibility, Command, Mediator y Observer, se encargan de desacoplar el transmisores y receptores, pero con diferentes acuerdos. Command normalmente pasa una petición de un emisor a lo largo de una cadena de potenciales receptores. Command normalmente especifica una conexión emisor-receptor con una subclase. Mediator ofrece referencias indirectas entre emisores y receptores. Observer define una interfaz desacoplada que permite que múltiples receptores puedan ser configurados en tiempo de ejecución.
- Mediator y Observer son patrones que compiten entre sí. La diferencia entre ellos es que Observer distribuye la comunicación introduciendo objetos “observador” y “sujeto”, mientras que el objeto Mediator encapsula la comunicación entre otros objetos. Los autores de sourcemaking.com consideran más reutilizables Observers y Subjects que los Mediators.
- En la otra mano, Mediator puede aprovechar Observer para registrar dinámicamente “colegas” y comunicarse con ellos.

¹³ En este último caso, el Sujeto al menos informa a los Observadores de que se produjo algún cambio