Patrón – Concepto

“Un patrón describe un problema el cual ocurre una y otra vez en nuestro ambiente, y además describen el núcleo de la solución a tal problema, en tal una manera que puedes usar esta solución millones de veces, sin hacer lo mismo dos veces.” [Alexander et al.]
Aunque Alexander se refería a patrones en ciudades y edificios, lo que dice también es válido para patrones de diseño orientados a objetos.

Patrones de Diseño – Concepto

Los patrones de diseño tratan los problemas del diseño de software que se repiten y que se presentan en situaciones particulares, con el fin de proponer soluciones a ellas. Son soluciones exitosas a problemas comunes.
Estas soluciones han ido evolucionando a través del tiempo. Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones se llaman estrategias.
En resumen, son soluciones simples y elegantes a problemas específicos del diseño de software orientado a objetos.

Historia

En 1994 se publicó el libro "Design Patterns: Elements of Reusable Object Oriented Software" escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Ellos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos. Ellos no fueron los únicos que inventaron los patrones, pero la idea de patrones de diseño comenzó a tomar fuerza luego de la publicación de dicho libro.

Cuándo utilizarlos

Como analistas y programadores vamos desarrollando a diario nuestras habilidades para resolver problemas usuales que se presentan en el desarrollo del software. Por cada problema que se nos presenta pensamos distintas formas de resolverlo, incluyendo soluciones exitosas que ya hemos usado anteriormente en problemas similares.
Antes de comenzar nuestro diseño, deberíamos realizar un estudio meticuloso del problema en el que nos encontramos y explorarlo en busca de patrones que hayan sido utilizados previamente con éxito. Estos patrones nos ayudarán a que nuestro proyecto evolucione mucho más rápidamente.

Cuándo no utilizarlos

Si hay una sensación que describa lo que puede sentir un desarrollador o diseñador después de conocer los

patrones de diseño esa podría ser la euforia. Esta euforia viene producida por haber encontrado un mecanismo que convierte lo que era una labor artesana y tediosa, en un proceso sólido y basado en estándares, mundialmente conocido y con probado éxito.
Después de recién iniciarse en los patrones, probablemente el paso siguiente que tomará será emprender el rediseño de algunos proyectos aún vigentes y que intente aplicar todas las maravillosas técnicas que ha aprendido para que así estos proyectos se aprovechen de todos los beneficios inherentes al uso de patrones de diseño.
Esto termina en intentar que se aplique estos patrones en toda situación donde sea posible, aún en aquellas donde no deba aplicarse.

Dónde utilizarlos

No es obligatorio utilizar los patrones siempre, sólo en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. Abusar o forzar el uso de los patrones es un error muy común.
Antes de abordar un proyecto con patrones se debe analizar minuciosamente qué patrones nos pueden ser útiles, cuáles son las relaciones entre los diferentes componentes de nuestro sistema, cómo podemos relacionar los patrones entre sí de modo que formen una estructura sólida, cuáles son los patrones que refleja nuestro dominio, etc. Esta tarea obviamente es mucho más compleja que el mero hecho de ponerse a codificar patrones "porque sí". Lo ideal es que los patrones se vean plasmados en el lenguaje de modelado UML, que veremos en la próxima sección.

Qué es GOF

Los ahora famosos Gang of Four (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Son los autores del famoso libro "Design Patterns: Elements of Reusable Object Oriented Software".
Aquí una breve reseña de cada uno de ellos:
Erich Gamma, informático suizo, es actualmente el director del centro tecnológico OTI en Zúrich y lidera el desarrollo de la plataforma Eclipse. Fue el creador de JUnit, junto a Kent Beck.
Richard Helm trabaja actualmente con The Boston Consulting Group, donde se desempeña como consultor de estrategias de IT aplicadas al mundo de los negocios.
Su carrera abarca la investigación de distintas tecnologías, desarrollo de productos, integración de sistemas y consultoría de IT. Antes de incorporarse a BCG, Richard trabajó en IBM: comenzó su carrera como científico investigador en "IBM Thomas J. Watson Research Center" en Nueva York. Es una autoridad internacional en arquitectura y diseño de software.
Ralph E. Johnson es profesor asociado en la Universidad de Illinois, en donde tiene a su cargo el Departamento de Ciencias de la Computación. Co-autor del famoso libro de Gof y pionero de la comunidad de Smalltalk, lidera el UIUC patterns/Software Architecture Group.

John Vlissides (1961-2005) era ingeniero eléctrico y se desempeñaba como consultor de la Universidad de

Standford. Autor de muchos libros, desde 1991 trabajó como investigador en el "IBM T.J. Watson Research Center".

Beneficios

Proporcionan elementos reusables en el diseño de sistemas software, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.
Efectividad comprobada en la resolución de problemas similares en ocasiones anteriores.
Formalizan un vocabulario común entre diseñadores.
Estandarizan el diseño, lo que beneficia notablemente a los desarrolladores.
Facilitan el aprendizaje de las nuevas generaciones de diseñadores y desarrolladores utilizando conocimiento ya existente.

Tipos de patrones

- Creacionales
- Comportamiento
- Estructurales

Clasificación según su propósito

Creacionales

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

Comportamiento

Permiten definir la comunicación entre los objetos del sistema y el flujo de la información entre los mismos.

Estructurales

Permiten crear grupos de objetos para ayudarnos a realizar tareas complejas.

Clasificación según alcance

De clase

Se basados en la herencia de clases.

De objeto

Se basan en la utilización dinámica de objetos.

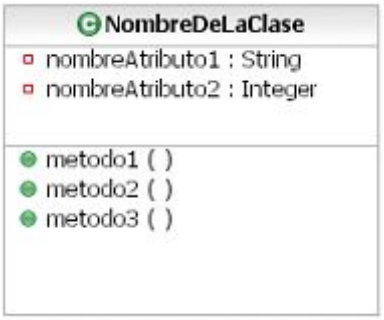
Qué es UML

UML significa Unified Modeling Language. Se trata de un conjunto de diagramas que buscan capturar una "perspectiva" de un sistema informático: por ejemplo, un diagrama está destinado a documentar los requerimientos del sistema y otro esta orientado a seguir el ciclo de vida de un determinado objeto.

Diagrama de Clases

Es el diagrama más importante ya que en él se diseñan las clases que serán parte de nuestro sistema. Es el esqueleto de nuestra aplicación.

Una clase se representa con un rectángulo de la siguiente forma:



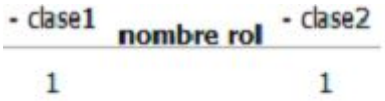
Simbología General

La simbología utilizada en el diagrama de clases es la siguiente:
- : privado
+ : público
# : protegido
subrayados: de clase
Muchas herramientas de diseño utilizan una simbología que consideran más amigable, basada en colores: por ejemplo el color rojo denota que un atributo/método es privado y el color verde que es público.

Asociación

Es una relación genérica entre dos clases que representa un enlace entre los objetos.
Se caracterizan por tener un nombre (nombre de la relación) y una cardinalidad (también denominada multiplicidad de la relación).
Las asociaciones se representan con líneas rectas sobre las cuales se puede escribir un texto descriptivo o rol

de la relación, así también como el grado de multiplicidad.



Generalización

Indica que una clase “hereda” atributos y métodos de otra, es decir, que es “hija” de la superclase a la cual se apunta. Se representa con:

Agregación

Es una relación que indica que un objeto es un componente o parte de otro objeto. Aún así, hay independencia entre ellos. Por ejemplo, Persona y Domicilio.
Se simboliza mediante:

Composición

Es una relación más fuerte que la agregación. A diferencia del caso anterior, aquí no hay independencia, por lo cual, las partes existen sólo si existe la otra. Se representa con:

Multiplicidad

Representa la cardinalidad que posee una clase con respecto a otra. Es un concepto muy similar a la multiplicidad que existe entre las distintas entidades de una base de datos.
Los distintos tipos de multiplicidad pueden representarse de la siguiente manera:

Uno a uno: 1 _____ 1

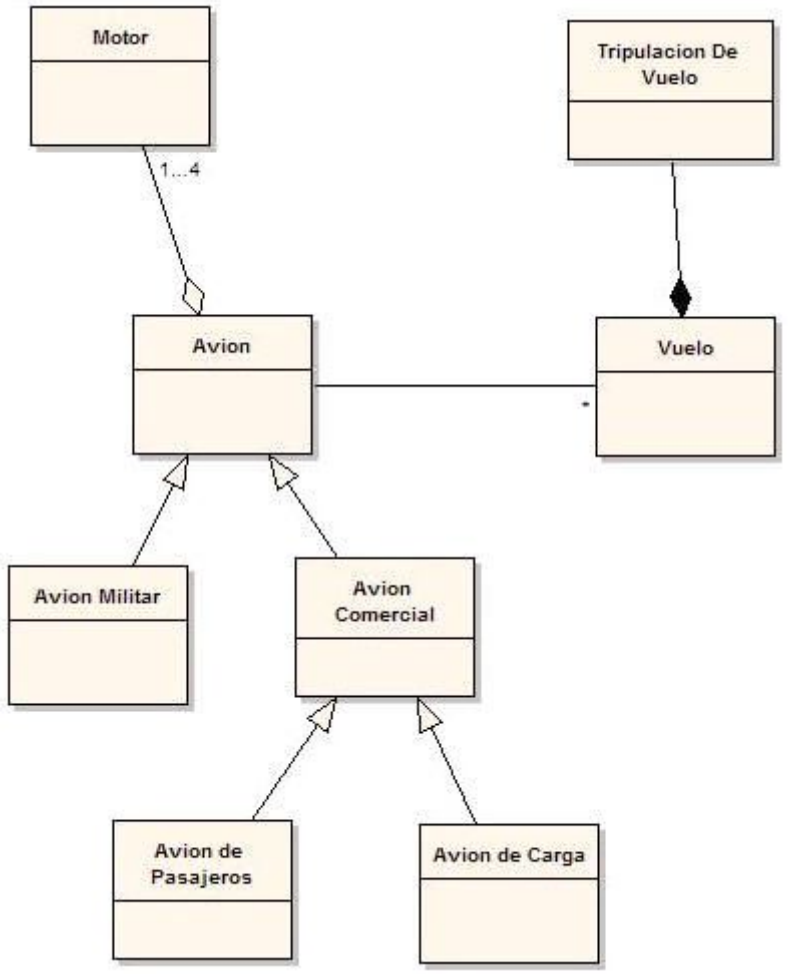
Uno a muchos: 1 _____ *

Uno a uno o más: 1 _____ 1..*

Uno a ninguno o a uno: 1 _____ 0..1

Combinaciones: 0..1, 3..4, 6..*

Ejemplo de Diagrama de clases



Singleton Pattern

Introducción y nombre

Singleton. Creacional. La idea del patrón Singleton es proveer un mecanismo para limitar el número de instancias de una clase. Por lo tanto el mismo objeto es siempre compartido por distintas partes del código. Puede ser visto como una solución más elegante para una variable global porque los datos son abstraídos por detrás de la interfaz que publica la clase singleton.

Intención

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

También conocido como

Algunas clases sólo pueden tener una instancia. Una variable global no garantiza que sólo se instancia una vez. Se utiliza cuando tiene que haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.

Motivación

Algunas clases sólo pueden tener una instancia. Una variable global no garantiza que sólo se instancia una vez. Se utiliza cuando tiene que haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.

Solución

Usaremos este patrón cuando:

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

Se requiere de un acceso estandarizado y conocido públicamente.

Diagrama UML

Participantes

Singleton: la clase que es Singleton define una instancia para que los clientes puedan accederla. Esta instancia es accedida mediante un método de clase.

Colaboraciones

Clientes: acceden a la única instancia mediante un método llamado getInstance().

Consecuencias

El patrón Singleton tiene muchas ventajas:

Acceso global y controlado a una única instancia: dado que hay una única instancia, es posible mantener un estricto control sobre ella.

Es una mejora a las variables globales: los nombres de las variables globlales no siguen un estándar para su acceso. Esto obliga al desarrollador a conocer detalles de su implementación.

Permite varias instancias de ser necesario: el patrón es lo suficientemente configurable como para configurar

más de una instancia y controlar el acceso que los clientes necesitan a dichas instancias.

Implementación

Privatizar el constructor

Definir un método llamado getInstance() como estático

Definir un atributo privado como estático.

Pueden ser necesarias operaciones de terminación (depende de la gestión de memoria del lenguaje)

En ambientes concurrentes es necesario usar mecanismos que garanticen la atomicidad del método getInstance(). En Java esto se puede lograr mediante la sincronización de un método.

Según el autor que se lea el método getInstance() también puede llamarse instance() o Instance()

Código de muestra

[code]
```
public class Singleton {
	private static Singleton instance;

	private Singleton(){
	}

	public synchronized static Singleton getInstance(){
		if (instance == null) {
			instance = new Singleton();
		}
		return instance;
	}
}
```

La forma de invocar a la clase Singleton sería:

```
public static void main(String[] args) {
		Singleton s2 = new Singleton();
		// lanza un error ya que el constructor esta privatizado.

		Singleton s1 = Singleton.getInstance();
		// forma correcta de convocar al Singleton
```
[/code]

Cuándo utilizarlo

Sus usos más comunes son clases que representan objetos unívocos. Por ejemplo, si hay un servidor que necesita ser representado mediante un objeto, este debería ser único, es decir, debería existir una sola instancia y el resto de las clases deberían de comunicarse con el mismo servidor. Un Calendario, por ejemplo, también es único para todos.

No debe utilizarse cuando una clase esta representando a un objeto que no es único, por ejemplo, la clase Persona no debería ser Singleton, ya que representa a una persona real y cada persona tiene su propio nombre, edad, domicilio, DNI, etc.

Template Method Pattern

Introducción y nombre

Template Method. De Comportamiento. Define una estructura algorítmica.

Intención

Escribe una clase abstracta que contiene parte de la lógica necesaria para realizar su finalidad. Organiza la clase de tal forma que sus métodos concretos llaman a un método abstracto donde la lógica buscada tendría que aparecer. Facilita la lógica buscada en métodos de subclases que sobrescriben a los métodos abstractos. Define un esqueleto de un algoritmo, delegando algunos pasos a las subclases. Permite redefinir parte del algoritmo sin cambiar su estructura.

También conocido como

Método plantilla.

Motivación

Usando el Template Method, se define una estructura de herencia en la cual la superclase sirve de plantilla ("Template" significa plantilla) de los métodos en las subclases. Una de las ventajas de este método es que evita la repetición de código, por tanto la aparición de errores.

Solución

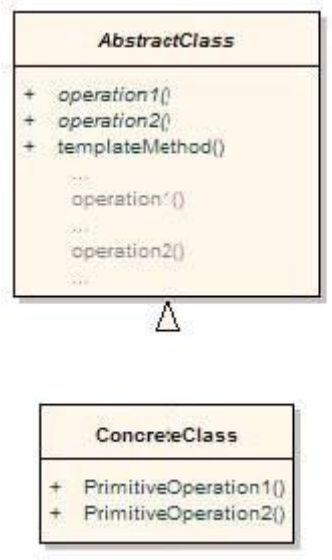
Se debe utilizar este patrón cuando:

Se quiera factorizar el comportamiento común de varias subclases.

Se necesite implementar las partes fijas de un algoritmo una sola vez y dejar que las subclases implementen las partes variables.

Se busque controlar las ampliaciones de las subclases, convirtiendo en métodos plantillas aquéllos métodos que pueden ser redefinidos.

Diagrama UML



Participantes

AbstractTemplate o AbstractClass: implementa un método plantilla que define el esqueleto de un algoritmo y define métodos abstractos que deben implementar las subclases concretas

TemplateConcreto o ConcreteClass: implementa los métodos abstractos para realizar los pasos del algoritmo que son específicos de la subclase.

Colaboraciones

Las clases concretas confían en que la clase abstracta implemente la parte fija del algoritmo.

Consecuencias

Favorece la reutilización del código.

Lleva a una estructura de control invertido: la superclase base invoca los métodos de las subclases.

Implementación

Una clase ClaseAbstracta proporciona la guía para forzar a los programadores a sobrescribir los métodos abstractos con la intención de proporcionar la lógica que rellena los huecos de la lógica de su método plantilla.

Los métodos plantilla no deben redefinirse. Los métodos abstractos deben ser protegidos (accesible a las subclases pero no a los clientes) y abstractos. Se debe intentar minimizar el número de métodos abstractos a fin de facilitar la implementación de las subclases.

Código de muestra

Como ejemplo, imaginemos una empresa que posee socios, clientes, empleados, etc. Cuando se les solicite que se identifiquen, cada uno lo realizará de distinta manera: quizás un empleado tiene un legajo, pero un cliente tiene un numero de cliente, etc.

[code]

```
public abstract class Persona {

    private String nombre;
    private String dni;

    public Persona() {
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }
```

```
    /*
     * Define el esquelo del algoritmo y luego las subclases deben
     * implementar los métodos: getIdentificacion() y getTipoId()
     */
    public String identificate() {
        String frase = "Me identifico con: ";
        frase = frase + getTipoId();
        frase = frase + ". El numero es: ";
        frase = frase + getIdentificacion();
        return frase;
    }

    protected abstract String getIdentificacion();

    protected abstract String getTipoId();
}



public class Socio extends Persona {

    private int numeroDeSocio;

    public Socio(int numeroDeSocio) {
        this.numeroDeSocio = numeroDeSocio;
    }

    @Override
    protected String getIdentificacion() {
        return String.valueOf(numeroDeSocio);
    }

    @Override
    protected String getTipoId() {
        return "numero de socio";
    }

    public int getNumeroDeSocio() {
        return numeroDeSocio;
    }

    public void setNumeroDeSocio(int numeroDeSocio) {
        this.numeroDeSocio = numeroDeSocio;
    }
}


public class Cliente extends Persona {

    private int numeroDeCliente;

    public Cliente(int numeroDeCliente) {
        this.numeroDeCliente = numeroDeCliente;
    }

    @Override
    protected String getIdentificacion() {
        return String.valueOf(numeroDeCliente);
    }

    @Override
    protected String getTipoId() {
        return "numero de cliente";
    }

    public int getNumeroDeCliente() {
```

```
        return numeroDeCliente;
    }

    public void setNumeroDeCliente(int numeroDeCliente) {
        this.numeroDeCliente = numeroDeCliente;
    }
}


public class Empleado extends Persona {

    private String legajo;

    public Empleado(String legajo) {
        this.legajo = legajo;
    }

    @Override
    protected String getIdentificacion() {
        return legajo;
    }

    @Override
    protected String getTipoId() {
        return "numero de legajo";
    }

    public String getLegajo() {
        return legajo;
    }

    public void setLegajo(String legajo) {
        this.legajo = legajo;
    }
}
```

Veamos el ejemplo en práctica:

```
                public static void main(String[] args) {
                Persona p = new Cliente(12121);
                System.out.println("El cliente dice: ");
                System.out.println(p.identificate());

                System.out.println("El empleado dice: ");
                p = new Empleado("AD 41252");
                System.out.println(p.identificate());

                System.out.println("El socio dice: ");
                p= new Socio(46232);
                System.out.println(p.identificate());
        }
```

El resultado por consola es:
El cliente dice:
Me identifico con: numero de cliente. El numero es: 12121

El emplado dice:
Me identifico con: numero de legajo. El numero es: AD 41252

El socio dice:
Me identifico con: numero de socio. El numero es: 46232

[/code]

Cuándo utilizarlo

Este patrón se vuelve de especial utilidad cuando es necesario realizar un algoritmo que sea común para muchas clases, pero con pequeñas variaciones entre una y otras. En este caso, se deja en las subclases cambiar una parte del algoritmo.

Decorator Pattern

Introducción y nombre

Decorator, Estructural. Añade dinámicamente funcionalidad a un objeto.

Intención

El patrón decorator permite añadir responsabilidades a objetos concretos de forma dinámica. Los decoradores ofrecen una alternativa más flexible que la herencia para extender las funcionalidades.

También conocido como

Decorador, Wrapper (igual que el patrón Adapter).

Motivación

A veces se desea adicionar responsabilidades a un objeto pero no a toda la clase. Las responsabilidades se pueden adicionar por medio de los mecanismos de Herencia, pero este mecanismo no es flexible porque la responsabilidad es adicionada estáticamente. La solución flexible es la de rodear el objeto con otro objeto que es el que adiciona la nueva responsabilidad. Este nuevo objeto es el Decorator.

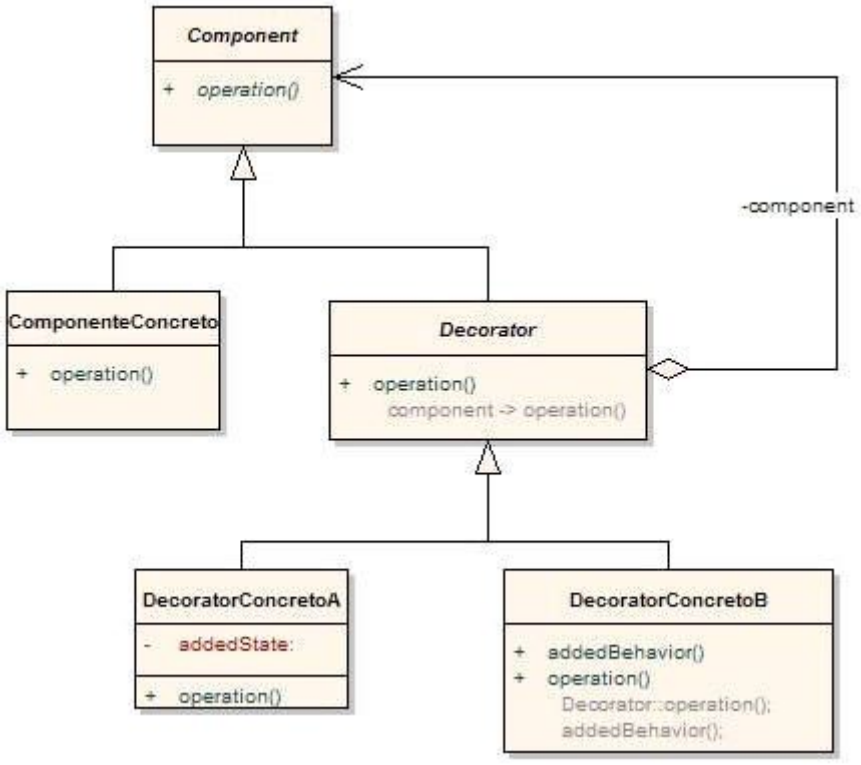
Solución

Este patrón se debe utilizar cuando:

Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.

Se quiere agregar o quitar dinámicamente la funcionalidad de un objeto.

Diagrama UML



Participantes

Component: define la interface de los objetos a los que se les pueden adicionar responsabilidades dinámicamente.

ComponenteConcreteo: define el objeto al que se le puede adicionar una responsabilidad.

Decorator: mantiene una referencia al objeto Component y define una interface de acuerdo con la interface de Component.

DecoratorConcreto: adiciona la responsabilidad al Component.

Colaboraciones

Decorator propaga los mensajes a su objeto Component. Opcionalmente puede realizar operaciones antes y después de enviar el mensaje.

Consecuencias

Es más flexible que la herencia: utilizando diferentes combinaciones de unos pocos tipos distintos de objetos decorator, se puede crear muchas combinaciones distintas de comportamientos. Para crear esos diferentes tipos de comportamiento con la herencia se requiere que definas muchas clases distintas.

Evita que las clases altas de la jerarquía estén demasiado cargadas de funcionalidad.

Un componente y su decorador no son el mismo objeto.

Provoca la creación de muchos objetos pequeños encadenados, lo que puede llegar a complicar la depuración.

La flexibilidad de los objetos decorator los hace más propenso a errores que la herencia. Por ejemplo, es posible combinar objetos decorator de diferentes formas que no funcionen, o crear referencias circulares entre los objetos decorator.

Implementación

La mayoría de las implementaciones del patrón Decorator son sencillas. Veamos algunas de las implementaciones más comunes:

Si solamente hay una clase ComponenteConcreto y ninguna clase Component, entonces la clase Decorator es normalmente una subclase de la clase ComponenteConcreto.

A menudo el patrón Decorator es utilizado para delegar a un único objeto. En este caso, no hay necesidad de tener la clase Decorator (abstracto) para mantener una colección de referencias. Sólo conservando una única referencia es suficiente.

Por otro lado, se debe tener en cuenta que un decorador y su componente deben compartir la misma interfaz. Los componentes deben ser clases con una interfaz sencilla.

Muchas veces se el decorator cabeza de jerarquía puede incluir alguna funcionalidad por defecto.

Código de muestra

Imaginemos que vendemos automóviles y el cliente puede opcionalmente adicionar ciertos componentes (aire acondicionado, mp3 player, etc). Por cada componente que se adiciona, el precio varía.

[code]

```
public interface Vendible {
   public String getDescripcion();

   public int getPrecio();
}

public abstract class Auto implements Vendible {
}


public class FiatUno extends Auto {

   @Override
   public String getDescripcion() {
      return "Fiat Uno modelo 2006";
   }

   @Override
   public int getPrecio() {
      return 15000;
   }
}

public class FordFiesta extends Auto {
   @Override
   public String getDescripcion() {
      return "Ford Fiesta modelo 2008";
   }

   @Override
   public int getPrecio() {
      return 25000;
   }
}

public abstract class AutoDecorator implements Vendible {
   private Vendible vendible;

   public AutoDecorator(Vendible vendible) {
      this.vendible = vendible;
   }
```

```
    public Vendible getVendible() {
        return vendible;
    }

    public void setVendible(Vendible vendible) {
        this.vendible = vendible;
    }
}

public class CdPlayer extends AutoDecorator {

    public CdPlayer(Vendible vendible) {
        super(vendible);
    }

    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + CD Player";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 100;
    }
}

public class AireAcondicionado extends AutoDecorator {

    public AireAcondicionado(Vendible vendible) {
        super(vendible);
    }

    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Aire Acondicionado";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 1500;
    }
}

public class Mp3Player extends AutoDecorator {

    public Mp3Player(Vendible vendible) {
        super(vendible);
    }

    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + MP3 Player";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 250;
    }
}

public class Gasoil extends AutoDecorator {

    public Gasoil(Vendible vendible) {
        super(vendible);
    }
```

```
    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Gasoil";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 1200;
    }
}
```

Probemos el funcionamiento del ejemplo:

```
        public static void main(String[] args) {
                Vendible auto = new FiatUno();
                auto = new CdPlayer(auto);
                auto = new Gasoil(auto);

                System.out.println(auto.getDescripcion());
                System.out.println("Su precio es: "+ auto.getPrecio());

                Vendible auto2 = new FordFiesta();
                auto2 = new Mp3Player(auto2);
                auto2 = new Gasoil(auto2);
                auto2 = new AireAcondicionado(auto2);

                System.out.println(auto2.getDescripcion());
                System.out.println("Su precio es: "+ auto2.getPrecio());
        }
```

La salida por consola es:
Fiat Uno modelo 2006 + CD Player + Gasoil
Su precio es: 16300
Ford Fiesta modelo 2008 + MP3 Player + Gasoil + Aire Acondicionado
Su precio es: 27950

[/code]

Cuándo utilizarlo

Dado que este patrón decora un objeto y le agrega funcionalidad, suele ser muy utilizado para adicionar opciones de "embellecimiento" en las interfaces al usuario. Este patrón debe ser utilizado cuando la herencia de clases no es viable o no es útil para agregar funcionalidad. Imaginemos que vamos a comprar una PC de escritorio. Una estándar tiene un precio determinado. Pero si le agregamos otros componentes, por ejemplo, un lector de CD, el precio varía. Si le agregamos un monitor LCD, seguramente también varía el precio. Y con cada componente adicional que le agreguemos al estándar, seguramente el precio cambiará. Este caso, es un caso típico para utilizar el Decorator.

Facade Pattern

Introducción y nombre

Facade, Estructural. Busca simplificar el sistema, desde el punto de vista del cliente.

Intención

Su intención es proporcionar una interfaz unificada para un conjunto de subsistemas, definiendo una interfaz de nivel más alto. Esto hace que el sistema sea más fácil de usar.

También conocido como

Fachada.

Motivación

Este patrón busca reducir al mínimo la comunicación y dependencias entre subsistemas. Para ello, utilizaremos una fachada, simplificando la complejidad al cliente. El cliente debería acceder a un subsistema a través del Facade. De esta manera, se estructura un entorno de programación más sencillo, al menos desde el punto de vista del cliente (por ello se llama "fachada").

Solución

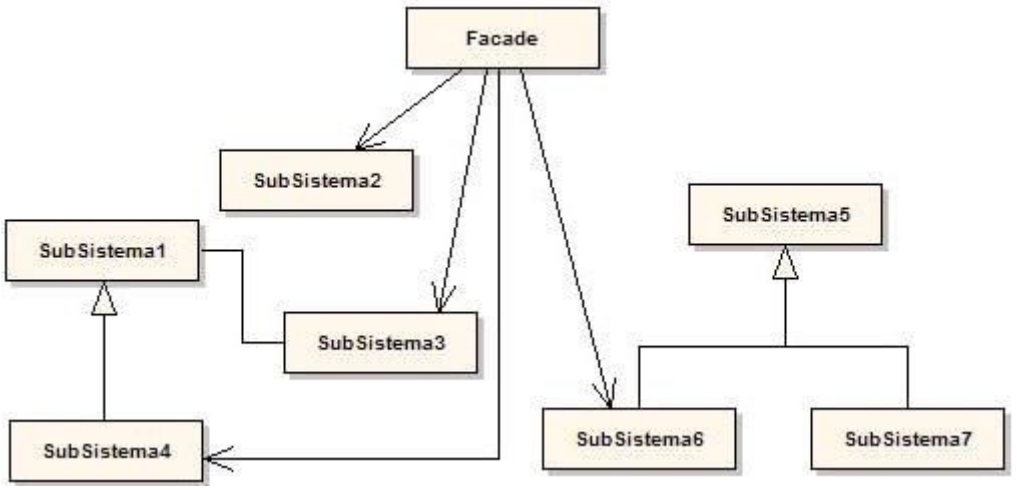
Este patrón se debe utilizar cuando:

Se quiera proporcionar una interfaz sencilla para un subsistema complejo.

Se quiera desacoplar un subsistema de sus clientes y de otros subsistemas, haciéndolo más independiente y portable.

Se quiera dividir los sistemas en niveles: las fachadas serían el punto de entrada a cada nivel. Facade puede ser utilizado a nivel aplicación.

Diagrama UML



Participantes

Facade:

Conoce cuales clases del subsistema son responsables de una petición.

Delega las peticiones de los clientes en los objetos del subsistema.

Subsistema:

Implementar la funcionalidad del subsistema.

Manejar el trabajo asignado por el objeto Facade.

No tienen ningún conocimiento del Facade (no guardan referencia de éste).

Colaboraciones:

Los clientes se comunican con el subsistema a través de la facade, que reenvía las peticiones a los objetos del subsistema apropiados y puede realizar también algún trabajo de traducción

Los clientes que usan la facade no necesitan acceder directamente a los objetos del sistema.

Consecuencias

Oculta a los clientes de la complejidad del subsistema y lo hace más fácil de usar.

Favorece un acoplamiento débil entre el subsistema y sus clientes, consiguiendo que los cambios de las clases del sistema sean transparentes a los clientes.

Facilita la división en capas y reduce dependencias de compilación.

No se impide el acceso a las clases del sistema.

Implementación

Es un patrón muy sencillo de utilizar. Se debe tener en cuenta que no siempre es tan sólo un "pasamanos". Si fuese necesario que el Facade realice una tarea específica antes de devolver una respuesta al cliente, podría hacerlo sin problema.

Lo más importante de todo es que este patrón se debe aplicar en las clases más representativas y no en las específicas. De no ser así, posiblemente no se tenga el nivel alto deseado.

Por aplicación, es ideal construir no demasiados objetos Facade. Sólo algunos representativos que contengan la mayoría de las operaciones básicas de un sistema.

Código de muestra

Imaginemos que estamos, con un equipo de desarrollo, realizando el software para una inmobiliaria. Obviamente una inmobiliaria realiza muchos trabajos diferentes, como el cobro de alquiler, muestra de inmuebles, administración de consorcios, contratos de ventas, contratos de alquiler, etc.

Por una cuestión de seguir el paradigma de programación orientada a objetos, es probable que no se realice todo a una misma clase, sino que se dividen las responsabilidades en diferentes clases.

[code]

```
public class Persona {
}
public class Cliente extends Persona {
}
public class Interesado extends Persona {
}
public class Propietario extends Persona {
}

public class AdministracionAlquiler {
   public void cobro(double monto) {
      // Algoritmo
   }
}
```

```
public class CuentasAPagar {
    public void pagoPropietario(double monto) {
        // Algoritmo
    }
}

public class MuestraPropiedad {
    public void mostraPropiedad(int numeroPropiedad) {
        // Algoritmo
    }
}

public class VentaInmueble {
    public void gestionaVenta() {
        // Algoritmo
    }
}

public class Inmobiliaria {

    private MuestraPropiedad muestraPropiedad;
    private VentaInmueble venta;
    private CuentasAPagar cuentasAPagar;
    private AdministracionAlquiler alquiler;

    public Inmobiliaria() {
        muestraPropiedad = new MuestraPropiedad();
        venta = new VentaInmueble();
        cuentasAPagar = new CuentasAPagar();
        alquiler = new AdministracionAlquiler();
    }

    public void atencionCliente(Cliente c) {
        System.out.println("Atendiendo a un cliente");
    }

    public void atencionPropietario(Propietario p) {
        System.out.println("Atendiendo a un propietario");
    }

    public void atencionInteresado(Interesado i) {
        System.out.println("Atencion a un interesado en una propiedad");
    }

    public void atencion(Persona p) {
        if (p instanceof Cliente) {
            atencionCliente((Cliente) p);
        } else if (p instanceof Propietario) {
            atencionPropietario((Propietario) p);
        } else {
            atencionInteresado((Interesado) p);
        }
    }

    public void mostraPropiedad(int numeroPropiedad) {
        muestraPropiedad.mostraPropiedad(numeroPropiedad);
    }

    public void gestionaVenta() {
        venta.gestionaVenta();
    }

    public void paga(int monto) {
        cuentasAPagar.pagoPropietario(monto);
    }
```

```
    public void cobraAlquiler(double monto) {
        alquiler.cobro(monto);
    }
}
```

Probemos como es el funcionamiento del Facade:

```
public class Main {

    public static void main(String[] args) {
        Cliente c = new Cliente();
        Interesado i = new Interesado();
        Inmobiliaria inmo = new Inmobiliaria();
        inmo.atencionCliente(c);
        inmo.atencionInteresado(i);

        MuestraPropiedad muestraPropiedad = new MuestraPropiedad();
        muestraPropiedad.mostraPropiedad(123);
        VentaInmueble venta = new VentaInmueble();
        venta.gestionaVenta();
        AdministracionAlquiler alquiler = new AdministracionAlquiler();
        alquiler.cobro(1200);
        CuentasAPagar cuentasAPagar = new CuentasAPagar();
        cuentasAPagar.pagoPropietario(1100);

        // Lo mismo pero despues del Facade
        Inmobiliaria inmo2 = new Inmobiliaria();
        inmo2.atencion(i);
        inmo2.atencion(c);
        inmo2.mostraPropiedad(123);
        inmo2.gestionaVenta();
        inmo2.cobraAlquiler(1200);
        inmo2.paga(1100);

    }
}
```
[/code]

Cuándo utilizarlo

Sabemos que el Facade busca reducir la complejidad de un sistema. Esto mismo ocurre en ciertos lugares donde tendremos muchas opciones: imaginemos a un banco o edificio público. Es un lugar donde cada persona hace trámites distintos y, por ende, cada persona se encuentra con complicaciones de distinta índole. Es complicado para las personas que trabajan en dichos lugares, por ello es que tienen un especialista por cada tema. Por ello, es muy raro que el cajero sea la misma persona que gestiona un préstamo hipotecario.

Esta misma complejidad se traspasa para el cliente: cuando entramos a un lugar grande con muchas ventanillas, es posible que hagamos la fila en el lugar incorrecto.

¿Cómo se soluciona este caos? Colocando un mostrador de información en la entrada para que todos los clientes vayan directamente al mostrador y allí se va direccionando a las personas al lugar correcto.

A grandes rasgos, se podría decir que el mostrador de información cumple un rol similar a un Facade: todos los clientes se dirigen allí y el se encarga de solucionarnos el problema. En realidad, sabe quién es la persona que lo va a solucionar.

En los proyectos grandes suele ocurrir que se pierde el control de la cantidad de clases y cuando este ocurre, no es bueno obligar a todos los clientes a conocer los subsistemas. Este caso, es un caso ideal para aplicar un Facade.