Tabla de contenidos

1. Introducción

1.1. Objetivos

1.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

1.2. 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.

1.3. 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.

1.4. 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.

1.5. Cuando 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.

1.6. Cuando 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.

1.7. Donde 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.

1.8. Que 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".

1.9. 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.

2. Tipos de Patrones

2.1. Objetivos

2.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

2.2. Clasificación según su propósito

2.2.1. 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.

2.2.2. De Comportamiento

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

2.2.3. Estructurales

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

2.3. Clasificación según alcance

2.3.1. De clase

Se basados en la herencia de clases.

2.3.2. De objeto

Se basan en la utilización dinámica de objetos.

2.4. Tabla de Clasificación

		Propósito		
		De Creación	**Estructural**	De Comportamiento
Alcance	**Clase**	Factory Method	Adapter	**Interpreter**
				Template Method
	Objeto	Abstract Factory	Adapter	**Chain of Responsability**
		Builder	Bridge	**Command**
		Prototype	Composite	**Iterator**
		Singleton	Decorator	**Mediator**
			Facade	**Memento**
			Flyweight	**Observer**
			Proxy	**State**
				Strategy
				Visitor

2.5. Especificación de patrones de diseño

Para todos los patrones de diseño se explicará la siguiente estructura:

1. Introducción y nombre: denota sutilmente la esencia del patrón.
2. Intención: narración que responde a qué hace y qué resuelve
3. También conocido como: otros nombres conocidos para el patrón.
4. Motivación: escenario y razón de porque es necesario el patrón.
5. Solución: situaciones en las que resulta aplicable.
6. Diagrama UML: estructura en un diagramas de clases.
7. Participantes: responsabilidad de cada clase participante.
8. Colaboraciones: descripción de las relaciones de los participantes.
9. Consecuencias: ventajas e inconvenientes.

10. Implementación: dificultades, técnicas y trucos a tener en cuenta al aplicar el PD

11. Código de muestra: ejemplo de implementación y de uso del PD

12. Cuando utilizarlo: ejemplos de uso en sistemas reales

13. Patrones relacionados: patrones con los que potencialmente puede interactuar.

3. UML Review

3.1. Objetivos

3.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

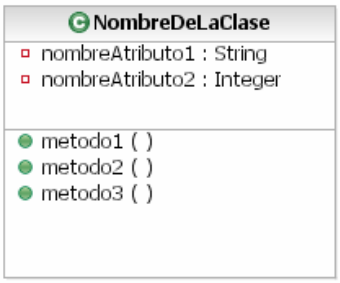
3.2. Que es UML

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.

3.3. 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:



3.3.1. Simbología General

La simbología utilizada en el diagrama de clases es la siguiente:

- : privado

+ : público

: protegido

subrayados: de clase

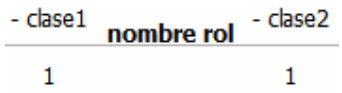
Muchas herramientas de diseño utilizan una simbología que consideran más amigable, basada en colores: por ejemplo el color rojo denota que un atributo/método es privado y el color verde que es público.

3.3.2. 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.



3.3.3. 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:

3.3.4. 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:

3.3.5. 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:

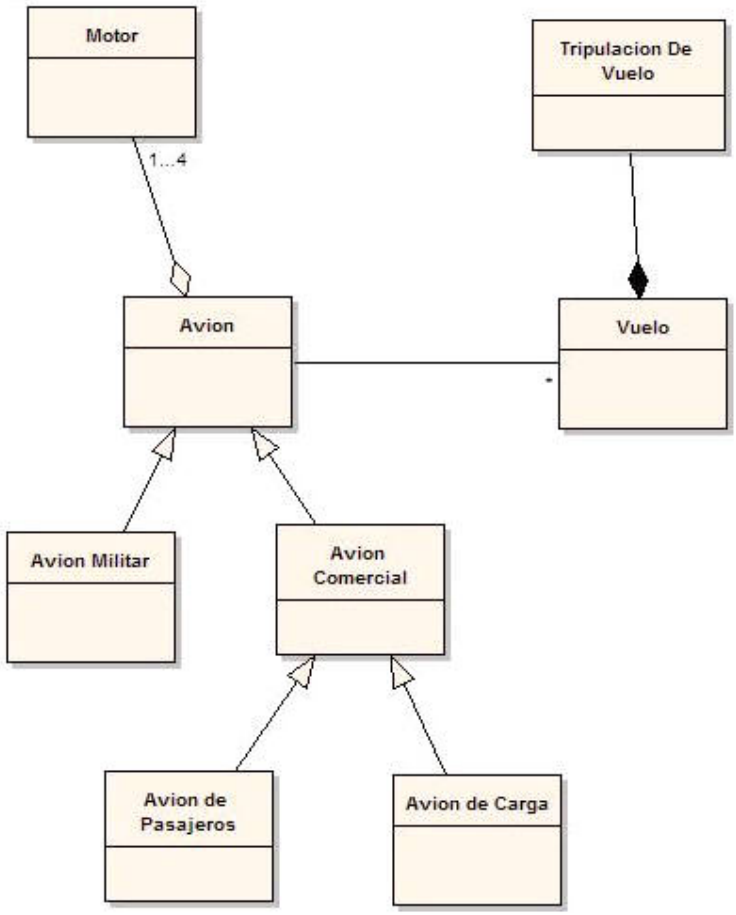
3.3.6. Multiplicidad

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 representase 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..*
- etc . . .

3.3.7. Ejemplo de Diagrama de clases



4. Patrones de Comportamiento (Behavioural Patterns)

4.1. Objetivos

4.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

4.2. Chain of Responsibility Pattern

4.2.1. Introducción y nombre

Chain of Responsibility. De comportamiento. El patrón de diseño Chain of Responsibility permite establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor.

4.2.2. Intención

Como se dijo anteriormente se busca establecer una cadena de receptores, donde cualquiera de ellos puede responder a la petición en función de un criterio establecido. Busca evitar un montón de if – else largos y complejos en nuestro código, pero sobre todas las cosas busca evitar que el cliente necesite conocer toda nuestra estructura jeráquica y que rol cumple cada integrante de nuestra estructura.

4.2.3. También conocido como

Cadena de responsabilidad.

4.2.4. Motivación

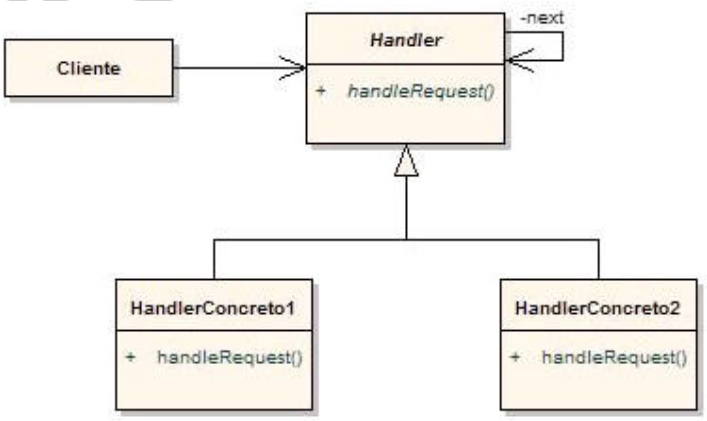
- En múltiples ocasiones, un cliente necesita que se realice una función, pero o no conoce al servidor concreto de esa función o es conveniente que no lo conozca para evitar un gran acoplamiento entre ambos.
- Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto. Este evita que el cliente deba conocer toda nuestra estructura de clases, ya que cualquiera le resuelve el problema.

4.2.5. Solución

Se utiliza cuando:

- Las peticiones emitidas por un objeto deben ser atendidas por distintos objetos receptores.
- No se sabe a priori cual es el objeto que me puede resolver el problema.
- Cuando un pedido debe ser manejado por varios objetos.
- El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

4.2.6. Diagrama UML



4.2.7. Participantes

- Handler: define una interfaz para tratar las peticiones. Implementa el enlace al sucesor.
- HandlerConcreto: trata las peticiones de las que es responsable. Si puede manejar la petición, lo hace, en caso contrario la reenvía a su sucesor.
- Cliente: inicializa la petición.

4.2.8. Colaboraciones

El cliente conoce a un gestor que es el que lanza la petición a la cadena hasta que alguien la recoge.

4.2.9. Consecuencias

- Reduce el acoplamiento.
- Añade flexibilidad para asignar responsabilidades a objetos.
- No se garantiza la recepción.

4.2.10. Implementación

Todos los objetos receptores implementarán la misma interfaz o extenderán la misma clase abstracta. En ambos casos se proveerá de un método que permita obtener el sucesor y así el paso de la petición por la cadena será lo más flexible y transparente posible.

La idea es crear un sistema que pueda servir a diversas solicitudes de manera jerárquica.

4.2.11. Código de muestra

Escenario: estamos realizando el software para un banco y uno de los puntos más importantes es saber quién puede aprobar un crédito. Por lo tanto el banco define las siguientes reglas de negocio:

Si el monto no supera los $ 10.000 entonces el ejecutivo de cuenta pueda aprobar el préstamo.

Si el monto esta entre los $10.000 y $50.000 entonces la persona indicada para realizar la aprobación es el líder inmediato de dicho ejecutivo.

Si el monto se encuentra entre $ 50.000 y $100.000 entonces es el Gerente quién debe realizar dicha aprobación.

Por montos superiores a los $100.000 entonces la aprobación la realizará el Director.

Para este caso se ha decidido realizar un patrón Chain of Responsibility. Se decide crear una interface llamada IAprobador que debe implmentar toda clase que pertenezca a nuestra cadena de responsabilidades.

```
public interface IAprobador {

      public void setNext(IAprobador aprobador);
      public IAprobador getNext();
      public void solicitudPrestamo(int monto);}
```

Ahora veamos las clases que son aprobadoras:

```
public class EjecutivoDeCuenta implements IAprobador {
     private IAprobador next;

     public IAprobador getNext() {
            return next;}

     public void solicitudPrestamo(int monto) {
            if (monto <= 10000) {
     System.out.println("Lo manejo yo, el ejecutivo de cuentas");
            } else {
                   next.solicitudPrestamo(monto);}      }

     public void setNext(IAprobador aprobador) {
            next = aprobador;}
}

public class LiderTeamEjecutivo implements IAprobador{
     private IAprobador next;

     public IAprobador getNext() {
            return next;}

     public void solicitudPrestamo(int monto) {
            if (monto > 10000 && monto <= 50000) {
                   System.out.println("Lo manejo yo, el lider");
            } else {
                   next.solicitudPrestamo(monto);}}

     public void setNext(IAprobador aprobador) {
            next = aprobador;      }
}


public class Gerente implements IAprobador {
     private IAprobador next;

     public IAprobador getNext() {
            return next;}

     public void solicitudPrestamo(int monto){
            if (monto > 50000 && monto <= 100000) {
                   System.out.println("Lo manejo yo, el gerente");
            } else {
                   next.solicitudPrestamo(monto);       }      }

     public void setNext(IAprobador aprobador) {
            next = aprobador;}
```

```
}


public class Director implements IAprobador {
    private IAprobador next;

    public IAprobador getNext() {
           return next;}

    public void solicitudPrestamo(int monto) {
           if (monto >= 100000) {
                  System.out.println("Lo manejo yo, el director");} }

    public void setNext(IAprobador aprobador) {
           next = aprobador;}
    }
```

Y, por último, el banco, que a fin de cuentas es quién decide las reglas del negocio.

```
public class Banco implements IAprobador {
    private IAprobador next;

    public IAprobador getNext() {
           return next;}

    public void solicitudPrestamo(int monto) {
           EjecutivoDeCuenta ejecutivo = new EjecutivoDeCuenta();
           this.setNext(ejecutivo);

           LiderTeamEjecutivo lider = new LiderTeamEjecutivo();
           ejecutivo.setNext(lider);

           Gerente gerente = new Gerente();
           lider.setNext(gerente);

           Director director = new Director();
           gerente.setNext(director);

           next.solicitudPrestamo(monto);}

    public void setNext(IAprobador aprobador) {
           next = aprobador;}}
```

Veamos como es su funcionamiento:

```
public static void main(String[] args) {
           Banco banco = new Banco();
           banco.solicitudPrestamo(56000);
    }
```

Y el resultado por consola es: Lo manejo yo, el gerente

4.2.12. Cuando utilizarlo

La motivación detrás de este patrón es crear un sistema que pueda servir a diversas solicitudes de manera jerárquica. En otras palabras, si un objeto que es parte de un sistema no sabe cómo responder a una solicitud, la pasa a lo largo del árbol de objetos. Como el nombre lo implica, cada objeto de dicho árbol puede tomar la responsabilidad y atender la solicitud.

Un ejemplo típico podría ser el lanzar un trabajo de impresión. El cliente no sabe siquiera qué impresoras están instaladas en el sistema, símplemente lanza el trabajo a la cadena de objetos que representan a las impresoras. Cada uno de ellos lo deja pasar, hasta que alguno, finalmente lo ejecuta.

Hay un desacoplamiento evidente entre el objeto que lanza el trabajo (el cliente) y el que lo realiza (impresora).

4.2.13. Patrones relacionados

Este patrón se suele aplicar junto con el patrón Composite. En él, los padres de los componentes pueden actuar como sucesores.

4.3. Command Pattern

4.3.1. Introducción y nombre

Command. De comportamiento. Especifica una forma simple de separar la ejecución de un comando, del entorno que generó dicho comando.

4.3.2. Intención

Permite solicitar una operación a un objeto sin conocer el contenido ni el receptor real de la misma. Encapsula un mensaje como un objeto.

4.3.3. También conocido como

Comando, Orden, Action, Transaction.

4.3.4. Motivación

Este patrón suele establecer en escenarios donde se necesite encapsular una petición dentro de un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolarlas, guardarlas en un registro de sucesos o implementar un mecanismo de deshacer/repetir.

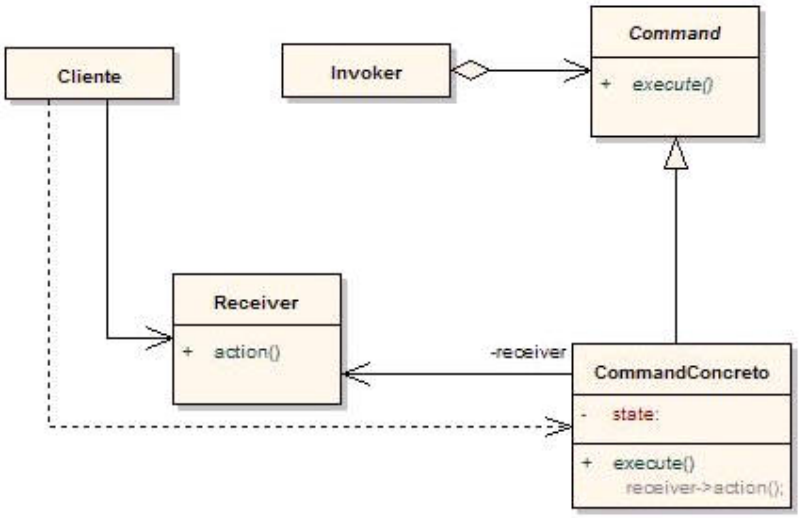
4.3.5. Solución

Dado que este patrón encapsula un mensaje como un objeto, este patrón debería ser usado cuando:

- Se necesiten colas o registros de mensajes.
- Tener la posibilidad de deshacer las operaciones realizadas.
- Se necesite uniformidad al invocar las acciones.

- Facilitar la parametrización de las acciones a realizar.
- Independizar el momento de petición del de ejecución.
- El parámetro de una orden puede ser otra orden a ejecutar.
- Desarrollar sistemas utilizando órdenes de alto nivel que se construyen con operaciones sencillas (primitivas).
- Se necesite sencillez al extender el sistema con nuevas acciones.

4.3.6. Diagrama UML



4.3.7. Participantes

Command: declara una interfaz para ejecutar una operación.

CommandConcreto: define un enlace entre un objeto "Receiver" y una acción. Implementa el método execute invocando la(s) correspondiente(s) operación(es) del "Receiver".

Cliente: crea un objeto "CommandConcreto" y establece su receptor.

Invoker: le pide a la orden que ejecute la petición.

Receiver: sabe como llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como receptor.

4.3.8. Colaboraciones

El cliente crea un objeto "CommandConcreto" y especifica su receptor.

Un objeto "Invoker" almacena el objeto "CommandConcreto".

El invocador envía una petición llamando al método execute sobre la orden.

El objeto "CommandConcreto", invoca operaciones de su receptor para llevar a cabo la petición.

4.3.9. Consecuencias

Command desacoplado: el objeto que invoca la operación de aquél que sabe como realizarla.

Las órdenes son objetos manipulados y extendidos de forma natural.

Se pueden ensamblar órdenes en una orden compuesta.

Facilidad de adición de nuevos objetos Command.

4.3.10. Implementación

La clave de este patrón es una clase abstracta o interfaz Command que define una operación execute. Son las subclases concretas quienes implementan la operación y especifican el receptor de la orden. Para ello debemos tener en cuenta:

- Permitir deshacer y repetir cuando sea necesario.
- Evitar la acumulación de errores en el proceso de deshacer.
- Los commands deberían invocar ordenes en el receptor.

4.3.11. Código de muestra

Escenario: una empresa maneja varios servidores y cada uno de ellos deben correr diversos procesos, como apagarse, prenderse, etc. Cada uno de estos procesos, a su vez, implican pequeños pasos como, por ejemplo, realizar una conexión a dicho servidor, guardar los datos en un log, etc.

Dado que cada servidor tiene su propia lógica para cada operación, se decidió crear una interfaz llamada Iserver que deben implementar los servidores:

```
public interface IServer {

    public void apagate();
    public void prendete();
    public void conectate();
    public void verificaConexion();
    public void guardaLog();
    public void cerraConexion();
}
```

Los Servidores son:

```
public class BrasilServer implements IServer {

    public void apagate() {
           System.out.println("Apagando el servidor de Brasil");       }

    public void cerraConexion() {
System.out.println("Cerrando conexion con el servidor de Brasil");}

    public void conectate() {
           System.out.println("Conectando al servidor de Brasil");     }

    public void guardaLog() {
    System.out.println("Guardar Log de Brasil");}

    public void prendete() {
           System.out.println("Prendiendo el servidor de Brasil");}

    public void verificaConexion() {
           System.out.println("Comprobando la conexion de Brasil");}
}
```

Obviamente en un caso real los métodos tendrían el algoritmo necesario para realizar tales operaciones. Hemos simplificado estos algoritmos y en su lugar realizamos una salida por consola en cada método.

```
public class USAServer implements IServer{
    public void apagate() {
           System.out.println("Apagando el servidor de USA");}

    public void cerraConexion() {
           System.out.println("Cerrando conexion con el servidor de USA");}
```

```
        public void conectate() {
                System.out.println("Conectando al servidor de USA");}

        public void guardaLog() {
        System.out.println("Guardar Log de USA");}

        public void prendete() {
                System.out.println("Prendiendo el servidor de USA");}

        public void verificaConexion() {
                System.out.println("Comprobando la conexion de USA");}
}

public class ArgentinaServer implements IServer {

        public void apagate() {
                System.out.println("Apagando el servidor de Argentina");}

        public void cerraConexion() {
System.out.println("Cerrando conexion con el servidor de Argentina");}

        public void conectate() {
                System.out.println("Conectando al servidor de Argentina");}

        public void guardaLog() {
        System.out.println("Guardar Log de Argentina");     }

        public void prendete() {
                System.out.println("Prendiendo el servidor de Argentina");}

        public void verificaConexion() {
                System.out.println("Comprobando la conexion de Argentina"); }
        }
```

Hasta aquí nada nuevo, sólo clases que implementan una interfaz y le dan inteligencia al método. Comenzaremos con el Command:

```
public interface Command {
    public void execute();
}
```

Y ahora realizamos las operaciones-objetos, es decir, los Command Concretos:

```
public class PrendeServer implements Command {
    private IServer servidor;

    public PrendeServer(IServer servidor){
            this.servidor = servidor;}

    public void execute() {
            servidor.conectate();
            servidor.verificaConexion();
            servidor.prendete();
            servidor.guardaLog();
            servidor.cerraConexion();}
}

public class ResetServer implements Command{
    private IServer servidor;
```

```
        public ResetServer(IServer servidor){
                this.servidor = servidor;}

        public void execute() {
                servidor.conectate();
                servidor.verificaConexion();
                servidor.guardaLog();
                servidor.apagate();
                servidor.prendete();
                servidor.guardaLog();
                servidor.cerraConexion();}
}

public class ApagarServer implements Command{
        private IServer servidor;

        public ApagarServer(IServer servidor){
                this.servidor = servidor;}

        public void execute() {
                servidor.conectate();
                servidor.verificaConexion();
                servidor.guardaLog();
                servidor.apagate();
                servidor.cerraConexion();}
}
```

Ahora realizaremos un invocador, es decir, una clase que simplemente llame al método execute:

```
public class Invoker {
        private Command command;

        public Invoker(Command command){
                this.command = command;}

        public void run(){
                command.execute();}
}
```

Veamos como funciona el ejemplo:

```
        public static void main(String[] args) {
                IServer server = new ArgentinaServer();

                Command command = new PrendeServer(server);

                Invoker serverAdmin = new Invoker(command);
                serverAdmin.run();
        }
```

La salida por consola es:

Conectando al servidor de Argentina

Comprobando la conexion de Argentina

Prendiendo el servidor de Argentina

Guardar Log de Argentina

Cerrando conexion con el servidor de Argentina

4.3.12. Cuando utilizarlo

Lo que permite el patrón Command es desacoplar al objeto que invoca a una operación de aquél que tiene el conocimiento necesario para realizarla. Esto nos otorga muchísima flexibilidad: podemos hacer, por ejemplo, que una aplicación ejecute tanto un elemento de menú como un botón para hacer una determinada acción. Además, podemos cambiar dinámicamente los objetos Command.

4.3.13. Patrones relacionados

El patrón Composite suele ser utilizado en conjunto con Command.

4.4. Interpreter Pattern

4.4.1. Introducción y nombre

Interpreter. De Comportamiento.

Utiliza una clase para representar una regla gramática

4.4.2. Intención

Este patrón busca representar un lenguaje mediante reglas gramáticas. Para ello define estas reglas gramáticas y como interpretarlas.

4.4.3. También conocido como

Intérprete.

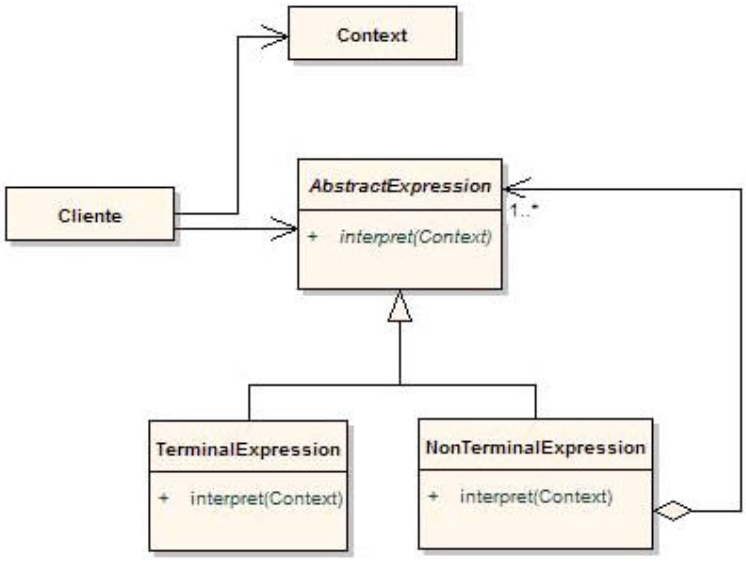
4.4.4. Motivación

Si un tipo particular de problema se presenta frecuentemente, puede ser provechoso expresar los diferentes casos del problema como sentencias de un lenguaje simple. Se puede, entonces, construir un intérprete que resuelva el problema interpretando dichas sentencias.

4.4.5. Solución

Este patrón se debe utilizar cuando hay un lenguaje que interpretar y se puede interpretar sus palabras como árboles sintácticos abstractos. Para ello, la gramática debe ser simple.

4.4.6. Diagrama UML



4.4.7. Participantes

- AbstractExpression: declara una interfaz para la ejecución de una operación.
- TerminalExpression: implementa una operación asociada con los símbolos terminales de la gramática
- NonterminalExpression: implementa una operación de interpretación asociada con los símbolos no terminales de la gramática.
- Context: contiene información global para el interprete.
- Client: construye un árbol sintáctico abstracto que representa una sentencia particular en el lenguaje que la gramática define.

4.4.8. Colaboraciones

El cliente construye una expresión

4.4.9. Consecuencias

Se puede destacar las siguientes ventajas:

- Facilidad para cambiar la gramática, mediante herencia, dado que las diferentes reglas se representan con objetos.
- Facilidad para implementar la gramática, dado que las implementaciones de las clases nodo del árbol sintáctico son similares, pudiendo usarse para ello generadores automáticos de código.
- Facilidad para introducir nuevas formas de "interpretar" las expresiones en la gramática.

Y las siguientes desventajas:

- Limitación en el tipo de gramática: si no es simple, es casi imposible implementarlo.
- No es conveniente utilizarlo si la eficiencia es un punto clave.

4.4.10. Implementación

Para llevar a cabo su implementación se debe tener en cuenta que si es frecuente la relación de nuevos intérpretes o de nuevas formas de "interpretar" es mejor tener un Visitante o patrón Visitor, donde poner la operación a realizar fuera de la estructura de árbol.

El "Cliente" construye (o recibe) una estructura de instancias de "Expresiones terminales" y "Expresiones no terminales", ensambladas para formar un árbol sintáctico que representa la sentencia que se quiere interpretar, mientras que el "Contexto" contiene información global para el intérprete.

Quizas lo más complicado en este patrón es definir las operaciones de interpretación. Es ideal tener operaciones sencillas y claras.

El problema de encontrar palabras que encajen en el patrón se puede resolver definiendo un lenguaje para ello, por ejemplo mediante "Expresiones Regulares". Esto nos permite ya aplicar un intérprete a dicho lenguaje para resolver el problema. Este lenguaje es muy utilizado en compiladores implementados con lenguajes orientados a objetos.

4.4.11. Código de muestra

Luego de observar el siguiente ejemplo, el alumno entenderá porque este patrón sólo funciona con lenguajes con reglas sencillas. Veamos un ejemplo donde se utiliza el interpreter para interpretar los números romanos mediante ciertas reglas matemáticas y convertirlo en un número de escala decimal.

Primero creamos el contexto:

```
public class Context {
    String input;
    int output;

    public Context(String s){
           input = s;
    }
}
```

Ahora vamos a crear las reglas para interpretar los números romanos. No son algoritmos sencillos, son reglas matemáticas para que funcione la conversión.

```
public abstract class Expression {
    public void interpret(Context context) {
      if (context.input.startsWith(nine()))  {
        context.output += (9 * multiplier());
        context.input = context.input.substring(2);    }

      else if (context.input.startsWith(four()))  {
       context.output += (4 * multiplier());
           context.input =context.input.substring(2); }

      else if (context.input.startsWith(five()))  {
       context.output += (5 * multiplier());
           context.input = context.input.substring(1);  }

      while (context.input.startsWith(one()))   {
       context.output += (1 * multiplier());
           context.input = context.input.substring(1); }  }

    public abstract String one();
    public abstract String four();
    public abstract String five();
    public abstract String nine();
    public abstract int multiplier();

}

public class OneExpression extends Expression {
     public  String one() { return "I"; }
        public  String four(){ return "IV"; }
        public  String five(){ return "V"; }
        public  String nine(){ return "IX"; }
        public  int multiplier() { return 1; }
```

```
}

public class TenExpression extends Expression {

    public  String one() { return "X"; }
    public  String four(){ return "XL"; }
    public  String five(){ return "L"; }
    public  String nine(){ return "XC"; }
    public  int multiplier() { return 10;}
    }
public class HundredExpression extends Expression {

     public String five() {return "D";    }
     public String four() {return "CD";}
     public String nine() {return "CM";}
     public String one() {  return "C";    }
     public int multiplier() {return 100;}
}

public class ThousandExpression extends Expression {

     public String five() {return " ";    }
     public String four() {return " ";    }
     public String nine() {return " ";    }
     public String one() {  return "M";    }
     public int multiplier() {return 0;   }
}
```

El funcionamiento es el siguiente:

```
public static void main(String[] args) {
            String romano = "LXI";
            Context context = new Context(romano);
            // Construimos el árbol
            ArrayList<Expression> tree = new ArrayList<Expression>();
            tree.add(new ThousandExpression());
            tree.add(new HundredExpression());
            tree.add(new TenExpression());
            tree.add(new OneExpression());
            // Lo interpretamos
            for (Expression exp : tree) {
                        exp.interpret(context);}
            System.out.println(context.output);
     }
```

El resultado por consola es: 61

4.4.12. Cuando utilizarlo

La aplicación de este patrón es quizas la más compleja de todo los patrones que veremos. Difícilmente el desarrollador utilice este patrón en algún momento de su vida, lo que no quita que no sea un patrón utilizado. La situación ideal que se debe considerar para aplicar este patrón es que exista un lenguaje sencillo que pueda interpretarse con palabras. El ejemplo más claro es JAVA: este lenguaje permite escribir en archivos .java entendibles por humanos y luego este archivo es

compilado e interpretado para que pueda ejecutar sentencias entendibles por una máquina.

4.4.13. Patrones relacionados

El árbol sintáctico abstracto suele ser una instancia de Composite.

Se puede usar un Iterator para recorrer el árbol.

Flyweigth muestra como compartir símbolos terminales en el árbol.

4.5. Iterator Pattern

4.5.1. Introducción y nombre

Iterator. De comportamiento. Provee un mecanismo estándar para acceder secuencialmente a los elementos de una colección.

4.5.2. Intención

Define un interface que declara métodos para acceder secuencialmente a los objetos de una colección. Una clase accede a una colección solamente a través de un interface independiente de la clase que implementa el interface.

4.5.3. También conocido como

Iterador, Cursor.

4.5.4. Motivación

La motivación de este patrón reside en la gran diversidad de colecciones y algoritmos que existe hoy en día para recorrer una colección. Lo que se busca es acceder a los contenidos de los objetos incluidos sin exponer su estructura.

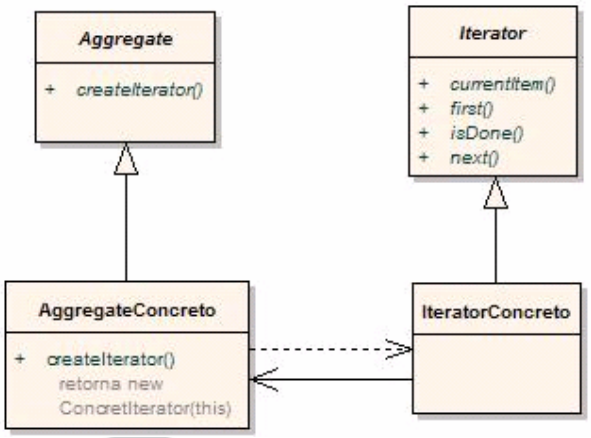
Podemos decir que este patrón nace para poder soportar diversas formas de recorrer objetos y para ofrecer una interfaz uniforme para recorrer distintos tipos de estructuras de agregación.

4.5.5. Solución

El patrón Iterator se puede utilizar en los siguientes casos:

- Una clase necesita acceder al contenido de una colección sin llegar a ser dependiente de la clase que es utilizada para implementar la colección, es decir sin tener que exponer su representación interna.
- Una clase necesita un modo uniforme de acceder al contenido de varias colecciones.
- Cuando se necesita soportar múltiples recorridos de una colección.

4.5.6. Diagrama UML



4.5.7. Participantes

- Agregado/Aggregate: define una interfaz para crear un objeto iterator.
- Iterator: define la interfaz para acceder y recorrer los elementos de un agregado.
- IteradorConcreto: implementa la interfaz del iterador y guarda la posición actual del recorrido en cada momento.
- AgregadoConcreto: implementa la interfaz de creación de iteradores devolviendo una instancia del iterador concreto apropiado.

4.5.8. Colaboraciones

Cliente: solicita recorrer una colección y lo hace siguiendo los métodos otorgados por la interfaz Iterator.

4.5.9. Consecuencias

- Es posible acceder a una colección de objetos sin conocer el código de los objetos.
- Utilizando varios objetos Iterator, es simple tener y manejar varios recorridos al mismo tiempo.
- Es posible para una clase Colecction proporcionar diferentes tipos de objetos Iterator que recorran la colección en diferentes modos. Por ejemplo, una clase Colecction que mantiene una asociación entre la clave de los objetos y el valor de los objetos podría tener diferentes métodos para crear Iterators que recorran sólo la clave de los objetos y crear Iterators que recorran sólo el valor de los objetos.
- Las clases Iterator simplifican el código de las colecciones, ya que el código de los recorridos se encuentra en los Iterators y no en las colecciones.

4.5.10. Implementación

La primer interface que nació con el patrón obligaba a implementar los siguientes métodos: was, first, next, isDone y currentItem. Pero Java le asignó los siguientes métodos: next, hasNext y remove. Este último método saca un objeto de la colección.

Dada la popularidad que tuvo este patrón con Java, estos últimos métodos quedaron como estándar para la interface Iterator. Se deben implementar de la siguiente manera:

boolean hasNext() debe devolver si hay un próximo elemento en la colección.

Object next() devuelve el objeto de esa iteración.

4.5.11. Código de muestra

Vamos a realizar un ejemplo muy sencillo: una división o sucursal que contiene empleados. Para ello internamente implemento un Array pero el cliente no tiene porque saberlo. Veamos la clase empleado:

```
public class Empleado {
     private String nombre;
     private String division;

     public Empleado(String n, String d) {
            nombre = n;
            division = d;
     }

     public String getName() {
            return nombre;
     }

     public void print() {
System.out.println("Nombre: " + nombre + " Division: " + division); }}
```

La clase contenedora será:

```
public class Division {
     private Empleado[] empleados = new Empleado[100];
     private int numero = 0;
     private String nombreDivision;

     public Division(String n) {
            nombreDivision = n;
     }

     public String getName() {
            return nombreDivision;
     }

     public void add(String nombre) {
            Empleado e = new Empleado(nombre, nombreDivision);
            empleados[numero++] = e;
     }

     public DivisionIterator iterator() {
            return new DivisionIterator(empleados);
     }
}
```

El método iterator devuelve un objeto DivisionIterator. Veamos como funciona:

```
public class DivisionIterator implements Iterator<Empleado> {
     private Empleado[] empleado;
     private int location = 0;

     public DivisionIterator(Empleado[] e) {
            empleado = e;
     }
```

```
        public Empleado next() {
                return empleado[location++];
        }

        public boolean hasNext() {
        if (location < empleado.length && empleado[location] != null) {
                        return true;
                } else {
                        return false;
                }
        }

        public void remove() {
        }
}
```

Si ha recorrido alguna colección mediante un objeto Iterator estará familiarizado con el siguiente código:

```
public static void main(String[] args) {

            Division d = new Division("Mi Sucursal");
            d.add("Empleado 1");
            d.add("Empleado 2");

    Iterator<Empleado> iter = d.iterator();
    while (iter.hasNext()) {
            Empleado e = (Empleado) iter.next();
            e.print();}
    }

La salida por consola es:
Nombre: Empleado 1 Division: Mi Sucursal
Nombre: Empleado 2 Division: Mi Sucursal
```

4.5.12. Cuando utilizarlo

Este patrón debe ser utilizado cuando se requiera una forma estándar de recorrer una colección, es decir, cuando no es necesario que un cliente sepa la estructura interna de una clase. Un cliente no siempre necesita saber si debe recorrer un List o un Set o un Queue y, menos que menos, que clase concreta esta recorriendo.

El ejemplo más importante de este patrón esta en el Java Framework Collection: las colecciones en el paquete java.util siguen el patrón Iterator. La causa de porque Java utiliza este patrón es muy simple. En la versión 1.6 Java posee alrededor de 50 colecciones. Y cada una de ellas es un caso de estudio: cada una funciona de manera distinta a la otra, con algoritmos muy distintos, por ejemplo: hay trees, binary trees, arrays, ring buffers, hashes, hash maps, array lists, sets, etc. Si nosotros como clientes de Java tendríamos que aprender como se debe recorrer una colección en Java, sería realmente muy molesto. Pero si todas las colecciones

se recorren de la misma forma estándar, y además, cada colección sabe cual es la mejor manera de recorrerla, entonces sería un gran avance que, de hecho, lo es.

4.5.13. Patrones relacionados

Adapter: el patrón Iterator es una forma especializada del patrón Adapter para acceder secuencialmente a los contenidos de una colección de objetos.

Factory Method: algunas clases Colecction podrían usar el patrón Factory Method para determinar que tipo de Iterator instanciar.

4.6. Mediator Pattern

4.6.1. Introducción y nombre

Mediator. De Comportamiento. Define un objeto que hace de procesador central.

4.6.2. Intención

Un Mediator es un patrón de diseño que coordina las relaciones entre sus asociados o participantes. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones. Todos los objetos se comunican con un mediador y es éste quién realiza la comunicación con el resto.

4.6.3. También conocido como

Mediador, Intermediario.

4.6.4. Motivación

Cuando muchos objetos interactúan con otros objetos, se puede formar una estructura muy compleja, con muchas conexiones entre distintos objetos. En un caso extremo cada objeto puede conocer a todos los demás objetos. Para evitar esto, el patrón Mediator, encapsula el comportamiento de todo un conjunto de objetos en un solo objeto.

4.6.5. Solución

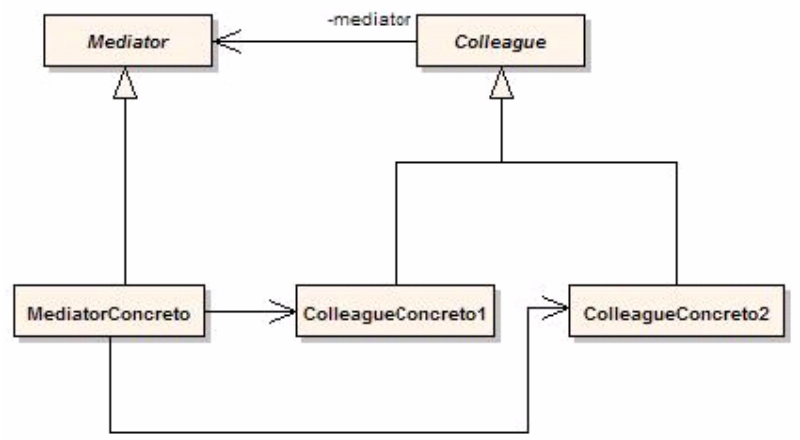
Usar el patrón Mediator cuando:

- Un conjunto grande de objetos se comunica de una forma bien definida, pero compleja.
- Reutilizar un objeto se hace difícil por que se relaciona con muchos objetos.

- Las clases son difíciles de reutilizar porque su función básica esta entrelazada con relaciones de dependencia.

4.6.6. Diagrama UML

Estructura en un diagramas de clases.



4.6.7. Participantes

Responsabilidad de cada clase participante.

- Mediator: define una interface para comunicarse con los objetos colegas.
- MediatorConcreto: implementa la interface y define como los colegas se comunican entre ellos. Además los conoce y mantiene, con lo cual hace de procesador central de todos ellos.
- Colleague: define el comportamiento que debe implementar cada colega para poder comunicarse el mediador de una manera estandarizada para todos.
- ColleagueConcreto: cada colega conoce su mediador, y lo usa para comunicarse con otros colegas.

4.6.8. Colaboraciones

Los colegas envían y reciben requerimientos de un objeto mediador. El mediador gestiona cada mensaje y se lo comunica a otro colega si fuese necesario.

4.6.9. Consecuencias

- Desacopla a los colegas: el patrón Mediator promueve bajar el acoplamiento entre colegas. Se puede variar y reusar colegas y mediadores independiéntemente.

- Simplifica la comunicación entre objetos: los objetos que se comunican de la forma "muchos a muchos" puede ser remplazada por una forma "uno a muchos" que es menos compleja y más elegante. Además esta forma de comunicación es más fácil de entender. Es decir, un objeto no necesita conocer a todos los objetos, tan sólo a un mediador.

- Clarifica como los objetos se relacionan en un sistema.

- Centraliza el control: el mediador es el que se encarga de comunicar a los colegas, este puede ser muy complejo, difícil de entender y modificar. Para que quién conoce el framework Struts, es muy similar al concepto del archivo struts-config.xml: centraliza el funcionamiento de la aplicación, aunque si llega a ser una aplicación muy compleja el archivo se vuelve un tanto complicado de entender y seguir.

4.6.10. Implementación

Sabemos que el patrón Mediator introduce un objeto para mediar la comunicación entre "colegas". Algunas veces el objeto Mediador implementa operaciones simplemente para enviarlas o otros objetos; otras veces pasa una referencia a él mismo y por consiguiente utiliza la verdadera delegación.

Entre los colegas puede existir dos tipos de dependencias:

1. Un tipo de dependencia requiere un objeto para conseguir la aprobación de otros objetos antes de hacer tipos específicos de cambios de estado.
2. El otro tipo de dependencia requiere un objeto para notificar a otros objetos después de que este ha hecho un tipo específico de cambios de estado.

Ambos tipos de dependencias son manejadas de un modo similar. Las instancias de Colega1, Colega2, ... están asociadas con un objeto mediator. Cuando ellos quieren conseguir la aprobación anterior para un cambio de estado, llaman a un

método del objeto Mediator. El método del objeto Mediator realiza cuidadoso el resto.

Pero hay que tener en cuenta lo siguiente con respecto al mediador: Poner toda la dependencia de la lógica para un conjunto de objetos relacionados en un lugar puede hacer incomprensible la dependencia lógica fácilmente. Si la clase Mediator llega a ser demasiado grande, entonces dividirlo en piezas más pequeñas puede hacerlo más comprensible.

4.6.11. Código de muestra

Nuestro ejemplo será un chat: donde habrá usuarios que se comunicaran entre sí en un salón de chat. Para ellos se define una interface llamada Chateable que todos los objetos que quieran participar de un chat deberán implementar.

```
public interface Chateable {
    public void recibe(String de, String msg);
    public void envia(String a, String msg);

}
```

La clase Ususario representa un usuario que quiera chatear.

```
public class Usuario implements Chateable {
    private String nombre;
    private SalonDeChat salon;

    public Usuario(SalonDeChat salonDeChat){
            salon = salonDeChat;
    }
    public void recibe(String de, String msg) {
            String s = "el usuario " + de + " te dice: "+msg;
            System.out.println(nombre + ": " + s);
    }
    public void envia(String a, String msg){
            salon.envia(nombre,a,msg);
    }
    public String getNombre() {
            return nombre;
    }
    public void setNombre(String nombre) {
            this.nombre = nombre;
    }
    public SalonDeChat getSalon() {
            return salon;
    }
    public void setSalon(SalonDeChat salon) {
            this.salon = salon;
    }
```

```
}

public interface IChat {
    public abstract void registra(Usuario participante);
    public abstract void envia(String from, String to, String message);

}

public class SalonDeChat implements IChat {

private HashMap<String, Usuario> participantes = new HashMap<String,
Usuario>();

    public void registra(Usuario user) {
           participantes.put(user.getNombre(), user);
    }

    public void envia(String de, String a, String msg) {
if (participantes.containsKey(de) && participantes.containsKey(a)) {
                  Usuario u = participantes.get(a);
                  u.recibe(de, msg);
           }else {
                  System.out.println("Usuario inexistente");
           }}
}

public static void main(String[] args) {
           SalonDeChat s = new SalonDeChat();
           Usuario u = new Usuario(s);
           u.setNombre("Juan");

           Usuario u1 = new Usuario(s);
           u1.setNombre("Pepe");

           Usuario u2 = new Usuario(s);
           u2.setNombre("Pedro");

           s.registra (u);
           s.registra (u1);
           s.registra (u2);

           u.envia("Pepe", "Hola como andas?");
           u1.envia("Juan", "Todo ok, vos?");
           u2.envia("Martin", "Martin estas?");
    }

El resultado por consola es:
Pepe: el usuario Juan te dice: Hola como andas?
Juan: el usuario Pepe te dice: Todo ok, vos?
Usuario inexistente
```

4.6.12. Cuando utilizarlo

Un ejemplo real que se puede comparar con este patrón es un framework que se llama Struts. Struts posee un clase que hace de Mediadora que se llama ActionServlet. Esta clase lee un archivo de configuración (el struts-config.xml) para ayudarse con la mediación, pero lo importante es que se encarga de comunicar el flujo de información de todos los componentes web de una aplicación. Si no

utilizamos Struts, entonces cada página debe saber hacia donde debe dirigir el flujo y el mantenimiento de dicha aplicación puede resultar complicado, especialmente si la aplicación es muy grande.

En cambio, si todas las páginas se comunican con un mediador, entonces la aplicación es mucho más robusta: con cambiar un atributo del mediador todo sigue funcionando igual.

Como conclusión podemos afirmar que este patrón debe ser utilizado en casos donde convenga utilizar un procesador central, en vez de que cada objeto tenga que conocer la implementación de otro. Imaginemos un aeropuerto: que pasaría si no tuviese una torre de control y todos los aviones que deban aterrizar/despegar se tienen que poner todos de acuerdo para hacerlo. Además cada avión debe conocer detalles de otros aviones (velocidad de despegue, nafta que le queda a cada uno que quiera aterrizar, etc).

Para evitar esto se utiliza un torre de control que sincroniza el funcionamiento de un aeropuerto. Esta torre de control se puede ver como un mediador entre aviones.

4.6.13. Patrones relacionados

Patrones con los que potencialmente puede interactuar:

Observer: los colegas pueden comunicarse entre ellos mediante un patrón observador.

Facade: es un concepto similar al mediador, pero este último es un poco más completo y la comunicación es bidireccional.

Singleton: el mediador puede ser Singleton.

4.7. Memento Pattern

4.7.1. Introducción y nombre

Memento. De Comportamiento. Permite capturar y exportar el estado interno de un objeto para que luego se peuda restaurar, sin romper la encapsulación.

4.7.2. Intención

Este patrón tiene como finalidad almacenar el estado de un objeto (o del sistema completo) en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece el objeto (pero sin romper la encapsulación), de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior.

4.7.3. También conocido como

Token.

4.7.4. Motivación

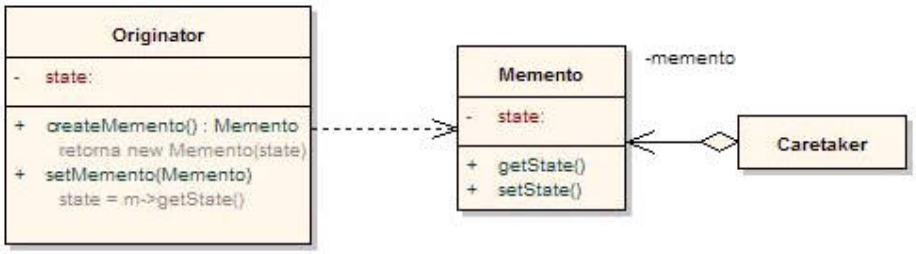
Muchas veces es necesario guardar el estado interno de un objeto. Esto debido a que tiempo después, se necesita restaurar el estado del objeto, al que previamente se ha guardado. Hoy en día, muchos aplicativos permiten el "deshacer" y "rehacer" de manera muy sencilla. Para ciertos aplicativos es casi una obligación tener estas funciones y sería impensado el hecho que no las posean. Sin embargo, cuando queremos llevar esto a código puede resultar complejo de implementar. Este patrón intenta mostrar una solución a este problema.

4.7.5. Solución

El patrón Memento se usa cuando:

- Se necesite restaurar el sistema desde estados pasados.
- Se quiera facilitar el hacer y deshacer de determinadas operaciones, para lo que habrá que guardar los estados anteriores de los objetos sobre los que se opere (o bien recordar los cambios de forma incremental).

4.7.6. Diagrama UML



4.7.7. Participantes

Caretaker

- Es responsable por mantener a salvo a Memento.
- No opera o examina el contenido de Memento.

Memento

- Almacena el estado interno de un objeto Originator. El Memento puede almacenar todo o parte del estado interno de Originator.
- Tiene dos interfaces. Una para Caretaker, que le permite manipular el Memento únicamente para pasarlo a otros objetos. La otra interfaz sirve para que Originator pueda almacenar/restaurar su estado interno, sólo Originator puede acceder a esta interfaz.

Originator

- Originator crea un objeto Memento conteniendo una fotografía de su estado interno.

4.7.8. Colaboraciones

Originator crea un Memento y el mismo almacena su estado interno.

4.7.9. Consecuencias

No es necesario exponer el estado interno como atributos de acceso público, preservando así la encapsulación.

Si el originador tendría que de almacenar y mantener a salvo una o muchas copias de su

estado interno, sus responsabilidades crecerían y sería inmanejable.

El uso frecuente de Mementos para almacenar estados internos de gran tamaño, podría

resultar costoso y perjudicar la performance del sistema.

Caretaker no puede hacer predicciones de tiempo ni de espacio.

4.7.10. Implementación

Si bien la implementación de un Memento no suele variar demasiado, cuando la secuencia de creación y restauración de mementos es conocida, se puede adoptar una estrategia de cambio incremental: en cada nuevo memento sólo se almacena la parte del estado que ha cambiado en lugar del estado completo.

Esta estrategia se aplica cuando memento se utiliza para mantener una lista de deshacer/rehacer.

Otra opción utilizada es no depender de índices en la colecciones y utilizar ciertos métodos no indexados como el .previus() que poseen algunas colecciones.

4.7.11. Código de muestra

Vamos a realizar un ejemplo de este patrón donde se busque salvar el nombre de una persona que puede variar a lo largo del tiempo.

```
public class Memento {
    private String estado;

    public Memento(String estado) {
        this.estado = estado;
    }

    public String getSavedState() {
        return estado;
    }
}

public class Caretaker {
    private ArrayList<Memento> estados = new ArrayList<Memento>();

    public void addMemento(Memento m) {
        estados.add(m);
    }

    public Memento getMemento(int index) {
        return estados.get(index);
    }
}


public class Persona {
    private String nombre;

    public Memento saveToMemento() {
        System.out.println("Originator: Guardando Memento...");
        return new Memento(nombre);
    }

    public void restoreFromMemento(Memento m) {
        nombre = m.getSavedState();
    }

    public String getNombre() {
        return nombre;
    }

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

Vamos a probar este ejemplo:

```
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();

        Persona p = new Persona();
        p.setNombre("Maxi");
        p.setNombre("Juan");
```

```
                caretaker.addMemento(p.saveToMemento());

                p.setNombre("Pedro");

                caretaker.addMemento(p.saveToMemento());

                p.setNombre("Diego");

                Memento m1 = caretaker.getMemento(0);
                Memento m2 = caretaker.getMemento(1);

                System.out.println(m1.getSavedState());
                System.out.println(m2.getSavedState());
        }
```

```
La salida por consola es:
Originator: Guardando Memento...
Originator: Guardando Memento...
Juan
Pedro
```

4.7.12. Cuando utilizarlo

Este patrón debe ser utilizado cuando se necesite salvar el estado de un objeto y tener disponible los distintos estados históricos que se necesiten. Por ello mismo, este patrón es muy intuitivo para darse cuando debe ser utilizado.

Hoy en día una gran variedad de aplicaciones poseen las opciones de "deshacer" y "rehacer". Por ejemplo, las herramientas de Microsoft Office como Word, Power Point, etc. Es imposible pensar que ciertas herramientas no tengan esta opción, como el Photoshop. También IDEs de programación como Eclipse utilizan una opción de historial local. Una solución para este problema es el patrón Memento.

4.7.13. Patrones relacionados

Command: puede usar “Mementos” para guardar el estado de operaciones restaurables.

Iterator: “Mementos” puede iterar con un Iterator para buscar en colecciones los estados específicos.

4.8. Observer Pattern

4.8.1. Introducción y nombre

Observer. De Comportamiento. Permite reaccionar a ciertas clases llamadas observadores sobre un evento determinado.

4.8.2. Intención

Este patrón permite a los objetos captar dinámicamente las dependencias entre objetos, de tal forma que un objeto notificará a los observadores cuando cambia su estado, siendo actualizados automáticamente.

4.8.3. También conocido como

Dependents, Publish-Subscribe, Observador.

4.8.4. Motivación

Este patrón es usado en programación para monitorear el estado de un objeto en un programa. Está relacionado con el principio de invocación implícita.

La motivación principal de este patrón es su utilización como un sistema de detección de eventos en tiempo de ejecución. Es una característica muy interesante en términos del desarrollo de aplicaciones en tiempo real.

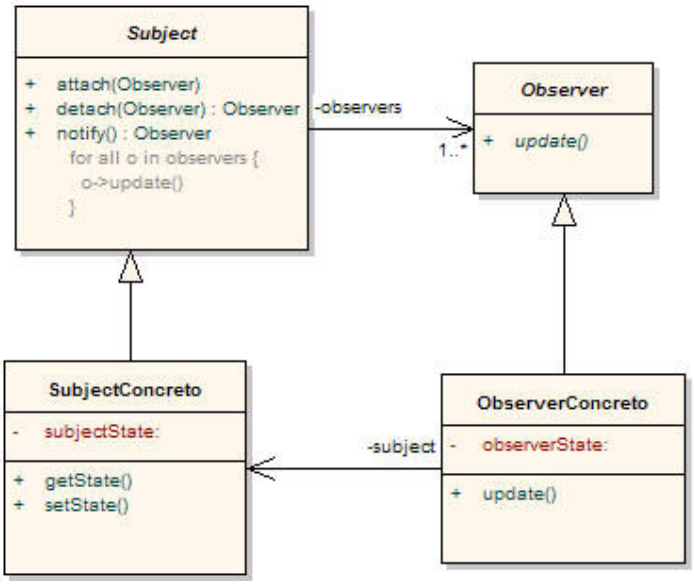
4.8.5. Solución

Este patrón debe ser utilizado cuando:

Un objeto necesita notificar a otros objetos cuando cambia su estado. La idea es encapsular estos aspectos en objetos diferentes permite variarlos y reutilizarlos independientemente.

Cuando existe una relación de dependencia de uno a muchos que puede requerir que un objeto notifique a múltiples objetos que dependen de él cuando cambia su estado.

4.8.6. Diagrama UML



4.8.7. Participantes

Subject: conoce a sus observadores y ofrece la posibilidad de añadir y eliminar observadores.

Observer: define la interfaz que sirve para notificar a los observadores los cambios realizados en el Subject.

SubjectConcreto: almacena el estado que es objeto de interés de los observadores y envía un mensaje a sus observadores cuando su estado cambia.

ObserverConcreto: mantiene una referencia a un SubjectConcreto. Almacena el estado del Subject que le resulta de interés. Implementa la interfaz de actualización de Observer para mantener la consistencia entre los dos estados.

4.8.8. Colaboraciones

El subject posee un método llamado attach() y otro detach() que sirven para agregar o remover observadores en tiempo de ejecución.

El subjectConcreto notifica a sus observadores cada vez que se produce el evento en cuestión. Esto suele realizarlo mediante el recorrido de una Collection.

4.8.9. Consecuencias

- Permite modificar las clases subjects y las observers independientemente.
- Permite añadir nuevos observadores en tiempo de ejecución, sin que esto afecte a ningún otro observador.
- Permite que dos capas de diferentes niveles de abstracción se puedan comunicar entre sí sin romper esa división.
- Permite comunicación broadcast, es decir, un objeto subject envía su notificación a todos los observers sin enviárselo a ningún observer en concreto (el mensaje no tiene un destinatario concreto). Todos los observers reciben el mensaje y deciden si hacerle caso ó ignorarlo.
- La comunicación entre los objetos subject y sus observadores es limitada: el evento siempre significa que se ha producido algún cambio en el estado del objeto y el mensaje no indica el destinatario.

4.8.10. Implementación

Si los observadores pueden observar a varios objetos subject a la vez, es necesario ampliar el servicio update() para permitir conocer a un objeto observer dado cuál de los objetos subject que observa le ha enviado el mensaje de notificación.

Una forma de implementarlo es añadiendo un parámetro al servicio update() que sea el objeto subject que envía la notificación (el remitente). Y añadir una lista de objetos subject observados por el objeto observer en la clase Observer.

Si los objetos observers observan varios eventos de interés que pueden suceder con los objetos subjects, es necesario ampliar el servicio add() y el update() además de la implementación del mapeo subject-observers en la clase abstracta Subject. Una forma de implementarlo consiste en introducir un nuevo parámetro al servicio add() que indique el evento de interés del observer a añadir e introducirlo también como un nuevo parámetro en el servicio update() para que el subject que reciba el mensaje de notificación sepa qué evento ha ocurrido de los que observa.

Cabe destacar que Java tiene una propuesta para el patrón observer:

Posee una Interfaz java.util.Observer:

public interface Observer: una clase puede implementar la interfaz Observer cuando dicha clase quiera ser informada de los cambios que se produzcan en los objetos observados. Tiene un servicio que es el siguiente: void update (Observable o, Object arg)

Este servicio es llamado cuando el objeto observado es modificado.

Java nos ofrece los siguientes servicios:

- void addObserver (Observer o)
- protected void clearChanged()
- int countObservers()
- void deleteObserver (Observer o)
- void deleteObservers()
- boolean hasChanged()
- void notifyObservers()
- void notifyObservers (Object arg)
- protected void setChanged()

Posee una clase llamada java.util.Observable:

```
public Class Observable extends Object
```

Esta clase representa un objeto Subject.

En el código de muestra haremos un ejemplo estándard y otro con la propuesta de Java.

4.8.11. Código de muestra

Vamos a suponer un ejemplo de una Biblioteca, donde cada vez que un lector devuelve un libro se ejecuta el método devuelveLibro(Libro libro) de la clase Biblioteca.

Si el lector devolvió el libro dañado entonces la aplicación avisa a ciertas clases que están interesadas en conocer este evento:

```
public interface ILibroMalEstado {
    public void update();
}
```

La interfaz IlibroMalEstado debe ser implementada por todos los observadores:

```
public class Stock implements ILibroMalEstado{

    public void update() {
           System.out.println("Stock: ");
           System.out.println("Le doy de baja...");
    }
}


public class Administracion implements ILibroMalEstado {

    public void update() {
           System.out.println("Administracion: ");
           System.out.println("Envio una queja formal...");
    }
}
public class Compras implements ILibroMalEstado {

    public void update() {
           System.out.println("Compras: ");
           System.out.println("Solicito nueva cotizacion...");
    }
}
```

Ahora realizaremos la clase que notifica a los observadores:

```
public interface Subject {
```

```
    public void attach(ILibroMalEstado observador);
    public void dettach(ILibroMalEstado observador);
    public void notifyObservers();
}

public class AlarmaLibro implements Subject{
    private static ArrayList<ILibroMalEstado> observadores = new
ArrayList<ILibroMalEstado>();

    public void attach(ILibroMalEstado observador){
           observadores.add(observador); }

    public void dettach(ILibroMalEstado observador){
           observadores.remove(observador);     }

    public  void notifyObservers() {
           for (int i = 0; i < observadores.size(); i++) {
                  ILibroMalEstado observador = observadores.get(i);
                  observador.update();}}
    }
```

Y, por último, la clase que avisa que ocurrió un evento determinado:

```
public class Biblioteca  {

    public void devuelveLibro(Libro libro){
           if (libro.getEstado().equals("MALO")) {
                  AlarmaLibro a = new AlarmaLibro();
                  a.notifyObservers();}}
    }

public class Libro {
    private String tiulo;
    private String estado;
    // Un libro seguramente tendrá más atributos
    // como autor, editorial, etc pero para nuestro
    // ejemplo no son necesarios.

    public String getTiulo() {
           return tiulo;
    }
    public void setTiulo(String tiulo) {
           this.tiulo = tiulo;
    }
    public String getEstado() {
           return estado;
    }
    public void setEstado(String estado) {
           this.estado = estado;
    }

}
```

Veamos como funciona este ejemplo:

```
    public static void main(String[] args) {
           AlarmaLibro a = new AlarmaLibro();
           a.attach(new Compras());
           a.attach(new Administracion());
           a.attach(new Stock());

           Libro libro = new Libro();
           libro.setEstado("MALO");
```

```
            Biblioteca b = new Biblioteca();
            b.devuelveLibro(libro);
            }
```

```
La salida por consola es:
Compras:
Solicito nueva cotizacion...
Administracion:
Envio una queja formal...
Stock:
Le doy de baja...
```

Aquí vemos el mismo ejemplo, pero con el API de Java:

```
public class Administracion implements Observer {

    public void update(Observable arg0, Object arg1) {
            System.out.println(arg1);
            System.out.print("Administracion: ");
            System.out.println("Envio una queja formal...");
    }
}

public class Compras implements Observer {

    public void update(Observable arg0, Object arg1) {
            System.out.println(arg1);
            System.out.print("Compras: ");
            System.out.println("Solicito nueva cotizacion...");
    }
}

public class Stock implements Observer{

    public void update(Observable arg0, Object arg1) {
            System.out.println(arg1);
            System.out.print("Stock: ");
            System.out.println("Le doy de baja...");
    }
}

public class AlarmaLibro extends Observable{

    public void disparaAlarma(Libro libro) {
            setChanged();
            notifyObservers("Rompieron el libro: "+libro.getTiulo());
    }
}

public class Biblioteca{

    public void devuelveLibro(Libro libro){
            if (libro.getEstado().equals("MALO")) {
                    AlarmaLibro a = new AlarmaLibro();
                    a.addObserver(new Compras());
                    a.addObserver(new Administracion());
                    a.addObserver(new Stock());
                    a.disparaAlarma(libro);
            }}
    }
```

```
public static void main(String[] args) {
        Libro libro = new Libro();
        libro.setTiulo("Windows es estable");
        libro.setEstado("MALO");

        Biblioteca b = new Biblioteca();
        b.devuelveLibro(libro);
}
```

```
La salida por consola es:
Rompieron el libro: Windows es estable
Stock: Le doy de baja...
Rompieron el libro: Windows es estable
Administracion: Envio una queja formal...
Rompieron el libro: Windows es estable
Compras: Solicito nueva cotizacion...
```

4.8.12. Cuando utilizarlo

Este patrón tiene un uso muy concreto: varios objetos necesitan ser notificados de un evento y cada uno de ellos deciden como reaccionar cuando esta evento se produzca.

Un caso típico es la Bolsa de Comercio, donde se trabaja con las acciones de las empresas. Imaginemos que muchas empresas estan monitoreando las acciones una empresa X. Posiblemente si estas acciones bajan, algunas personas esten interesadas en vender acciones, otras en comprar, otras quizas no hagan nada y la empresa X quizas tome alguna decisión por su cuenta. Todos reaccionan distinto ante el mismo evento. Esta es la idea de este patrón y son estos casos donde debe ser utilizado.

4.8.13. Patrones relacionados

Se pueden encapsular semánticas complejas entre subjects y observers mediante el patrón Mediator.

Dicha encapsulación podría ser única y globalmente accesible si se usa el patrón Singleton.

Singleton: el Subject suele ser un singleton.

4.9. State Pattern

4.9.1. Introducción y nombre

State. De Comportamiento. Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

4.9.2. Intención

Busca que un objeto pueda reaccionar según su estado interno. Si bien muchas veces esto se puede solucionar con un boolean o utilizando constantes, esto suele terminar con una gran cantidad de if-else, código ilegible y dificultad en el mantenimiento. La intención del State es desacoplar el estado de la clase en cuestión.

4.9.3. También conocido como

Objects for State, Patrón de estados.

4.9.4. Motivación

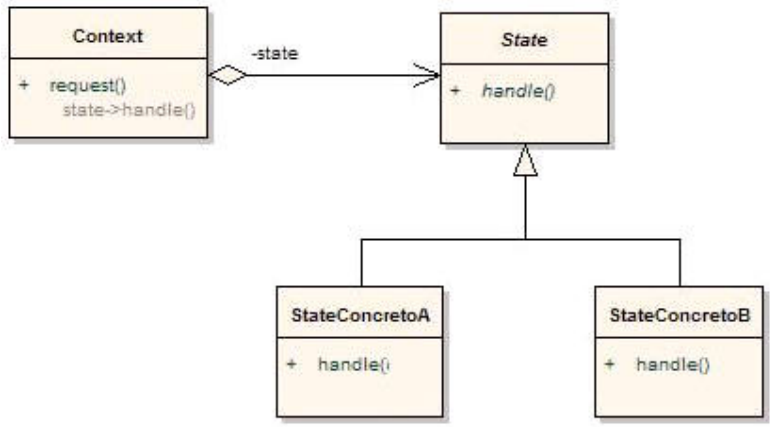
En determinadas ocasiones se requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra. Esto resulta complicado de manejar, sobretodo cuando se debe tener en cuenta el cambio de comportamientos y estados de dicho objeto, todos dentro del mismo bloque de código. El patrón State propone una solución a esta complicación, creando un objeto por cada estado posible.

4.9.5. Solución

Este patrón debe ser utilizado cuando:

- El comportamiento de un objeto depende de un estado, y debe cambiar en tiempo de ejecución según el comportamiento del estado.
- Cuando las operaciones tienen largas sentencias con múltiples ramas que depende del estado del objeto.

4.9.6. Diagrama UML



4.9.7. Participantes

Context: mantiene una instancia con el estado actual

State: define interfaz para el comportamiento asociado a un determinado estado del Contexto.

StateConcreto: cada subclase implementa el comportamiento asociado con un estado del contexto.

4.9.8. Colaboraciones

El Context delega el estado específico al objeto StateConcreto actual. Un objeto Context puede pasarse a sí mismo como parámetro hacia un objeto State. De esta manera la clase State puede acceder al contexto si fuese necesario. Context es la

interfaz principal para el cliente. El cliente puede configurar un contexto con los objetos State. Una vez hecho esto estos clientes no tendrán que tratar con los objetos State directamente. Tanto el objeto Context como los objetos de StateConcreto pueden decidir el cambio de estado.

4.9.9. Consecuencias

- Se localizan fácilmente las responsabilidades de los estados concretos, dado que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior. Esta facilidad la brinda el hecho que los diferentes estados están representados por un único atributo (state) y no envueltos en diferentes variables y grandes condicionales.
- Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables, mientras que aquí al estar representado cada estado.
- Facilita la ampliación de estados mediante una simple herencia, sin afectar al Context.
- Permite a un objeto cambiar de estado en tiempo de ejecución.
- Los estados pueden reutilizarse: varios Context pueden utilizar los mismos estados.
- Se incrementa el número de subclases.

4.9.10. Implementación

La clase Context envía mensajes a los estados concretos dentro de su código para brindarle a éstos la responsabilidad que debe cumplir el objeto Context. Así el objeto Context va cambiando las responsabilidades según el estado en que se encuentra. Para llevar a cabo esto se puede utilizar dos tácticas: que el State sea una interfaz o una clase abstracta. Para resolver este problema, siempre se debe intentar utilizar una interfaz, exceptuando aquellos casos donde se necesite repetir un comportamiento en los estados concretos: para ello lo ideal es utilizar una clase abstracta (state) con los métodos repetitivos en código, de manera que los estados

concretos lo hereden. En este caso, el resto de los métodos no repetitivos deberían ser métodos abstractos.

Un tipo muy importante a tener en cuenta: el patrón no indica exactamente dónde definir las transiciones de un estado a otro. Existen dos formas de solucionar esto: definiendo estas transiciones dentro de la clase contexto, la otra es definiendo estas transiciones en las subclaes de State. Es más conveniente utilizar la primer solución cuando el criterio a aplicar es fijo, es decir, no se modificará. En cambio la segunda resulta conveniente cuando este criterio es dinámico, el inconveniente aquí se presenta en la dependencia de código entre las subclases.

4.9.11. Código de muestra

Veamos un código muy sencillo de entender:

```
public interface SaludState {
     public String comoTeSentis();
     }


public class DolorDeCabeza implements SaludState {

     public String comoTeSentis() {
            return "Todo mal: me duele la cabeza";
     }}

public class DolorDePanza implements SaludState {

     public String comoTeSentis() {
            return "Todo mal: me duele la panza";
     }}

public class Saludable implements SaludState {

     public String comoTeSentis() {
            return "Pipi Cucu!!";
     }
}


public class Persona {
     private String nombre;
     private SaludState salud;

     public Persona(){
            salud = new Saludable();}

     public void estoyBien(){
            salud = new Saludable();
     }
     public void dolorDeCabeza(){
            salud = new DolorDeCabeza();
     }
```

```
        public void dolorDePanza(){
                salud = new DolorDePanza();
        }

        public String comoTeSentis(){
                return salud.comoTeSentis();
        }

        public String getNombre() {
                return nombre;
        }
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }
        public SaludState getSalud() {
                return salud;
        }
        public void setSalud(SaludState salud) {
                this.salud = salud;
        }
}

        public static void main(String[] args) {
                Persona p = new Persona();
                p.setNombre("Juan");
                System.out.println(p.comoTeSentis());

                p.dolorDeCabeza();
                System.out.println(p.comoTeSentis());}
```

```
La salida por consola es:
Pipi Cucu!!
Todo mal: me duele la cabeza
```

4.9.12. Cuando utilizarlo

Este patrón se utiliza cuando un determinado objeto tiene diferentes estados y también distintas responsabilidades según el estado en que se encuentre en determinado instante. También puede utilizarse para simplificar casos en los que se tiene un complicado y extenso código de decisión que depende del estado del objeto.

Imaginemos que vamos a un banco y cuando llegamos nos colocamos en la fila de mostrador: si la misma esta abierta, seguiremos en la fila. En cambio, si esta cerrada nos colocaremos en otra fila o tomaremos alguna decisión acorde. Por otro lado, si vemos un cartel que dice "enseguida vuelvo" quizás tenemos que contemplar el tiempo disponible que tenemos. Es decir, para nosotros, el comportamiento de un banco cambia radicalmente según el estado en el que se encuentre. Para estas ocasiones, es ideal el uso de un patrón de estados.

4.9.13. Patrones relacionados

El patrón State puede utilizar el patrón Singleton cuando requiera controlar que una sola instancia de cada estado. Lo puede utilizar cuando se comparten los objetos como Flyweight existiendo una sola instancia de cada estado y esta instancia es compartida con más de un objeto.

4.10. Strategy Pattern

4.10.1. Introducción y nombre

Strategy. De Comportamiento. Convierte algoritmos en clases y los vuelve intercambiables.

4.10.2. Intención

Encapsula algoritmos en clases, permitiendo que éstos sean re-utilizados e intercambiables. En base a un parámetro, que puede ser cualquier objeto, permite a una aplicación decidir en tiempo de ejecución el algoritmo que debe ejecutar.

4.10.3. También conocido como

Policy, Estrategia.

4.10.4. Motivación

La esencia de este patrón es encapsular algoritmos relacionados que son subclases de una superclase común, lo que permite la selección de un algoritmo que varia según el objeto y también le permite la variación en el tiempo. Esto se define en tiempo de ejecución. Este patrón busca desacoplar bifurcaciones inmensas con algoritmos dificultosos según el camino elegido.

4.10.5. Solución

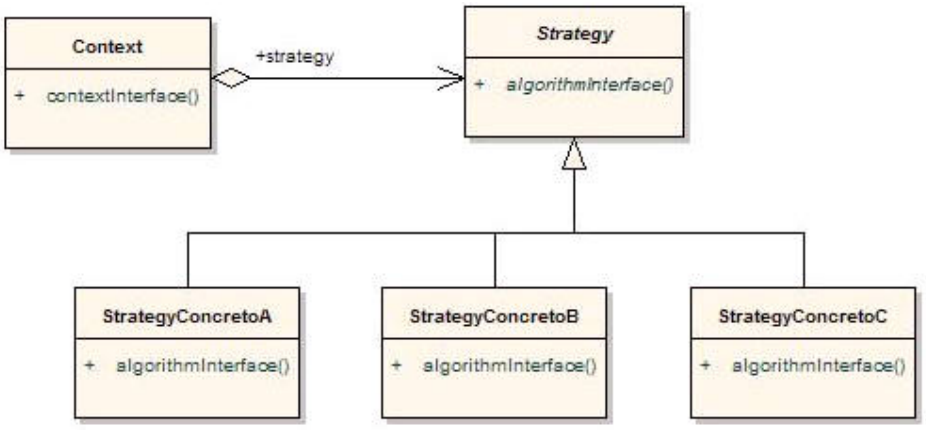
Este patrón debe utilizarse cuando:

- Un programa tiene que proporcionar múltiples variantes de un algoritmo o comportamiento.

- Es posible encapsular las variantes de comportamiento en clases separadas que proporcionan un modo consistente de acceder a los comportamientos.

- Permite cambiar o agregar algoritmos, independientemente de la clase que lo utiliza.

4.10.6. Diagrama UML



4.10.7. Participantes

Strategy: declara una interfaz común a todos los algoritmos soportados.

StrategyConcreto: implementa un algoritmo utilizando la interfaz Strategy. Es la representación de un algoritmo.

Context: mantiene una referencia a Strategy y según las características del contexto, optará por una estrategia determinada..

4.10.8. Colaboraciones

Context / Cliente: solicita un servicio a Strategy y este debe devolver el resultado de un StrategyConcreto.

4.10.9. Consecuencias

Permite que los comportamientos de los Clientes sean determinados dinámicamente sobre un objeto base.

Simplifica los Clientes: les reduce responsabilidad para seleccionar comportamientos o implementaciones de comportamientos alternativos. Esto simplifica el código de los objetos Cliente eliminando las expresiones if y switch.

En algunos casos, esto puede incrementar también la velocidad de los objetos Cliente porque ellos no necesitan perder tiempo seleccionado un comportamiento.

4.10.10. Implementación

Los distintos algoritmos se encapsulan y el cliente trabaja contra el Context. Como hemos dicho, el cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo Context el que elija el más apropiado para cada situación.

Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón Strategy. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución.

4.10.11. Código de muestra

Supongamos un caso donde un instituto educativo tiene una lista de alumnos ordenados por promedio. Dicho instituto suele competir en torneos intercolegiales, en campeonatos nacionales y también en competencias internacionales.

Obviamente la cantidad de gente que participa en cada competencia es distinta: por ejemplo, para los campeonatos locales participan los 3 mejores promedios, pero para las competencias internacionales, sólo se envía al mejor promedio de todos.

```
public class Alumno {
    private String nombre;
    private double promedio;

    public Alumno(String nombre, double promedio){
```

```
                setNombre(nombre);
                setPromedio(promedio);
        }

        public String getNombre() {
                return nombre;
        }
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }
        public double getPromedio() {
                return promedio;
        }
        public void setPromedio(double promedio) {
                this.promedio = promedio;
        }
}

public interface ListadoStrategy {

        public  List getListado(ArrayList lista);
}


public class CompetenciaInternacional implements ListadoStrategy {

        public List getListado(ArrayList lista) {
                ArrayList resultado = new ArrayList();
                resultado.add(lista.get(0));
                return resultado;}
}

public class CompetenciaNacional implements ListadoStrategy{
        public List getListado(ArrayList lista) {
                List resultado = new ArrayList();
                resultado =  lista.subList(0,3);
                return resultado;}
}

public class InterColegial implements ListadoStrategy{

        public List getListado(ArrayList lista) {
                List resultado = new ArrayList();
                resultado = lista.subList(0, 5);
                return resultado;}
}

public class Colegio {
        private ArrayList<Alumno> alumnos;
        private ListadoStrategy lista;

        public Colegio(){
                alumnos = new ArrayList<Alumno>();
                Alumno a1 = new Alumno("Juan", 10);
                Alumno a2 = new Alumno("Sebastian", 9.5);
                Alumno a3 = new Alumno("Mario",9);
                Alumno a4 = new Alumno("Pedro", 8.5);
                Alumno a5 = new Alumno("Matias", 8);
                Alumno a6 = new Alumno("Diego", 7.8);
                alumnos.add(a1);alumnos.add(a2);alumnos.add(a3);
                alumnos.add(a4);alumnos.add(a5);alumnos.add(a6);}

        public ArrayList<Alumno> getAlumnos() {
                return alumnos;       }
```

```
        public void setPersonas(ArrayList<Alumno> alumnos) {
                this.alumnos = alumnos;       }
}


        public static void main(String[] args) {

                Colegio colegio = new Colegio();
                ArrayList<Alumno> alumnos =colegio.getAlumnos();

                ListadoStrategy st = new CompetenciaNacional();
                // se puede evitar que el cliente
                // conozca los strategy concretos

                List rta = st.getListado(alumnos);

                //veamos el resultado del patrón
                System.out.println("Los participantes son:");
                for (int i = 0; i < rta.size(); i++) {
                        Alumno alumno = (Alumno) rta.get(i);
                        System.out.println(alumno.getNombre());
                }}
```

```
La salida por consola es:
Los participantes son:
Juan
Sebastian
Mario
```

4.10.12. Cuando utilizarlo

Este patrón debe ser utilizado cuando un algoritmo es cambiado según un parámetro. Imaginemos una biblioteca de un instituto educativo que presta libros a los alumnos y profesores. Imaginemos que los alumnos pueden asociarse a la biblioteca pagando una mensualidad. Con lo cual un libro puede ser prestado a Alumnos, Socios y Profesores.

Por decisión de la administración, a los socios se les prestará el libro más nuevo, luego aquel que se encuentre en buen estado y, por último, aquel que estuviese en estado regular.

En cambio, si un Alumno pide un libro, ocurre todo lo contrario. Por último, a los profesores sólo se les puede otorgar libros buenos o recién comprados.

Este caso es ideal para el patrón Strategy, ya que dependiendo de un parámetro (el tipo de persona a la que se le presta el libro) puede realizar una búsqueda con distintos algoritmos.

4.10.13. Patrones relacionados

Flyweight: si hay muchos objetos Cliente, los objetos StrategyConcreto puede estar mejor implementados como Flyweights.

El patrón Template Method maneja comportamientos alternativos a través de subclases más que a través de delegación.

4.11. Template Method Pattern

4.11.1. Introducción y nombre

Template Method. De Comportamiento. Define una estructura algorítmica.

4.11.2. 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.

4.11.3. También conocido como

Método plantilla.

4.11.4. 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.

4.11.5. Solución

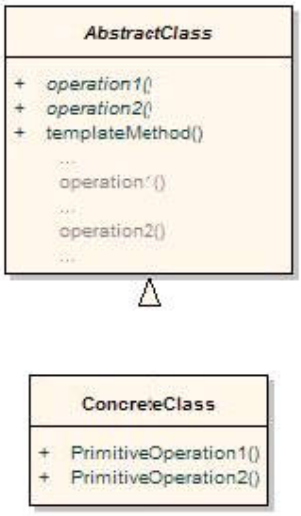
Se debe utilizar este patrón cuando:

- 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.

4.11.6. Diagrama UML



4.11.7. 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.

4.11.8. Colaboraciones

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

4.11.9. Consecuencias

Favorece la reutilización del código.

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

4.11.10. 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.

4.11.11. 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: quizas un empleado tiene un legajo, pero un cliente tiene un numero de cliente, etc.

```
public abstract class Persona {
    private String nombre;
    private String DNI;

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

    protected abstract String getIdentificacion();
    protected abstract String getTipoId();

    public String getNombre() {
           return nombre;
```

```
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getDNI() {
		return DNI;
	}
	public void setDNI(String dni) {
		DNI = dni;
	}
}


public class Socio extends Persona{
	private int numeroDeSocio;

	public Socio(int numeroDeSocio){
		setNumeroDeSocio(numeroDeSocio);
	}
	protected String getIdentificacion() {

		return String.valueOf(numeroDeSocio);
	}
	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){
		setNumeroDeCliente(numeroDeCliente);
	}
	protected String getIdentificacion() {

		return String.valueOf(numeroDeCliente);
	}
	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){
		setLegajo(legajo);
	}
	protected String getIdentificacion() {
		return legajo;
	}
	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 emplado 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
```

4.11.12. Cuando 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.

4.11.13. Patrones relacionados

El Strategy puede usar la composición para cambiar todo el algoritmo, los métodos plantilla usan la herencia para cambiar parte de un algoritmo.

Los Factory Method suelen llamarse desde métodos plantilla.

4.12. Visitor Pattern

4.12.1. Introducción y nombre

Visitor. De Comportamiento. Busca separar un algoritmo de la estructura de un objeto.

4.12.2. Intención

Este patrón representa una operación que se aplica a las instancias de un conjunto de clases. Dicha operación se implementa de forma que no se modifique el código de las clases sobre las que opera.

4.12.3. También conocido como

Visitante.

4.12.4. Motivación

Si un objeto es el responsable de mantener un cierto tipo de información, entonces es

lógico asignarle también la responsabilidad de realizar todas las operaciones necesarias sobre esa información. La operación se define en cada una de las clases que representan los posibles tipos sobre los que se aplica dicha operación, y por medio del polimorfismo y la vinculación dinámica se elige en tiempo de ejecución qué versión de la operación se debe ejecutar. De esta forma se evita un análisis de casos sobre el tipo del parámetro.

4.12.5. Solución

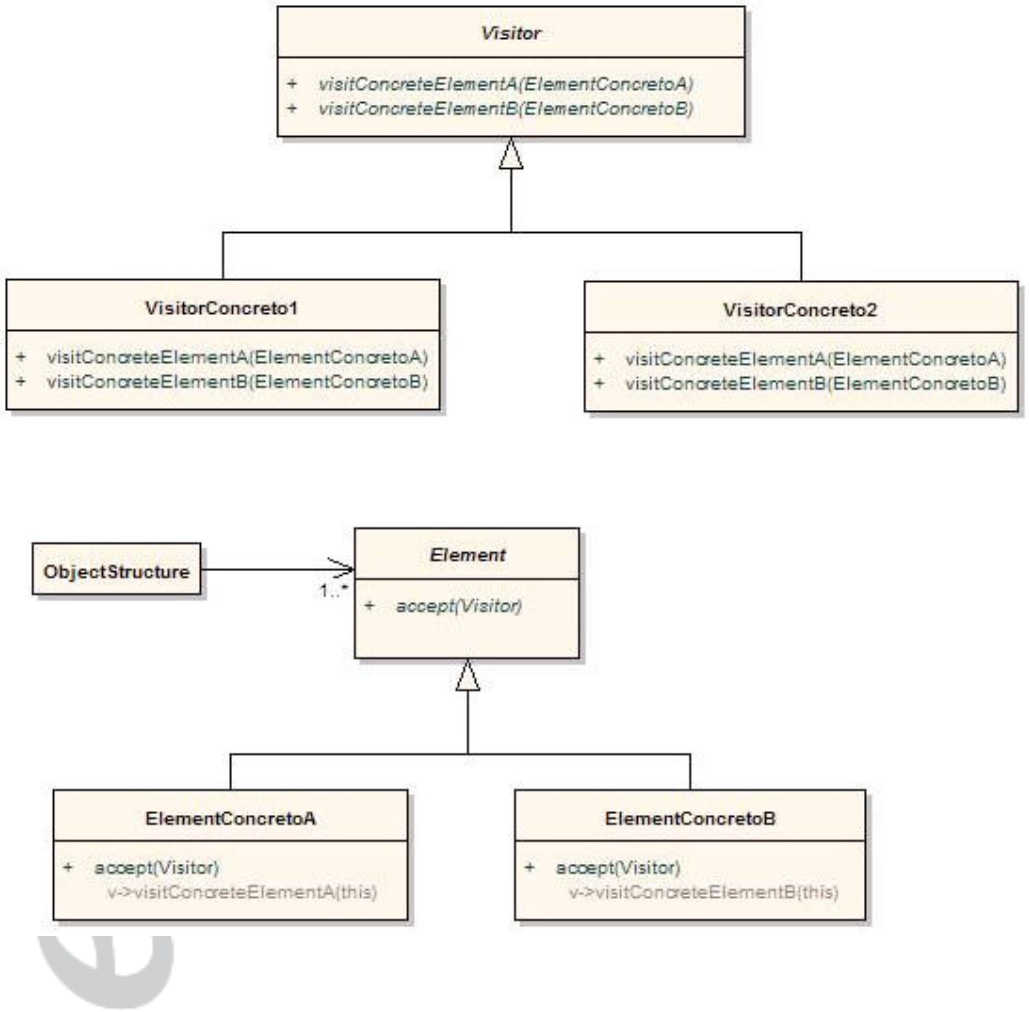
Este patrón debe utilizarse cuando:

- Una estructura de objetos contiene muchas clases de objetos con distintas interfaces y se desea llevar a cabo operaciones sobre estos objetos que son distintas en cada clase concreta.

- Se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir dichas operaciones en las clases.

- Las clases que definen la estructura de objetos no cambian, pero las operaciones que se llevan a cabo sobre ellas.

4.12.6. Diagrama UML



4.12.7. Participantes

Visitor: declara una operación de visita para cada uno de los elementos concretos de la estructura de objetos. Esto es, el método visit().

VisitorConcreto : implementa cada una de las operaciones declaradas por Visitor.

Element: define la operación que le permite aceptar la visita de un Visitor.

ConcreteElement: implementa el método accept() que se limita a invocar su correspondiente método del Visitor.

ObjectStructure: gestiona la estructura de objetos y puede ofrecer una interfaz de alto nivel para permitir a los Visitor visitar a sus elementos.

4.12.8. Colaboraciones

El Element ejecuta el método de visitar y se pasa a sí mismo como parámetro.

4.12.9. Consecuencias

Facilita la inclusión de nuevas operaciones.

Agrupa las operaciones relacionadas entre sí.

La inclusión de nuevos ElementsConcretos es una operación costosa.

Posibilita visitar distintas jerarquías de objetos u objetos no relacionados por un padre común.

4.12.10. Implementación

En patrón Visitor posee un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método aceptar (accept()) que recibe al objeto visitador (visitor) como argumento. El visitador es una interfaz que tiene un método visit() diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz visitor de la forma: visitorClase1, visitorClase2... visitorClaseN. El método accept() de una clase elemento llama al método visit de su clase. Los visitadores concretos pueden entonces ser escritas para hacer una operación en particular.

Cada método visit() de un visitador concreto puede ser pensado como un método que no es de una sola clase, sino de un par de clases: el visitador concreto y clase elemento particular. Así el patrón visitor simula el envío doble (en inglés éste término se conoce como Double-Dispatch).

4.12.11. Código de muestra

En Argentina todos los productos pagan IVA. Algunos productos poseen una tasa reducida. Utilizaremos el Visitor para solucionar este problema.

```
public interface Visitable {
     public double accept(Visitor visitor);
}

public class ProductoDescuento implements Visitable {
     private double precio;

     public double accept(Visitor visitor) {
            return visitor.visit(this);
     }

     public double getPrecio() {
            return precio;
     }

     public void setPrecio(double precio) {
            this.precio = precio;
     }
}

public class ProductoNormal implements Visitable {
     private double precio;

     public double accept(Visitor visitor) {
            return visitor.visit(this);
     }

     public double getPrecio() {
            return precio;
     }

     public void setPrecio(double precio) {
            this.precio = precio;
     }
}


public interface Visitor {
     public double visit(ProductoNormal normal);
     public double visit(ProductoDescuento reducido);
}

public class IVA implements Visitor {
     private final double impuestoNormal = 1.21;
     private final double impuestoReducido = 1.105;
```

```
        public double visit(ProductoNormal normal) {
                return normal.getPrecio()*impuestoNormal;
        }

        public double visit(ProductoDescuento reducido) {
                return reducido.getPrecio()*impuestoReducido;
        }
}

Probemos como funciona este ejemplo:
    public static void main(String[] args) {
            ProductoDescuento producto1 = new ProductoDescuento();
            producto1.setPrecio(100);
            ProductoNormal producto2 = new ProductoNormal();
            producto2.setPrecio(100);

            IVA iva = new IVA();
            double resultado1 = producto1.accept(iva);
            double resultado2 = producto2.accept(iva);

            System.out.println(resultado1);
            System.out.println(resultado2);
    }

La salida por consola es:
110.5
121.0
```

4.12.12. Cuando utilizarlo

Dado que este patrón separa un algoritmo de la estructura de un objeto, es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general.

Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere "contaminar" a dichas clases.

4.12.13. Patrones relacionados

Interpreter: el visitor puede usarse para realizar la interpretación.

Composite: el visitor puede ayudar al Composite para aplicar una operación sobre la estructura del objeto definido en el composite.

5. Patrones de Creación (Creational Patterns)

5.1. Objetivos

5.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

5.2. Abstract Factory Pattern

5.2.1. Introducción y nombre

Abstract Factory. Creacional.

5.2.2. Intención

Ofrece una interfaz para la creación de familias de productos relacionados o dependientes sin especificar las clases concretas a las que pertenecen.

5.2.3. También conocido como

Factoría Abstracta, Kit.

5.2.4. Motivación

El problema que intenta solucionar este patrón es el de crear diferentes familias de objetos. Su objetivo principal es soportar múltiples estándares que vienen definidos por las diferentes familias de objetos. Es similar al Factory Method, sólo le añade mayor abstracción.

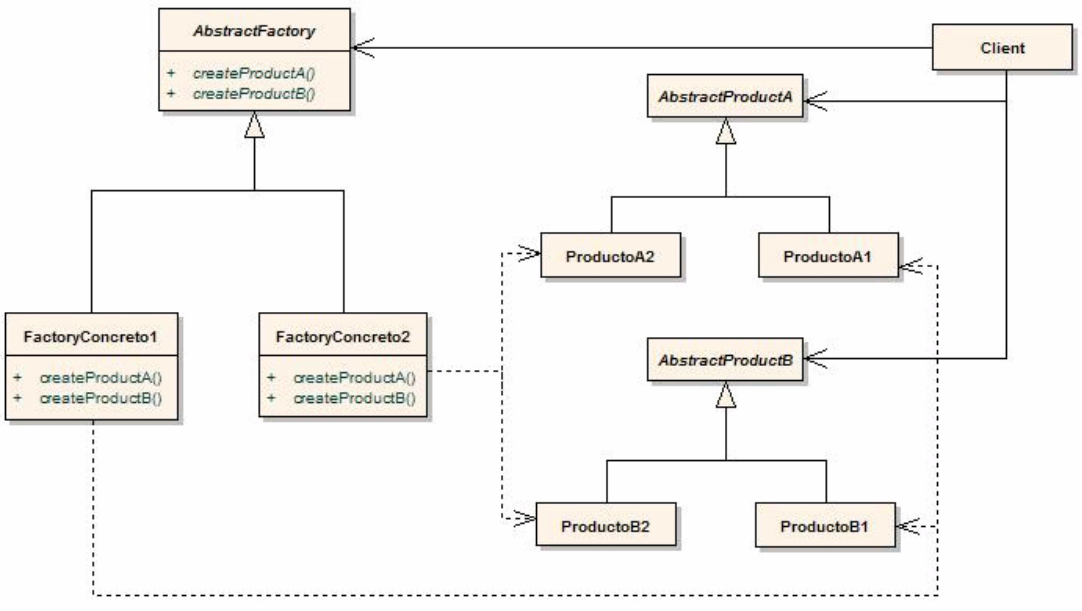
5.2.5. Solución

Se debe utilizar este patrón cuando:

- La aplicación debe ser independiente de como se crean, se componen y se representan sus productos.
- Un sistema se debe configurar con una de entre varias familias de productos.
- Una familia de productos relacionados están hechos para utilizarse juntos (hay que hacer que esto se cumpla).

- Para ofrecer una librería de clases, mostrando sólo sus interfaces y no sus implementaciones.

5.2.6. Diagrama UML



5.2.7. Participantes

AbstractFactory: declara una interfaz para la creación de objetos de productos abstractos.

ConcreteFactory: implementa las operaciones para la creación de objetos de productos concretos.

AbstractProduct: declara una interfaz para los objetos de un tipo de productos.

ConcreteProduct: define un objeto de producto que la correspondiente factoría concreta se encargaría de crear, a la vez que implementa la interfaz de producto abstracto.

5.2.8. Colaboraciones

Client: utiliza solamente las interfaces declaradas en la factoría y en los productos abstractos.

Una única instancia de cada FactoryConcreto es creada en tiempo de ejecución. AbstractFactory delega la creación de productos a sus subclases FactoryConcreto.

5.2.9. Consecuencias

Se oculta a los clientes las clases de implementación: los clientes manipulan los objetos a través de las interfaces o clases abstractas.

Facilita el intercambio de familias de productos: al crear una familia completa de objetos con una factoría abstracta, es fácil cambiar toda la familia de una vez simplemente cambiando la factoría concreta.

Mejora la consistencia entre productos: el uso de la factoría abstracta permite forzar a utilizar un conjunto de objetos de una misma familia.

Como inconveniente podemos decir que no siempre es fácil soportar nuevos tipos de productos si se tiene que extender la interfaz de la Factoría abstracta.

5.2.10. Implementación

Veamos los puntos más importantes para su implementación:

- Factorias como singletons: lo ideal es que exista una instancia de FactoryConcreto por familia de productos.
- Definir factorías extensibles: añadiendo un parámetro en las operaciones de creación que indique el tipo de objeto a crear.
- Para crear los productos se usa un Factory Method para cada uno de ellos.

5.2.11. Código de muestra

Hagamos de cuenta que tenemos dos familias de objetos:

1) La clase TV, que tiene dos hijas: Plasma y LCD.

2) La clase Color, que tiene dos hijas: Amarillo y Azul.

Más alla de todos los atributos/métodos que puedan tener la clase Color y TV, lo importante aquí es destacar que Color define un método abstracto:

```
    public abstract void colorea(TV tv);
```

Dado que es un método abstracto, Azul debe redefinirlo:

```
public void colorea(TV tv) {
System.out.println("Pintando de azul el "+ tv.getDescripcion());}
```

Lo mismo ocurre con Amarillo:

```
public void colorea(TV tv) {
System.out.println("Pintando de amarillo el "+ tv.getDescripcion());}
```

Escenario: nuestra empresa se dedica a darle un formato estético específico a los televisores LCD y Plasma. Se ha decidido que todos los LCD que saldrán al mercado serán azules y los plasma serán amarillos. Por este motivo se ha decidido realizar el patrón Abstract Factory:

```
public abstract class AbstractFactory {
     public abstract TV createTV();
     public abstract Color createColor();
}
```

Los Factory Concretos serían:

```
public class FactoryPlasmaAmarillo extends AbstractFactory {

     public Color createColor() {
            return new Amarillo();}

     public TV createTV() {
            return new Plasma();}
}


public class FactoryLcdAzul extends AbstractFactory {

     public Color createColor() {
            return new Azul();    }

     public TV createTV() {
            return new LCD();}
}
```

Y, por último, la clase Cliente de estas Factory:

```
public class EnsamblajeTV {
```

```
        public EnsamblajeTV(AbstractFacorty factory){
                Color color = factory.createColor();
                TV tv = factory.createTV();
                color.colorea(tv);}
        }
```

Como se puede observar el código es bastante genérico. El secreto de este patrón ocurre en los Factory Concretos que es donde se definen las reglas del negocio.

Veamos esto en funcionamiento:

```
    public static void main(String[] args) {
                // probando el factory LCD + Azul
                AbstractFactory f1 = new FactoryLcdAzul();
                EnsamblajeTV e = new EnsamblajeTV(f1);

                // probando el factory Plasma + Amarillo
                AbstractFactory f2 = new FactoryPlasmaAmarillo();
                EnsamblajeTV e2 = new EnsamblajeTV(f2);
                }
```

```
El resultado por consola es:
Pintando de azul el LCD
Pintando de amarillo el Plasma
```

5.2.12. Cuando utilizarlo

Este patrón proporciona una interfaz que permite crear familias de objetos relacionados entre sí, sin especificar (ni por lo tanto, conocer a priori) sus clases concretas.

Una factoría abstracta es una clase pero que los objetos que devuelve son a su vez factorías. Por este motivo, para que sea efectiva, estas factorías que devuelve, deben ser de la misma familia (es decir, tener antecesores comunes), como ocurría con las factorías normales.

Un ejemplo típico son los distintos look&feel de Java: en Java podemos tener el mismo programa con distintos aspectos cambiando solo una línea de código, la que se encarga de indicar el look&feel (en estos momentos los estándar son Metal, Motif y Windows) que queremos utilizar.

En Java esto es posible gracias a que la clase UIManager es una factoría abstracta que genera factorías de componentes visuales. Así , cuando creamos un componente visual para nuestra interfaz de usuario, Java acude al look&feel seleccionado, que como hemos dicho es una factoría, y le pide el componente escogido con el aspecto que corresponda.

Como consecuencia podemos decir que la situación ideal para la aplicación de este patrón es cuando existe la necesidad de creación de familias de productos relacionados sin especificar las clases concretas a las que pertenecen.

5.2.13. Patrones relacionados

Factory Method/Prototype: se pueden implementar con Factory Method o Prototype.

Singleton: las factorias concretas suelen ser Singleton.

5.3. Builder Pattern

5.3.1. Introducción y nombre

Builder. Creacional.

Permite la creación de una variedad de objetos complejos desde un objeto fuente, el cual se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo.

5.3.2. Intención

Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

5.3.3. También conocido como

Patrón constructor o virtual builder.

5.3.4. Motivación

Los objetos que dependen de un algoritmo tendrán que cambiar cuando el algoritmo cambia. Por lo tanto, los algoritmos que estén expuestos a dicho cambio deberían ser separados, permitiendo de esta manera reutilizar algoritmos para crear diferentes representaciones. En otras palabras, permite a un cliente construir un objeto complejo especificando sólo su tipo y contenido, ocultándole todos los detalles de la construcción del objeto.

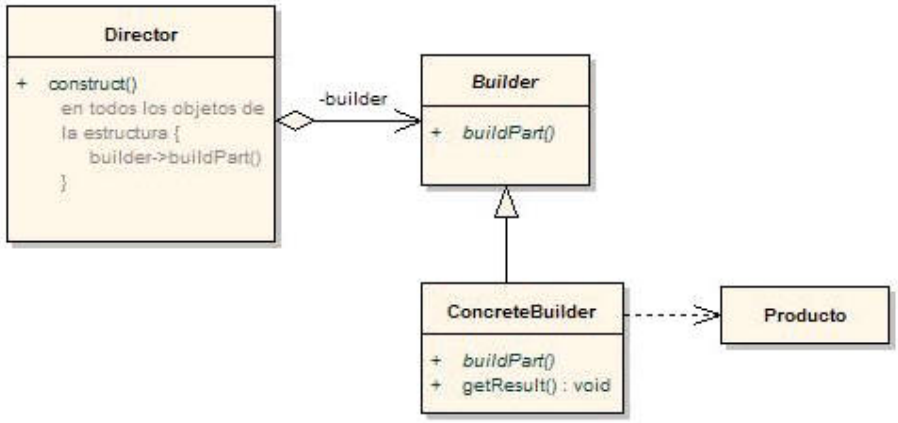
5.3.5. Solución

Se debe utilizar este patrón cuando sea necesario:

- Independizar el algoritmo de creación de un objeto complejo de las partes que constituyen el objeto y cómo se ensamblan entre ellas.

- Que el proceso de construcción permita distintas representaciones para el objeto construido, de manera dinámica.

5.3.6. Diagrama UML



5.3.7. Participantes

Producto: representa el objeto complejo a construir.

Builder: especifica una interface abstracta para la creación de las partes del Producto. Declara las operaciones necesarias para crear las partes de un objeto concreto.

ConcreteBuilder: implementa Builder y ensambla las partes que constituyen el objeto complejo.

Director: construye un objeto usando la interfaz Builder. Sólo debería ser necesario especificar su tipo y así poder reutilizar el mismo proceso para distintos tipos.

5.3.8. Colaboraciones

- El Cliente crea el objeto Director y lo configura con el objeto Builder deseado.
- El Director notifica al constructor cuándo una parte del Producto se debe construir.
- El Builder maneja los requerimientos desde el Director y agrega partes al producto.
- El Cliente recupera el Producto desde el constructor.

5.3.9. Consecuencias

- Permite variar la representación interna de un producto.
- El Builder ofrece una interfaz al Director para construir un producto y encapsula la representación interna del producto y cómo se juntan sus partes.
- Si se cambia la representación interna basta con crear otro Builder que respete la interfaz.
- Separa el código de construcción del de representación.
- Las clases que definen la representación interna del producto no aparecen en la interfaz del Builder.
- Cada ConcreteBuilder contiene el código para crear y juntar una clase específica de producto.
- Distintos Directores pueden usar un mismo ConcreteBuilder.
- Da mayor control en el proceso de construcción.
- Permite que el Director controle la construcción de un producto paso a paso.
- Sólo cuando el producto está acabado lo recupera el director del builder.

5.3.10. Implementación

Generalmente un Builder abstracto define las operaciones para construir cada componente que el Director podría solicitar.

El ConcreteBuilder implementa estas operaciones y le otorga la inteligencia necesaria para su creación.

Para utilizarlo el Director recibe un ContreBuilder.

5.3.11. Código de muestra

Realizaremos un ejemplo de un auto, el cual consta de diferentes partes para poder construirse.

```
public class Auto {
     private int cantidadDePuertas;
     private String modelo;
     private String marca;
     private Motor motor;

     public int getCantidadDePuertas() {
            return cantidadDePuertas;
     }
     public void setCantidadDePuertas(int cantidadDePuertas) {
            this.cantidadDePuertas = cantidadDePuertas;
     }
     public String getModelo() {
            return modelo;
     }
     public void setModelo(String modelo) {
            this.modelo = modelo;
     }
     public String getMarca() {
            return marca;
     }
     public void setMarca(String marca) {
            this.marca = marca;
     }
     public Motor getMotor() {
            return motor;
     }
     public void setMotor(Motor motor) {
            this.motor = motor;
     }
```

Utilizaremos la clase AutoBuilder para que sirve para base de construcción de los distintos tipos de Autos:

```
public abstract class AutoBuilder {
     protected Auto auto = new Auto();

     public Auto getAuto() {
            return auto;
            }
     public void crearAuto() {
      auto = new Auto();
      }

     public abstract void buildMotor();
```

```
    public abstract void buildModelo();
    public abstract void buildMarca();
    public abstract void buildPuertas();
}
```

Realizaremos dos builders concretos que son: FordBuilder y FiatBuilder. Cada Builder tiene el conocimiento necesario para saber como se construye su auto.

```
public class FiatBuilder extends AutoBuilder {

    public void buildMarca() {
        auto.setMarca("Fiat");
    }

    public void buildModelo() {
        auto.setModelo("Palio");
    }

    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(232323);
        motor.setPotencia("23 HP");
        auto.setMotor(motor);
    }

    public void buildPuertas() {
        auto.setCantidadDePuertas(2);
    }

}
```

A modo de simplificar el aprendizaje, estos builders construyen objetos relativamente sencillos. Se debe tener en cuenta que la complejidad para la construcción de los objetos suele ser mayor.

```
public class FordBuilder extends AutoBuilder {

    public void buildMarca() {
        auto.setMarca("Ford");
    }

    public void buildModelo() {
        auto.setModelo("Focus");
    }

    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(21212);
        motor.setPotencia("20 HP");
        auto.setMotor(motor);
    }

    public void buildPuertas() {
        auto.setCantidadDePuertas(4);

    }

}
```

Por último, realizaremos el Director. Lo primero que debe hacerse con esta clase es enviarle el tipo de auto que se busca construir (Ford, Fiat, etc). Luego, al llamar al método constructAuto(), la construcción se realizará de manera automática.

```
public class AutoDirector {
// No es necesario que exista la palabra Director
// Esta clase podría llamarse Concesionaria, Garage, FabricaDeAutos, //etc

    private AutoBuilder autoBuilder;

    public void constructAuto() {
       autoBuilder.buildMarca();
       autoBuilder.buildModelo();
       autoBuilder.buildMotor();
       autoBuilder.buildPuertas();
    }

    public void setAutoBuilder(AutoBuilder ab) {
     autoBuilder = ab;
}
    public Auto getAuto() {
     return autoBuilder.getAuto();
     }
}
```

La invocación desde un cliente sería:

```
public static void main(String[] args) {

            AutoDirector director = new AutoDirector();
            director.setAutoBuilder(new FordBuilder());
            director.constructAuto();
            Auto auto = director.getAuto();
            System.out.println(auto.getMarca());
```

5.3.12. Cuando utilizarlo

Esta patrón debe utilizarse cuando el algoritmo para crear un objeto suele ser complejo e implica la interacción de otras partes independientes y una coreografía entre ellas para formar el ensamblaje. Por ejemplo: la construcción de un objeto Computadora, se compondrá de otros muchos objetos, como puede ser un objeto PlacaDeSonido, Procesador, PlacaDeVideo, Gabinete, Monitor, etc.

5.3.13. Patrones relacionados

Con el patrón Abstract Factory también se pueden construir objetos complejos, pero el objetivo del patrón Builder es construir paso a paso, en cambio, el énfasis del Abstract Factory es tratar familias de objetos.

El objeto construido con el patrón Builder suele ser un Composite.

El patrón Factory Method se puede utilizar el Builder para decidir qué clase concreta instanciar para construir el tipo de objeto deseado.

El patrón Visitor permite la creación de un objeto complejo, en vez de paso a paso, dando todo de golpe como objeto visitante.

5.4. Factory Method Pattern

5.4.1. Introducción y nombre

Factory Method. Creacional. Libera al desarrollador sobre la forma correcta de crear objetos.

5.4.2. Intención

Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.

5.4.3. También conocido como

Virtual Constructor, Método de Factoría.

5.4.4. Motivación

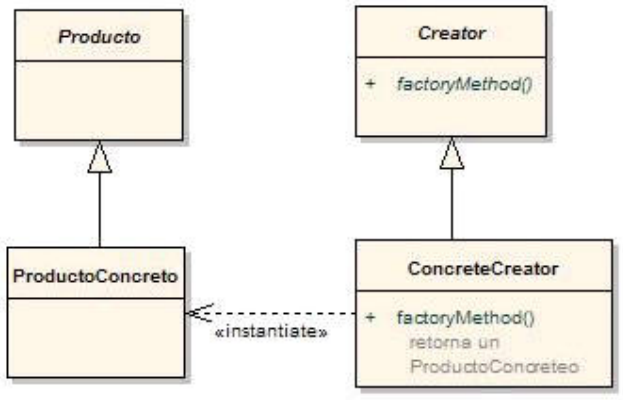
Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear, ya que la jerarquía de clases que tiene requiere que deba delegar la responsabilidad a una subclase.

5.4.5. Solución

Este patrón debe ser utilizado cuando:

- Una clase no puede anticipar el tipo de objeto que debe crear y quiere que sus subclases especifiquen dichos objetos.
- Hay clases que delegan responsabilidades en una o varias subclases.
- Una aplicación es grande y compleja y posee muchos patrones creacionales.

5.4.6. Diagrama UML



5.4.7. Participantes

Responsabilidad de cada clase participante:

Creator: declara el método de fabricación, que devuelve un objeto de tipo product. Puede llamar a dicho método para crear un objeto producto .

ConcretCreator: redefine el método de fabricación para devolver un objeto concretProduct.

5.4.8. Colaboraciones

ProductoConcreto: es el resultado final. El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.

5.4.9. Consecuencias

Como ventaja se destaca que elimina la necesidad de introducir clases específicas en el código del creador. Solo maneja la interfaz Product, por lo que permite añadir cualquier clase ConcretProduct definida por el usuario.

Otra ventaja: es más flexible crear un objeto con un Factory Method que directamente: un método factoría puede dar una implementación por defecto.

Un inconveniente es tener que crear una subclase de Creator en los casos en los que esta no fuera necesaria de no aplicar el patrón.

5.4.10. Implementación

Dificultades, técnicas y trucos a tener en cuenta al aplicar el PD

Implementación de la clase Creator

- El método factoría es abstracto
- El método factoría proporciona una implementación por defecto:

Permite extensibilidad: se pone la creación de objetos en una operación separada por si el usuario quiere cambiarla

5.4.11. Código de muestra

Asumamos que tenes una clase Triángulo y necesitamos crear un tipo de triángulo: escaleno, isosceles o equilatero. Para ello, esta la clase abstracta Triangulo y sus clases hijas:

```
public abstract class Triangulo {
     private int ladoA;
     private int ladoB;
     private int ladoC;

     public Triangulo(int ladoA, int ladoB, int ladoC){
            setLadoA(ladoA);
            setLadoB(ladoB);
            setLadoC(ladoC);
     }

     public abstract String getDescripcion();
     public abstract double getSuperficie();
     public abstract void dibujate();


public class Equilatero extends Triangulo {

     public Equilatero(int anguloA, int anguloB, int anguloC) {
            super(anguloA, anguloB, anguloC);}

     public String getDescripcion() {
            return "Soy un Triangulo Equilatero";}

     public double getSuperficie() {
            // algoritmo para calcular superficie
            return 0;      }
```

```
    public void dibujate() {
           // algoritmo para dibujarse
    }}


public class Escaleno extends Triangulo {

    public Escaleno(int ladoA, int ladoB, int ladoC) {
           super(ladoA, ladoB, ladoC);   }

    public String getDescripcion() {
           return "Soy un Triangulo Escaleno";   }

    public double getSuperficie() {
           // algoritmo para calcular superficie
           return 0;}

    public void dibujate() {
           // algoritmo para dibujarse
    }}


public class Isosceles extends Triangulo {

    public Isosceles(int ladoA, int ladoB, int ladoC) {
           super(ladoA, ladoB, ladoC);   }

    public String getDescripcion() {
           return "Soy un Triangulo Isosceles";}

    public double getSuperficie() {
           // algoritmo para calcular superficie
           return 0;}

    public void dibujate() {
           // algoritmo para dibujarse
    }}
```

Para evitar que nuestros clientes deban conocer la estructura de nuestra jerarquía creamos un Factory de triángulos:

```
public class TrianguloFactory{

public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC){

    if ((ladoA == ladoB) && (ladoA == ladoC)) {
                  return new Equilatero(ladoA,ladoB,ladoC);}

    else if ((ladoA != ladoB)&&(ladoA != ladoC)&&(ladoB != ladoC)) {
                  return new Escaleno(ladoA,ladoB,ladoC);
           }
    else {
                  return new Isosceles(ladoA,ladoB,ladoC);
           }}}
```

De esta forma, no sólo no tienen que conocer nuestra estructura de clases, sino que además tampoco es necesario que conozcan nuestra implementación (el conocimiento del algoritmo que resuelve que clases se deben crear).

Simplemente deberian crear un triángulo de la siguiente forma:

```
public static void main(String[] args) {

            TrianguloFactory factory = new TrianguloFactory();

            Triangulo triangulo = factory.createTriangulo(10, 10, 10);
            System.out.println(triangulo.getDescripcion());     }
```

```
La salida por consola es: Soy un Triangulo Equilatero
```

Cabe aclara que las clases Factory deberian ser Singleton, pero para evitar confundir al estudiante nos hemos concentrado sólo en este patrón.

5.4.12. Cuando utilizarlo

El escenario típico para utilizar este patrón es cuando existe una jerarquía de clases complejas y la creación de un objeto específico obliga al cliente a tener que conocer detalles específicos para poder crear un objeto. En este caso se puede utilizar un Factory Method para eliminar la necesidad de que el cliente tenga la obligación de conocer las todas especificaciones y variaciones posibles.

Por otro lado, imaginemos una aplicación grande, realizada por un grupo de desarrolladores que han utilizado muchos patrones creacionales. Esto hace que el grupo de programadores pierda el control sobre como se debe crear una clase. Es decir, cual es un Singleton o utiliza un Builder o un Prototype. Entonces se realiza un Factory Method para que instancie los objetos de una manera estandard. El desarrollador siempre llama a uno o varios Factory y se olvida de la forma en que se deben crear todos los objetos.

5.4.13. Patrones relacionados

Abstract Factory: suele ser implementada por varios Factory Method.

Prototype y Builder: si bien hay casos donde parece que se excluyen mutuamente, en la mayoría de los casos suelen trabajar juntos.

Singleton: un Factory Method suele ser una clase Singleton.

5.5. Prototype Pattern

5.5.1. Introducción y nombre

Prototype. Creacional. Los objetos se crean a partir de un modelo.

5.5.2. Intención

Permite a un objeto crear objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos. El objetivo de este patrón es especificar prototipos de objetos a crear. Los nuevos objetos que se crearan se clonan de dichos prototipos. Vale decir, tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente.

5.5.3. También conocido como:

Patrón prototipo.

5.5.4. Motivación

Este patrón es necesario en ciertos escenarios es preciso abstraer la lógica que decide que tipos de objetos utilizará una aplicación, de la lógica que luego usarán esos objetos en su ejecución. Los motivos de esta separación pueden ser variados, por ejemplo, puede ser que la aplicación deba basarse en alguna configuración o parámetro en tiempo de ejecución para decidir el tipo de objetos que se debe crear.

Por otro lado, también aplica en un escenario donde sea necesario la creación de objetos parametrizados como "recién salidos de fábrica" ya listos para utilizarse, con la gran ventaja de la mejora de la performance: clonar objetos es más rápido que crearlos y luego setear cada valor en particular.

5.5.5. Solución

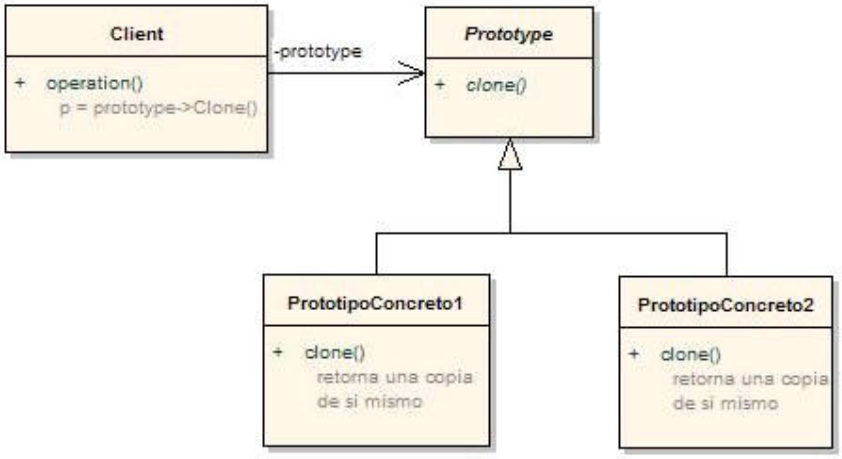
Situaciones en las que resulta aplicable.

Se debe aplicar este patrón cuando:

- Las clases a instanciar sean especificadas en tiempo de ejecución.
- Los objetos a crear tienen características comunes, en cuanto a los valores de sus atributos.
- Se quiera evitar la construcción de una jerarquía Factory paralela a la jerarquía de clases de producto.

5.5.6. Diagrama UML

Estructura en un diagramas de clases.



5.5.7. Participantes

Responsabilidad de cada clase participante.

Cliente: solicita nuevos objetos.

Prototype: declara la interface del objeto que se clona. Suele ser una clase abstracta.

PrototypeConcreto: las clases en este papel implementan una operación por medio de la clonación de sí mismo.

5.5.8. Colaboraciones

Cliente: crea nuevos objetos pidiendo al prototipo que se clone.

Los objetos de Prototipo Concreto heredan de Prototype y de esta forma el patrón se asegura de que los objetos prototipo proporcionan un conjunto consistente de métodos para que los objetos clientes los utilicen.

5.5.9. Consecuencias

- Un programa puede dinámicamente añadir y borrar objetos prototipo en tiempo de ejecución. Esta es una ventaja que no ofrece ninguno de los otros patrones de creación.
- Esconde los nombres de los productos específicos al cliente.
- Se pueden especificar nuevos objetos prototipo variando los existentes.
- La clase Cliente es independiente de las clases exactas de los objetos prototipo que utiliza. y, además, no necesita conocer los detalles de cómo construir los objetos prototipo.
- Clonar un objeto es más rápido que crearlo.
- Se desacopla la creación de las clases y se evita repetir la instanciación de objetos con parámetros repetitivos.

5.5.10. Implementación

Dificultades, técnicas y trucos a tener en cuenta al aplicar el PD

Debido a que el patrón Prototye hace uso del método clone(), es necesaria una mínima explicación de su funcionamiento: todas las clases en Java heredan un método de la clase Object llamado clone. Un método clone de un objeto retorna una copia de ese objeto. Esto solamente se hace para instancias de clases que dan permiso para ser clonadas. Una clase da permiso para que su instancia sea clonada si, y solo si, ella implementa el interface Cloneable.

Por otro lado, es importante destacar que si va a variar el número de prototipos se puede utilizar un "administrador de prototipos". Otra opción muy utilizada es un Map como se ve en el ejemplo.

Debe realizarse un gestor de prototipos, para que realice el manejo de los distintos modelos a clonar.

Por último, se debe inicializar los prototipos concretos.

5.5.11. Código de muestra

Imaginemos que estamos haciendo el software para una empresa que vende televisores plasma y LCD. La gran mayoría de los TVs plasma comparten ciertas características: marca, color, precio, etc. Lo mismo ocurre con los LCD. Esto es normal en ciertos rubros ya que compran por mayor y se obtienen datos muy repetitivos en ciertos productos. También es muy normal en las fábricas: salvo algún serial, los productos son todos iguales.

Volviendo a nuestro ejemplo, se decidió realizar una clase genérica TV con los atributos básicos que debe tener un televisor:

```
public abstract class TV implements Cloneable{
    private String marca;
    private int pulgadas;
    private String color;
    private double precio; //todos con sus get y set

public TV(String marca,int pulgadas, String color, double precio){
            setMarca(marca);
            setPulgadas(pulgadas);
            setPrecio(precio);
            setColor(color);
    }

public Object clone() throws CloneNotSupportedException{
            return super.clone();
    }
```

También tenemos los televisores específicos:

```
public class Plasma extends TV{
    private double anguloVision;
    private double tiempoRespuesta; //todos con sus get y set

public Plasma(String marca,int pulgadas, String color, double precio, double
anguloVision, double tiempoRespuesta){

            super(marca, pulgadas,  color,  precio);
            setAnguloVision(anguloVision);
```

```
                setTiempoRespuesta(tiempoRespuesta);

        }


    public class LCD extends TV {
        private double costoFabricacion; // con sus get y set


    public LCD(String marca, int pulgadas, String color, double precio, double
    costoFabricacion){

                super(marca, pulgadas,  color,  precio);
                setCostoFabricacion(costoFabricacion);
        }
```

Debido a que se decidió realizar ciertos prototipos, estos se ven plasmados en la clase TvPrototype:

```
    public class TvPrototype{
    private HashMap<String, TV> prototipos = new HashMap<String, TV>();

    public TvPrototype(){

    Plasma plasma = new Plasma("Sony", 21, "Plateado", 399.99, 90, 0.05);
    LCD lcd = new LCD("Panasonic",42, "Plateado", 599.99, 290);

                prototipos.put("Plasma", plasma);
                prototipos.put("LCD", lcd);
        }

    public Object prototipo(String tipo)throws CloneNotSupportedException {
                return prototipos.get(tipo).clone();
        }


    La invocación al método prototipo() sería:

    public static void main(String[] args) throws Exception {
                TvPrototype tvp = new TvPrototype();
                TV tv = (TV) tvp.prototipo("Plasma");

                System.out.println(tv.getPrecio());
        }

    El resultado de la consola es: 399.99
```

5.5.12. Cuando utilizarlo

Este patrón debe ser utilizado cuando un sistema posea objetos con datos repetitivos: por ejemplo, si una biblioteca posee una gran cantidad de libros de una misma editorial, mismo idioma, etc.

Por otro lado, el hecho de poder agregar o eliminar prototipos en tiempo de ejecución es una gran ventaja que lo hace muy flexible.

5.5.13. Patrones relacionados

Abstract Factory: el patrón Abstract Factory puede ser una buena alternativa al Prototype donde los cambios dinámicos que Prototype permite para los objetos prototipo no son necesarios. Pueden competir en su objetivo, pero también pueden colaborar entre sí.

Facade: la clase cliente normalmente actúa comúnmente como un facade que separa las otras clases que participan en el patrón Prototype del resto del programa.

Factory Method: puede ser una alternativa al Prototype cuando los objetos prototipo nunca contiene más de un objeto.

Singleton: una clase Prototype suele ser Singleton.

5.6. Singleton Pattern

5.6.1. 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.

5.6.2. Intención

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

5.6.3. 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.

5.6.4. 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.

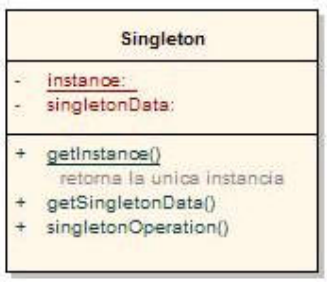
5.6.5. Solución

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.

5.6.6. Diagrama UML



5.6.7. 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.

5.6.8. Colaboraciones

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

5.6.9. 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.

5.6.10. 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()

5.6.11. Código de muestra

```
public class Singleton {
    private static Singleton instance;

    private Singleton(){
    }

    public 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
```

5.6.12. Cuando 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.

5.6.13. Patrones relacionados

Factory Method, Builder y Prototype, que próximamente los veremos, suelen ser Singleton

6. Patrones de Estructura (Structural Patterns)

6.1. Objetivos

6.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

6.2. Adapter Pattern

6.2.1. Introducción y nombre

Adapter. Estructural. Busca una manera estandarizada de adaptar un objeto a otro.

6.2.2. Intención

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

6.2.3. También conocido como

Adaptador. Wrapper (al patrón Decorator también se lo llama Wrapper, con lo cual es nombre Wrapper muchas veces se presta a confusión).

6.2.4. Motivación

Una clase Adapter implementa un interfaz que conoce a sus clientes y proporciona acceso a una instancia de una clase que no conoce a sus clientes, es decir convierte la interfaz de una clase en una interfaz que el cliente espera. Un objeto Adapter proporciona la funcionalidad prometida por un interfaz sin tener que conocer que clase es utilizada para implementar ese interfaz. Permite trabajar juntas a dos clases con interfaces incompatibles.

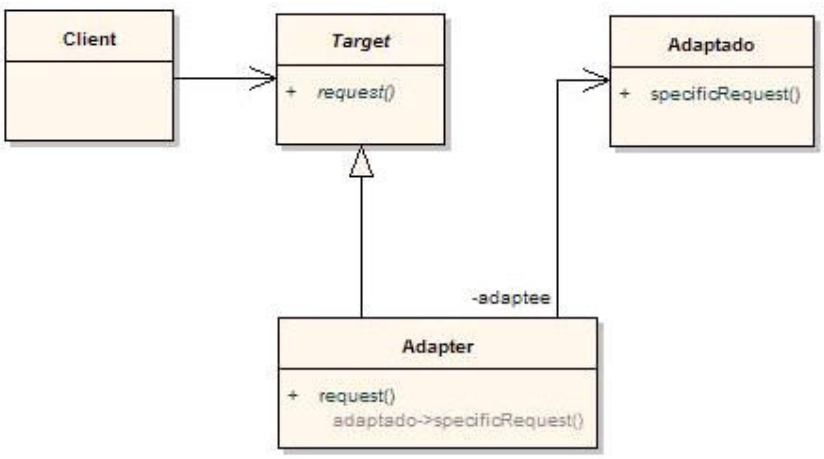
6.2.5. Solución

Este patrón se debe utilizar cuando:

- Se quiere utilizar una clase que llame a un método a través de una interface, pero se busca utilizarlo con una clase que no implementa ese interface.

- Se busca determinar dinámicamente que métodos de otros objetos llama un objeto.
- No se quiere que el objeto llamado tenga conocimientos de la otra clase de objetos.

6.2.6. Diagrama UML



6.2.7. Participantes

Target: define la interfaz específica del dominio que Cliente usa.

Cliente: colabora con la conformación de objetos para la interfaz Target.

Adaptado: define una interfaz existente que necesita adaptarse

Adapter: adapta la interfaz de Adaptee a la interfaz Target

6.2.8. Colaboraciones

El Cliente llama a las operaciones sobre una instancia Adapter. De hecho, el adaptador llama a las operaciones de Adaptee que llevan a cabo el pedido.

6.2.9. Consecuencias

El cliente y las clases Adaptee permanecen independientes unas de las otras.

Puede hacer que un programa sea menos entendible.

Permite que un único Adapter trabaje con muchos Adaptees, es decir, el Adapter por sí mismo y las subclases (si es que la tiene). El Adapter también puede agregar funcionalidad a todos los Adaptees de una sola vez.

6.2.10. Implementación

Se debe tener en cuenta que si bien el Adapter tiene una implementación relativamente sencilla, se puede llevar a cabo con varias técnicas:

1) Creando una nueva clase que será el Adaptador, que extienda del componente existente e implemente la interfaz obligatoria. De este modo tenemos la funcionalidad que queríamos y cumplimos la condición de implementar la interfaz.

2) Pasar una referencia a los objetos cliente como parámetro a los costructores de los objetos adapter o a uno de sus métodos. Esto permite al objeto adapter ser utilizado con cualquier instancia o posiblemente muchas instancias de la clase Adaptee. En este caso particular, el Adapter tiene una implementación casi idéntica al patrón Decorator.

3) Hacer la clase Adapter una clase interna de la clase Adaptee. Esto asume que tenemos acceso al código de dicha clase y que es permitido la modificación de la misma.

4) Utilizar sólo interfaces para la comunicación entre los objetos.

Las opciones más utilizadas son la 1 y la 4.

6.2.11. Código de muestra

Vamos a plantear el siguiente escenario: nuestro código tiene una clase Persona (la llamamos PersonaVieja) que se utiliza a lo largo de todo el código y hemos importado un API que también necesita trabajar con una clase Persona (la

llamamos PersonaNueva), que si bien son bastante similares tienen ciertas diferencias:

Nosotros trabajamos con los atributos nombre, apellido y fecha de nacimiento.

Sin embargo, la PersonaNueva tiene un solo atributo nombre (que es el nombre y apellido de la persona en cuestión) y la edad actual, en vez de la fecha de nacimiento.

Para esta situación lo ideal es utilizar el Adapter:

```
public class PersonaVieja implements IPersonaVieja{
     private String nombre;
     private String apellido;
     private Date fechaDeNacimiento;

     public String getNombre() {
            return nombre;
     }
     public void setNombre(String nombre) {
            this.nombre = nombre;
     }
     public String getApellido() {
            return apellido;
     }
     public void setApellido(String apellido) {
            this.apellido = apellido;
     }
     public Date getFechaDeNacimiento() {
            return fechaDeNacimiento;
     }
     public void setFechaDeNacimiento(Date fechaDeNacimiento) {
            this.fechaDeNacimiento = fechaDeNacimiento;
     }
}

public interface IPersonaVieja {
     public String getNombre();
     public void setNombre(String nombre);
     public String getApellido();
     public void setApellido(String apellido);
     public Date getFechaDeNacimiento();
     public void setFechaDeNacimiento(Date fechaDeNacimiento);
}

public class ViejaToNuevaAdapter implements IPersonaNueva {
     private IPersonaVieja vieja;

     public ViejaToNuevaAdapter(IPersonaVieja vieja){
            this.vieja = vieja;
     }

     public int getEdad() {
            GregorianCalendar c = new GregorianCalendar();
            GregorianCalendar c2 = new GregorianCalendar();
            c2.setTime(vieja.getFechaDeNacimiento());
            return c.get(1)-c2.get(1);
     }
```

```
	public String getNombre() {
		return vieja.getNombre() +" " + vieja.getApellido();
	}

	public void setEdad(int edad) {
		GregorianCalendar c = new GregorianCalendar();
		int anioActual = c.get(1);
		c.set(1, anioActual-edad);
		vieja.setFechaDeNacimiento(c.getTime());
	}

	public void setNombre(String nombreCompleto) {
		String []name = nombreCompleto.split(" ");
		String firstName = name[0];
		String lastName = name[1];
		vieja.setNombre(firstName);
		vieja.setApellido(lastName);
	}
}
```

Veremos como funciona este ejempplo:

```
	public static void main(String[] args) {
		PersonaVieja pv = new PersonaVieja();
		pv.setApellido("Perez");
		pv.setNombre("Maxi");
		GregorianCalendar g = new GregorianCalendar();
		g.set(2000, 01, 01);
		// seteamos que nacio en el año 2000
		Date d = g.getTime();
		pv.setFechaDeNacimiento(d);

		ViejaToNuevaAdapter adapter = new ViejaToNuevaAdapter(pv);

		System.out.println(adapter.getEdad());
		System.out.println(adapter.getNombre());

		adapter.setEdad(10);
		adapter.setNombre("Juan Perez");

		System.out.println(adapter.getEdad());
		System.out.println(adapter.getNombre());
	}
```

La salida por consola es:

```
8
Maxi Perez
10
Juan Perez
```

6.2.12. Cuando utilizarlo

Este patrón convierte la interfaz de una clase en otra interfaz que el cliente espera. Esto permite a las clases trabajar juntas, lo que de otra manera no podrían hacerlo debido a sus interfaces incompatibles.

Por lo general, esta situación se da porque no es posible modificar la clase original, ya sea porque no se tiene el código fuente de la clase o porque la clase es una clase de propósito general, y es inapropiado para ella implementar un interface par un propósito específico. En resumen, este patrón debe ser aplicado cuando debo transformar una estructura a otra, pero sin tocar la original, ya sea porque no puedo o no quiero cambiarla.

6.2.13. Patrones relacionados

Decorator: una forma de llevar a cabo la implementación del Adapter se asemeja a la imlementación del Decorator.

Iterator: es un versión especializada del patrón Adapter para acceder secuencialmente al contenido de una colección de objetos.

6.3. Bridge Pattern

6.3.1. Introducción y nombre

Bridge. Estructural. Desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

6.3.2. Intención

El patrón Bridge es una técnica usada en programación para desacoplar la interface de una clase de su implementación, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra.

6.3.3. También conocido como

Puente, Handle/Body.

6.3.4. Motivación

Cuando un objeto tiene unas implementaciones posibles, la manera habitual de implementación es el uso de herencias. Muchas veces la herencia se puede tornar inmanejable y, por otro lado, acopla el código cliente con una implementación concreta. Este patrón busca eliminar la inconveniencia de esta solución.

6.3.5. Solución

Este patrón debe ser utilizado cuando:

- Se desea evitar un enlace permanente entre la abstracción y su implementación. Esto puede ser debido a que la implementación debe ser seleccionada o cambiada en tiempo de ejecución.

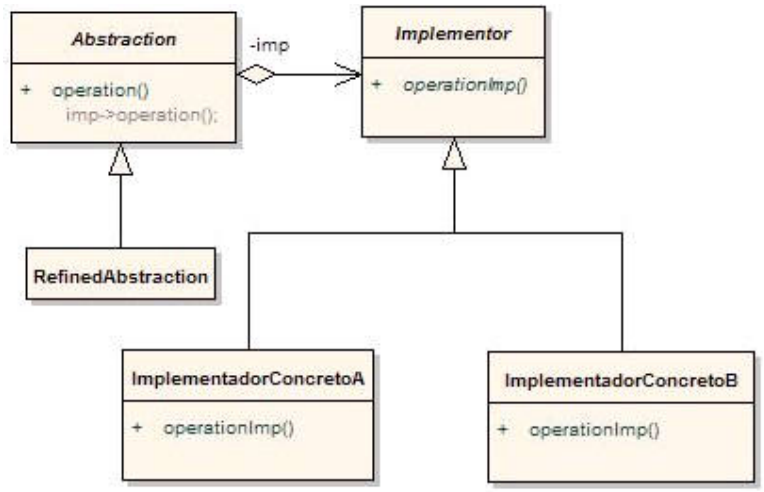
- Tanto las abstracciones como sus implementaciones deben ser extensibles por medio de subclases. En este caso, el patrón Bridge permite combinar abstracciones e implementaciones diferentes y extenderlas independientemente.

- Cambios en la implementación de una abstracción no deben impactar en los clientes, es decir, su código no debe tener que ser recompilado.

- Se desea compartir una implementación entre múltiples y este hecho debe ser escondido a los clientes.

- Permite simplificar jerarquías demasiado pobladas.

6.3.6. Diagrama UML



6.3.7. Participantes

- Abstraction: define una interface abstracta. Mantiene una referencia a un objeto de tipo Implementor.

- RefinedAbstraction: extiende la interface definida por Abstraction
- Implementor: define la interface para la implementación de clases. Esta interface no se tiene que corresponder exactamente con la interface de Abstraction; de hecho, las dos interfaces pueden ser bastante diferentes entre sí. Típicamente la interface Implementor provee sólo operaciones primitivas, y Abstraction define operaciones de alto nivel basadas en estas primitivas.
- ImplementadorConcreto: implementa la interface de Implementor y define su implementación concreta.

6.3.8. Colaboraciones

Abstraction: emite los pedidos de los clientes a su objeto Implementor. El cliente no tiene que conocer los detalles de la implementación.

6.3.9. Consecuencias

- Desacopla interface e implementación: una implementación no es limitada permanentemente a una interface. Le es posible a un objeto cambiar su implementación en tiempo de ejecución. Este desacoplamiento fomenta las capas, que pueden conducir a un sistema mejor estructurado.
- La parte de alto nivel de un sistema sólo tiene que conocer Abstraction e Implementor.
- Mejora la extensibilidad: se puede extender las jerarquías de Abstraction e Implementor independientemente.
- Esconde los detalles de la implementación a los clientes.

6.3.10. Implementación

Se debe tener en cuenta donde en situaciones donde existe sólo una implementación, crear una clase Implementor abstracta no es necesario. Este es un

caso especial del patrón; hay una relación uno-a-uno entre Abstraction e Implementor.

6.3.11. Código de muestra

Imaginemos que necesitamos dibujar distintas figuras geométricas (círculo, rectángulo, etc). Cada figura geométrica puede ser dibujada con diferentes tipos de líneas (normal, punteada, etc).

Realizaremos la clase Dibujo que sería la implemtación abstracta:

```
public abstract class Dibujo {
     public abstract  void dibujaRectangulo
     (double x1, double y1,double x2, double y2);

     abstract public void dibujaCirculo(double x,double y,double r);
}
```

Veamos las implementaciones concretas:

```
public class DibujandoPunteado extends Dibujo {
     public void dibujaRectangulo(double x1, double y1, double x2, double
y2) {
             System.out.println("Dibujando un rectangulo punteado...");  }

     public void dibujaCirculo(double x, double y, double r) {
             System.out.println("Dibujando un circulo punteado...");
}


public class DibujandoNormal extends Dibujo {
     public void dibujaRectangulo(double x1, double y1, double x2, double
y2) {
             System.out.println("Dibujando un rectangulo normal...");
     }

     public void dibujaCirculo(double x, double y, double r) {
             System.out.println("Dibujando un circulo...");
     }
}
```

Obviamente estas clases no debería realizar una simple salida por consola sino que debería poseer el algoritmo del dibujo, pero nos sirve a modo de ejemplo.

Veamos ahora las figuras geométricas:

```
public abstract class Forma {
  private Dibujo _d;
```

```
  public Forma(Dibujo d) {
            _d = d;
    }
  public abstract void dibuja();

  public void dibujaRectangulo(double x1, double y1, double x2, double y2) {
            _d.dibujaRectangulo(x1, y1, x2, y2);
    }

  public void dibujaCirculo(double x, double y, double r) {
            _d.dibujaCirculo(x, y, r);}
}


public class Circulo extends Forma {
  private double _x, _y, _r;
  public Circulo(Dibujo d, double x, double y, double r) {
            super(d);
            _x = x;
            _y = y;
            _r = r;
    }

  public void dibuja() {
            dibujaCirculo(_x, _y, _r);
    }
}

public class Rectangulo extends Forma {private double _x1,_x2,_y1,_y2;
 public Rectangulo(Dibujo dp,double x1,double y1,double x2,double y2) {
    super( dp) ;
    _x1= x1; _x2= x2 ;_y1= y1; _y2= y2;
    }
  public void dibuja () {
    dibujaRectangulo(_x1,_y1,_x2,_y2);}
}
```

Veamos como funciona el ejemplo:

```
    public static void main(String[] args) {
            Dibujo dibujo = new DibujandoPunteado();
            Rectangulo rectangulo= new Rectangulo(dibujo,1,1,2,2);
            rectangulo.dibuja();

            Dibujo dibujo2= new DibujandoNormal ();
            Circulo circulo = new Circulo(dibujo2,2,2,3);
            circulo.dibuja();
    }
```

La salida por consola es:

```
Dibujando un rectangulo punteado...

Dibujando un circulo...
```

6.3.12. Cuando utilizarlo

Este patrón debe ser utilizado concretamente cuando se quiere se quiera compartir una misma implementación entre múltiples objetos y, por otro lado, aislar a los clientes de cambios en la implementación evitando recompilaciones de código.

Tomemos en cuenta el siguiente ejemplo: interfaces gráficas que soporta distintas

plataformas de Windows, Unix y Linux.

Tenemos la clase abstracta Ventana, de la cual heredan VentanaWindows, VentanaUnix y VentanaLinux.

Después tenemos una ventana con ícono…que hereda de Ventana. Si utilizamos herencia tendríamos que aumentar 4 clases: VentanaIcono, VentanaIconoUnix, VentanaIconoLinux y VentanaIconoWindows. Es decir, por cada sistema operativo deberíamos tener una implementación distinta. Que pasaría si se agrega un nuevo tipo de ventana? Y que pasaría si luego se agregaría un nuevo sistema operativo? Básicamente el código se vuelve inmanejable.

Este caso es ideal para utilizar el patrón Bridge dado que permite, de forma transparente al cliente, que distintos objetos compartan la misma implementación. Por esta razón, es que Java utiliza el patrón Bridge en sus componentes gráficos.

6.3.13. Patrones relacionados

Se suele utilizar el patrón abstract factory para elegir una implementación.

Adapter: muchas veces es confundido con el Bridge. Sebe tener en cuenta que el Adapter se suele aplicar a clases ya diseñadas, mientras que el patrón Puente se usa en las primeras etapas de diseño.

6.4. Composite Pattern

6.4.1. Introducción y nombre

Composite. Estructural. Permite construir objetos complejos componiendo de forma recursiva objetos similares en una estructura de árbol.

6.4.2. Intención

El patrón Composite sirve para construir algoritmos u objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera.

6.4.3. También conocido como

Compuesto.

6.4.4. Motivación

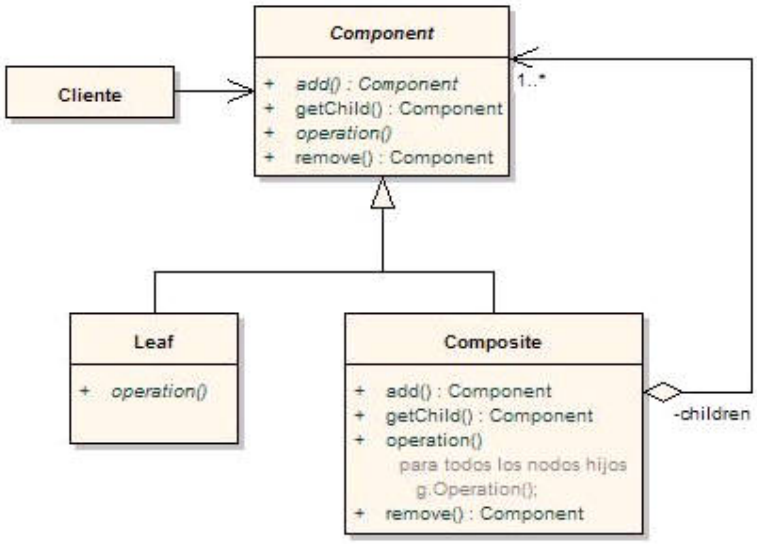
Este patrón propone una solución basada en composición recursiva. Además, describe como usar correctamente esta recursividad. Esto permite tratar objetos individuales y composiciones de objetos de manera uniforme. Este patrón busca representar una jerarquía de objetos conocida como “parte-todo”, donde se sigue la teoría de que las "partes" forman el "todo", siempre teniendo en cuenta que cada "parte" puede tener otras "parte" dentro.

6.4.5. Solución

Se debe utilizar este patrón cuando:

- Se busca representar una jerarquía de objetos como “parte-todo”.
- Se busca que el cliente puede ignorar la diferencia entre objetos primitivos y compuestos (para que pueda tratarlos de la misma manera).

6.4.6. Diagrama UML



6.4.7. Participantes

Component

- Declara la interface para los objetos de la composición.
- Implementa un comportamiento común entre las clases.
- Declara la interface para acceso y manipulación de hijos.

- Declara una interface de manipulación a los padres en la estructura recursiva.

Leaf

- Representa los objetos “hoja” (no poseen hijos).
- Define comportamientos para objetos primitivos.

Composite

- Define un comportamiento para objetos con hijos.
- Almacena componentes hijos.
- Implementa operaciones de relación con los hijos.

Cliente

- Manipula objetos de la composición a través de Component.

6.4.8. Colaboraciones

Los clientes usan la interfaz de Component para interactuar con objetos en la estructura Composite. Si el receptor es una hoja, la interacción es directa. Si es un Composite, se debe llegar a los objetos “hijos”, y puede llevar a utilizar operaciones adicionales.

6.4.9. Consecuencias

Define jerarquías entre las clases.

Simplifica la interacción de los clientes.

Hace más fácil la inserción de nuevos hijos.

Hace el diseño más general.

6.4.10. Implementación

Hay muchas formas de llevar a cabo este patrón. Muchas implementaciones implican referencias explicitas al padre para simplificar la gestión de la estructura.

Si el orden de los hijos provoca problemas utilizar Iterator.

Compartir componentes reduce los requerimientos de almacenamiento.

Para borrar elementos hacer un compuesto responsable de suprimir los hijos.

La mejor estructura para almacenar elementos depende de la eficiencia: si se atraviesa la estructura con frecuencia se puede dejar información en los hijos. Por otro lado, el componente no siempre debería tener una lista de componentes, esto depende de la cantidad de componentes que pueda tener.

6.4.11. Código de muestra

Imaginemos una escuela, que esta compuesta de diferentes sectores (Dirección, Aulas, etc) y personas (profesores, alumnos, portero, etc). Se busca que la escuela pueda identificar las personas que posee y la edad de cada una. Todos deben identificarse con la misma interface:

```
public interface IPersonal {
    public void getDatosPersonal();
}


public class Composite implements IPersonal{
    private ArrayList<IPersonal> a = new ArrayList<IPersonal>();

    public void agrega(IPersonal p){
           a.add(p);}

    public void getDatosPersonal() {
           for (int i = 0; i < a.size(); i++) {
                  IPersonal p = a.get(i);
                  p.getDatosPersonal();}}
    }


public class Escuela extends Composite{}
public class Direccion extends Composite {}
public class Aula extends Composite{}

public class Persona implements IPersonal{
    private String nombre;
    private int edad;
```

```
	public Persona(String nombre, int edad){
		setEdad(edad);
		setNombre(nombre);
	}

	public void getDatosPersonal() {
		String msg = "Me llamo "+nombre;
		msg = msg + ", tengo "+edad+" años";
		System.out.println(msg);
	}

	public String getNombre() {
		return nombre;
	}

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

	public int getEdad() {
		return edad;
	}

	public void setEdad(int edad) {
		this.edad = edad;
	}}
```

Veamos como funciona:

```
	public static void main(String[] args) {
		Persona alumno1 = new Persona("Juan Perez",21);
		Persona alumno2 = new Persona("Vanesa Gonzalez",23);
		Persona alumno3 = new Persona("Martin Palermo",26);
		Persona alumno4 = new Persona("Sebastian Paz",30);
		Persona alumno5 = new Persona("Pepe Pillo", 22);

		Persona profesor1 = new Persona("Jacinto Dalo", 54);
		Persona profesor2 = new Persona("Guillermo Tei", 43);

		Persona director = new Persona("Dr Cito  Maximo", 65);
		Persona portero = new Persona("Don Manolo",55);

		Escuela escuela = new Escuela();
		escuela.agrega(portero);

		Direccion direccion = new Direccion();
		direccion.agrega(director);

		Aula aula1 = new Aula();
		aula1.agrega(profesor1);
		aula1.agrega(alumno1);
		aula1.agrega(alumno2);
		aula1.agrega(alumno3);

		Aula aula2 = new Aula();
		aula2.agrega(profesor2);
		aula2.agrega(alumno4);
		aula2.agrega(alumno5);

		escuela.agrega(direccion);
		escuela.agrega(aula1);
		escuela.agrega(aula2);

		escuela.getDatosPersonal();
	}
```

```
Y la salida por consola es:
Me llamo Don Manolo, tengo 55 años
Me llamo Dr Cito  Maximo, tengo 65 años
Me llamo Jacinto Dalo, tengo 54 años
Me llamo Juan Perez, tengo 21 años
Me llamo Vanesa Gonzalez, tengo 23 años
Me llamo Martin Palermo, tengo 26 años
Me llamo Guillermo Tei, tengo 43 años
Me llamo Sebastian Paz, tengo 30 años
Me llamo Pepe Pillo, tengo 22 años
```

6.4.12. Cuando utilizarlo

Este patrón busca formar un "todo" a partir de las composiciones de sus "partes". A su vez, estas "partes" pueden estar formadas de otras "partes" o de "hojas". Esta es la idea básica del Composite.

Java utiliza este patrón en su paquete de AWT (interfaces gráficas). En el paquete java.awt.swing el Componente es la clase Component, el Compuesto es la clase Container (sus Compuestos Concretos son Panel, Frame, Dialog y las hojas Label, TextField y Button.

Imaginemos que tenemos un software donde si pinta un gráfico. Algunas partes se pintan, mientras que otras partes de nuestro gráfico, de hecho, son otros gráficos y ciertas partes de estos últimos gráficos son, a su vez, otros gráficos. Este ejemplo es un caso típico para el patrón Composite.

6.4.13. Patrones relacionados

Decorator: si se usan juntos normalmente tienen una clase común padre.

Flyweight: permite compartir componentes.

Iterator: sirve para recorrer las estructuras de los componentes.

Visitor: localiza comportamientos en componentes y hojas.

6.5. Decorator Pattern

6.5.1. Introducción y nombre

Decorator, Estructural. Añade dinámicamente funcionalidad a un objeto.

6.5.2. 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.

6.5.3. También conocido como

Decorador, Wrapper (igual que el patrón Adapter).

6.5.4. 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.

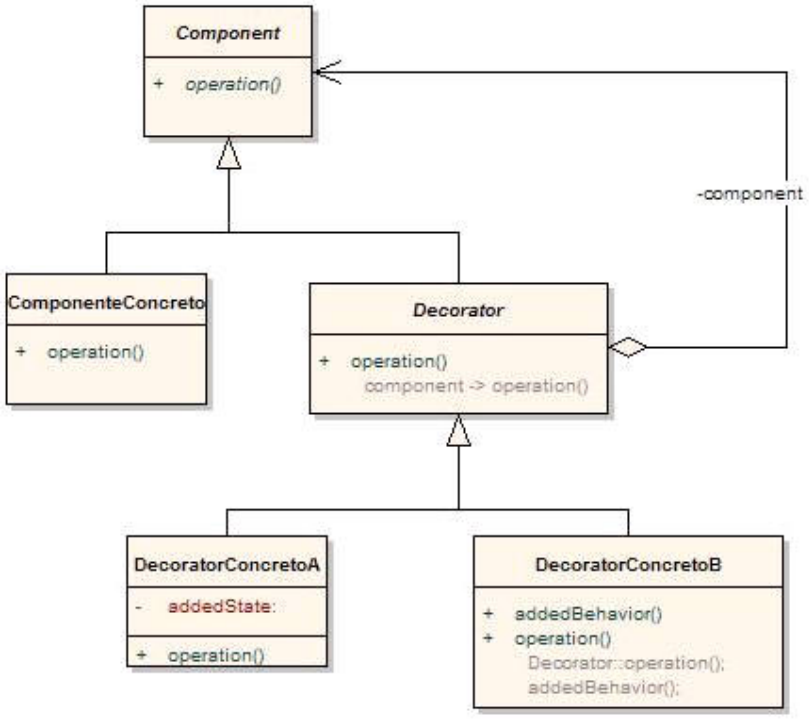
6.5.5. Solución

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.

6.5.6. Diagrama UML



6.5.7. 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.

6.5.8. Colaboraciones

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

6.5.9. 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.

6.5.10. 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.

6.5.11. 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.

```
public abstract class Auto implements Vendible{}

public class FiatUno extends Auto{

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

     public int getPrecio() {
            return 15000;}
}

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

     public int getPrecio() {
            return 25000;}
}


public abstract class AutoDecorator implements Vendible{
     private Vendible vendible;

     public AutoDecorator(Vendible vendible){
            setVendible(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);
     }
     public String getDescripcion() {
            return getVendible().getDescripcion()+ " + CD Player";
```

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


public class AireAcondicionado extends AutoDecorator {

    public AireAcondicionado(Vendible vendible) {
            super(vendible);
    }
    public String getDescripcion() {
      return getVendible().getDescripcion()+ " + Aire Acondicionado";
    }
    public int getPrecio() {
            return getVendible().getPrecio()+ 1500;
    }
}


public class Mp3Player extends AutoDecorator {

    public Mp3Player(Vendible vendible) {
            super(vendible);
    }
    public String getDescripcion() {
            return getVendible().getDescripcion()+ " + MP3 Player";
    }
    public int getPrecio() {
            return getVendible().getPrecio()+ 250;
    }
}

public class Gasoil extends AutoDecorator {

    public Gasoil(Vendible vendible) {
            super(vendible);
            }
    public String getDescripcion() {
            return getVendible().getDescripcion()+ " + Gasoil";
            }
    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
```

6.5.12. Cuando 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.

6.5.13. Patrones relacionados

Strategy; el patrón Decorator es útil para organizar las cosas que suceden antes o después de que se llaman a los métodos de otro objeto. Si se busca planificar diferentes cosas que ocurren en medio de las llamadas a un método quizás lo ideal sea utilizar el patrón Strategy.

Template Method: es otra alternativa al patrón Decorator que permite variar el comportamiento en medio de una llamada a un método en lugar de antes o después.

Composite: un decorator se puede mirar como un Composite de un solo elemento.

Adapter: un decorador solo cambia las responsabilidades del objeto, no su interface. El patrón Adapter cambia el interface. Sin embargo, ambos funcionan como "envoltorios" de objetos e incluso a ambos se los puede llamar "Wrappers".

Observemos un ejemplo:

```
public class Persona {
    private String nombre;
    private String apellido;
    private int edad;
```

```
	private String dni;
	private String domicilio;
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getApellido() {
		return apellido;
	}
	public void setApellido(String apellido) {
		this.apellido = apellido;
	}
	public int getEdad() {
		return edad;
	}
	public void setEdad(int edad) {
		this.edad = edad;
	}
	public String getDni() {
		return dni;
	}
	public void setDni(String dni) {
		this.dni = dni;
	}
	public String getDomicilio() {
		return domicilio;
	}
	public void setDomicilio(String domicilio) {
		this.domicilio = domicilio;
	}
}

public class AgenteEncubierto {
	private Persona persona;

	public AgenteEncubierto(Persona persona){
		this.persona = persona;
	}

	public String getNombre() {
		return persona.getNombre();
	}
	public void setNombre(String nombre) {
		persona.setNombre(nombre);
	}
	public String getApellido() {
		return "Perez";
	}
	public void setApellido(String apellido) {
		persona.setApellido(apellido);
	}
	public int getEdad() {
		return persona.getEdad();
	}
	public void setEdad(int edad) {
		persona.setEdad(edad);
	}
	public String getDni() {
		return "111111";
	}
	public void setDni(String dni) {
		persona.setDni(dni);
	}
```

```
        public String getDomicilio() {
                return "Av Independencia 5321";
        }
        public void setDomicilio(String domicilio) {
                persona.setDomicilio(domicilio);
                }

        public Persona getPersona() {
                return persona;
        }

        public void setPersona(Persona persona) {
                this.persona = persona;
        }
}


        public static void main(String[] args) {

                Persona persona = new Persona();
                persona.setNombre("Juan");
                persona.setApellido("Gilli");
                persona.setDni("3243232590");
                persona.setDomicilio("Av Rivadavia 423");
                persona.setEdad(43);
                System.out.println("Datos de la persona: ");
                System.out.println(persona.getApellido());
                System.out.println(persona.getNombre());
                System.out.println(persona.getDni());
                System.out.println(persona.getDomicilio());
                System.out.println(persona.getEdad());

                System.out.println("Datos del agente: ");
                AgenteEncubierto agente = new AgenteEncubierto(persona);
                System.out.println(agente.getApellido());
                System.out.println(agente.getNombre());
                System.out.println(agente.getDni());
                System.out.println(agente.getDomicilio());
                System.out.println(agente.getEdad());}
```

```
La salida por consola es:
Datos de la persona:
Gilli
Juan
3243232590
Av Rivadavia 423
43
Datos del agente:
Perez
Juan
111111
Av Independencia 5321
43
```

Como se puede observar la clase AgenteEncubierto "envuelve" a la clase Persona. Muchas veces se utiliza el patrón Decorator de esta forma y, como se puede observar, es una propuesta muy cercana al Patrón Adapter. ¿AgenteEncubierto decora o adapta a Persona? En realidad, la "envuelve".

6.6. Façade Pattern

6.6.1. Introducción y nombre

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

6.6.2. 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.

6.6.3. También conocido como

Fachada.

6.6.4. 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").

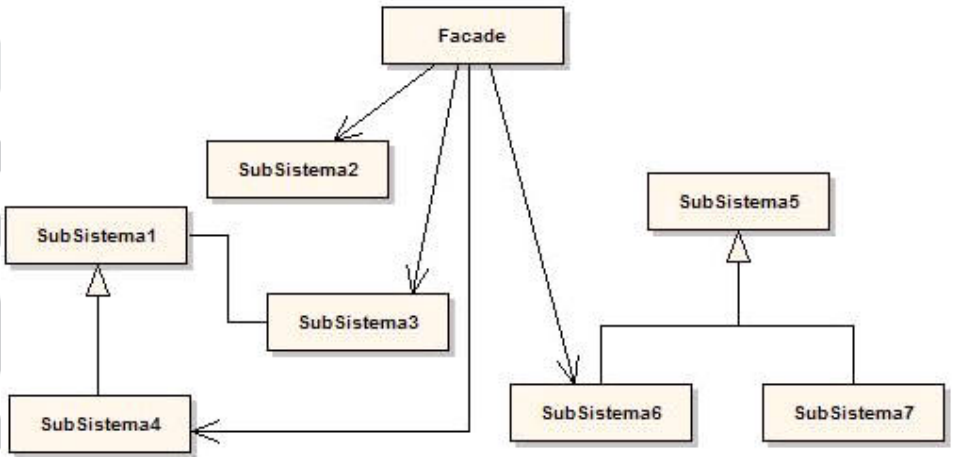
6.6.5. Solución

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.

6.6.6. Diagrama UML



6.6.7. 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.

6.6.8. 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.

6.6.9. 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.

6.6.10. 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.

```
public abstract 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 propitario");
    }
    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 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);

    }
```

6.6.11. Cuando 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.

6.6.12. Patrones relacionados

Mediator: es similar al Facade ya que éste abstrae la funcionalidad de las clases existentes.

Singleton: usualmente solo es necesario un objeto Facade, de esa manera los objetos Facade: a menudo son Singletons.

6.7. Flyweight Pattern

6.7.1. Introducción y nombre

Flyweight, Estructural. Busca reducir la cantidad de objetos en memoria.

6.7.2. Intención

Este patrón sirve para eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos que contienen información idéntica, además de lograr un equilibrio entre flexibilidad y rendimiento (uso de recursos).

6.7.3. También conocido como

Peso mosca.

6.7.4. Motivación

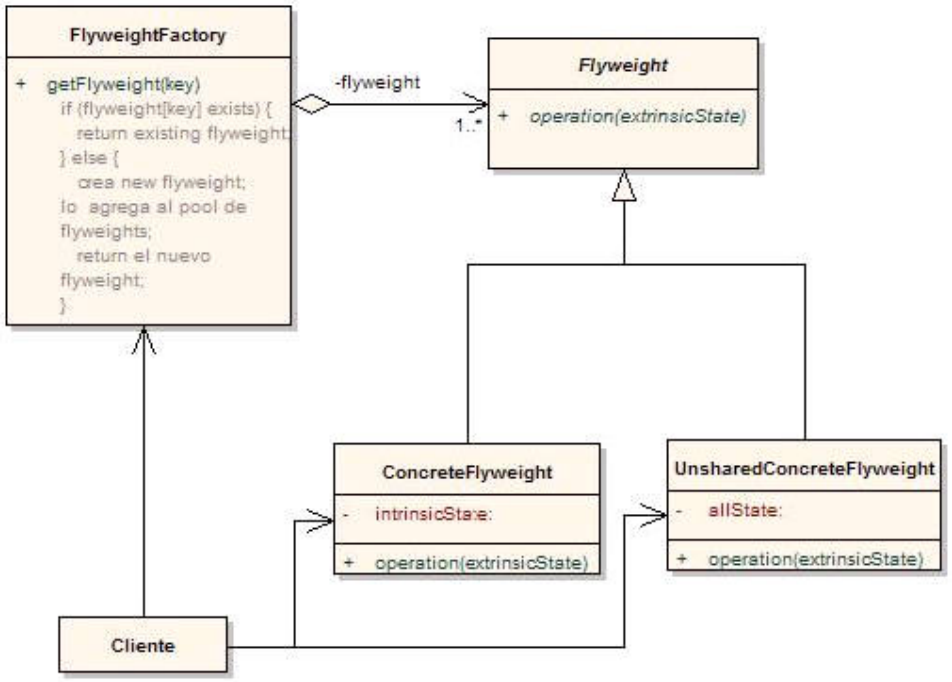
Este patrón quiere evitar el hecho de crear un gran número estados de objeto para representar a un sistema. Permite compartir estados para soportar un gran número de objetos pequeños aumentando la eficiencia en espacio.

6.7.5. Solución

Este patrón se debe utilizar cuando:

- Para eliminar o reducir la redundancia cuando se tiene gran cantidad de objetos que contienen la misma información.
- Cuando la memoria es crítica para el rendimiento de la aplicación.
- La aplicación no depende de la identidad de los objetos.

6.7.6. Diagrama UML



6.7.7. Participantes

Flyweight: declara una interfaz a través de la cual los flyweights pueden recibir y actuar sobre los estados no compartidos.

ConcreteFlyweight: implementa la interfaz Flyweight y almacena los estados compartidos, si los hay. Un objeto ConcreteFlyweight debe ser compartible. Cualquier estado que almacene debe ser intrínseco; es decir, debe ser independiente de su contexto.

UnsharedConcreteFlyweight: no todas las subclases de Flyweight tienen por qué ser compartidas. La interfaz Flyweight permite que se comparta; no lo fuerza. Es común que los objetos de esta clase tengan hijos de la clase ConcreteFlyweight en algún nivel de su estructura.

FlyweightFactory: crea y gestiona los objetos flyweight. Garantiza que los objetos flyweight se comparten de forma apropiada. Cuando un cliente solicita un flyweight, el objeto de la clase FlyweightFactory proporciona una instancia existente, o crea una.

6.7.8. Colaboraciones

Client: contiene referencias a los flyweights. Calcula o almacena los estados no compartidos de los flyweights.

6.7.9. Consecuencias

- Reduce en gran cantidad el peso de los datos en un servidor.
- Consume un poco mas de tiempo para realizar las búsquedas.
- Se complica la codificación lo que puede redundar en el aumento en la aparición de errores.
- El cliente debe tener algún conocimiento de la implementación. Por esto, puede romper con la estructura cliente-servidor.

6.7.10. Implementación

Un objeto flyweight debe ser clasificado como compartido o no compartido. Los compartidos se almacenan en el objeto ConcreteFlyweight; los no compartidos se almacenan o se calculan en el objeto Cliente. Los clientes pasan este estado al objeto flyweight cuando invocan sus operaciones.

Los clientes no deberían instanciar objetos de la clase ConcreteFlyweight directamente. Deben obtenerlos exclusivamente del objeto FlyweightFactory para garantizar que son compartidos apropiadamente.

6.7.11. Código de muestra

```
public class Alumno {
```

```
	private String nombre;
	private String apellido;
	private double promedio;
	private double promedioGeneral;

	public Alumno(double promedioGeneral){
		setPromedioGeneral(promedioGeneral);
	}

	public double compara(){
	return (((double) promedio) / promedioGeneral -1) * 100.0;
	// devuelve el porcentaje en que difiere del promedio general
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getApellido() {
		return apellido;
	}
	public void setApellido(String apellido) {
		this.apellido = apellido;
	}
	public double getPromedio() {
		return promedio;
	}
	public void setPromedio(double promedio) {
		this.promedio = promedio;
	}
	public double getPromedioGeneral() {
		return promedioGeneral;
	}
	public void setPromedioGeneral(double promedioGeneral) {
		this.promedioGeneral = promedioGeneral;
	}
}
```

Veamos como se utiliza este patrón:

```
	public static void main(String[] args) {
		double promedioDelAlumnado = 6;

		String nombres[] = {"Juan", "Maxi", "Pedro"};
		String apellidos[] = {"Perez", "Lopez", "Minna"};
		double promedios[] = {6,7,9};

		Alumno alumno = new Alumno(promedioDelAlumnado);
		for (int i = 0; i < nombres.length; i++) {
			alumno.setNombre(nombres[i]);
			alumno.setApellido(apellidos[i]);
			alumno.setPromedio(promedios[i]);
		    System.out.println(nombres[i]+ ": "+ alumno.compara());
		}
	}
```

```
La salida por consola es:
Juan: 0.0
Maxi: 16.666666666666675
Pedro: 50.0
```

6.7.12. Cuando utilizarlo

El patrón Flyweight debe utilizarse únicamente después de que un análisis de sistema determine que la economía de espacio de la memoria es crítica para el rendimiento, esto es, la memoria es un cuello de botella del programa. Al introducir estas construcciones en un programa, estamos complicando su código y aumentando las posibilidades de aparición de errores. Este patrón debe ponerse en práctica sólo en circunstancias muy limitadas.

6.7.13. Patrones relacionados

Facade: para reducir la complejidad del Flyweight si fuese necesario.

FactoryMethod: es utilizado por el Flyweight para instanciar objetos.

6.8. Proxy Pattern

6.8.1. Introducción y nombre

Proxy, Estructural. Controla el acceso a un recurso.

6.8.2. Intención

El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.

El patrón obliga que las llamadas a un objeto ocurran indirectamente a través de un objeto proxy, que actúa como un sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.

6.8.3. También conocido como

Surrogate, Virtual Proxy.

6.8.4. Motivación

Necesitamos crear objetos que consumen muchos recursos, pero no queremos instanciarlos a no ser que el cliente lo solicite o se cumplan otras condiciones determinadas.

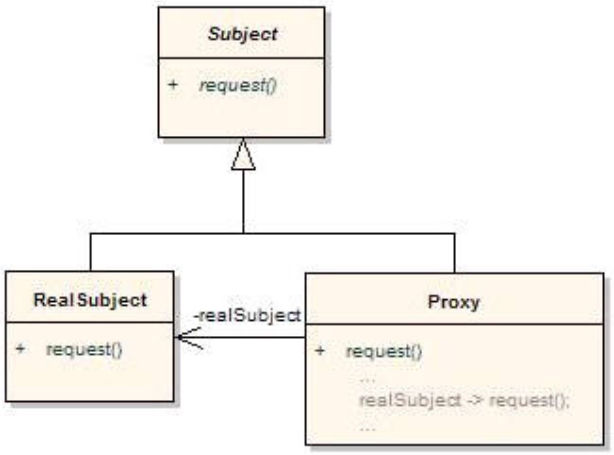
Puede haber ocasiones en que se desee posponer el coste de la creación de un objeto hasta que sea necesario usarlo.

6.8.5. Solución

Este patrón se debe utilizar cuando:

- Se necesite retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario.
- Se necesita una referencia a un objeto más flexible o sofisticada que un puntero.
- Algunas situaciones comunes de aplicación son:
- Proxy remoto: representa un objeto en otro espacio de direcciones. Esto quiere decir que el proxy será utilizado de manera tal que la conexión con el objeto remoto se realice de forma controlada sin saturar el servidor.
- Proxy virtual: crea objetos costosos por encargo. Cuando se utiliza un software no siempre se cargan todas las opciones por default. Muchas veces se habilitan ciertos módulos sólo cuando el usuario decide utilizarlos.
- Proxy de protección: controla el acceso a un objeto. Controla derechos de acceso diferentes.
- Referencia inteligente: sustituto de un puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto (ej. contar el número de referencias, cargar un objeto persistente en memoria, bloquear el objeto para impedir acceso concurrente, ...).

6.8.6. Diagrama UML



6.8.7. Participantes

Subject: interfaz o clase abstracta que proporciona un acceso común al objeto real y su representante (proxy).

Proxy: mantiene una referencia al objeto real. Controla la creación y acceso a las operaciones del objeto real.

RealSubject: define el objeto real representado por el Proxy.

6.8.8. Colaboraciones

Cliente: solicita el servicio a través del Proxy y es éste quién se comunica con el RealSubject.

6.8.9. Consecuencias

Introduce un nivel de dirección con diferentes usos:

- Un proxy remoto puede ocultar el hecho de que un objeto reside en otro espacio de direcciones.
- Un proxy virtual puede realizar optimizaciones, como la creación de objetos bajo demanda.
- Los proxies de protección y las referencias inteligentes permiten realizar tareas de mantenimiento adicionales al acceder a un objeto.
- Copiar un objeto grande puede ser costoso. Si la copia no se modifica, no es necesario incurrir en dicho gasto.

6.8.10. Implementación

Tenemos un objeto padre Subject del que heredan: RealSubject y Proxy, todos ellos tienen un método request(). El cliente llamaría al método request () de Subject, el cual pasaría la petición a Proxy, que a su vez instanciaría RealSubject y llama a su request().

Esto nos permite controlar las peticiones a RealSubject mediante el Proxy, por ejemplo instanciando RealSubject cuando sea necesario y eliminándolo cuando deje de serlo.

6.8.11. Código de muestra

Vamos a realizar un ejemplo de un proxy remoto: la finalidad es que nuestra aplicació guarde datos en un servidor remoto, pero vamos a impedir se la aplicación se conecte todo el tiempo, sino que aproveche a guardar todo cuando se encuentre conectada. Caso contrario guardará en el disco duro local la información hasta que sea el momento adecuado.

```
public class ConnectionManager {
    private static boolean hayConexion;
    // esta clase deberia ser un singleton

    public ConnectionManager(){
           hayConexion = false;
    }
    public static void conectate(){
           // se abre la conexion
           hayConexion = true;
    }

    public static void desconectate(){
           // se cierra la conexion
           hayConexion = false;
    }

    public static boolean hayConexion(){
           return hayConexion;    }
}


public interface IGuardar {
    public void save(ArrayList datosAGuardar);
}


public class GuardarDiscoDuro implements IGuardar {

    public void save(ArrayList datosAGuardar) {
    System.out.println("Guardando datos en el HD...");  }
}

public class ObjetoRemoto implements IGuardar {

    public void save(ArrayList datosAGuardar) {
    System.out.println("Guardando datos en el objeto remoto...");
    }
}


public class GuardarDatos implements IGuardar {
```

```
        private IGuardar guardarLista;

        public void save(ArrayList datosAGuardar)  {

              if (ConnectionManager.hayConexion())
                new ObjetoRemoto().save(datosAGuardar);
              else
                 new GuardarDiscoDuro().save(datosAGuardar);
           }
}

        public static void main(String[] args) {
               ArrayList datos = new ArrayList();
               datos.add("Datos a guardar!!");

               IGuardar g = new GuardarDatos();
               g.save(datos);

               ConnectionManager.conectate();
               g.save(datos);
               }
}
```

6.8.12. Cuando utilizarlo

El patrón Proxy es muy versátil. Puede ser utilizado en infinitas ocasiones y se le puede otorgar varios usos. Tiene una gran ventaja y es que no obliga al desarrollador a crear demasiada estructura para realizar este patrón, sino que es una forma estándar de acceder a una clase que potencialmente puede ser conflictiva. Por otro lado, no ayuda al desarrollador a crear un algoritmo, sino que el desarrollador tiene que hacer toda la lógica.

Por estas razones, es un patrón donde no siempre se puede saber a priori cuando utilizarlo. Sin embargo, un Proxy es un concepto utilizado fuera del ámbito de los patrones de diseño: un servidor proxy es un equipo intermediario situado entre el sistema del usuario e Internet. Puede utilizarse para registrar el uso de Internet y también para bloquear el acceso a una sede Web. El servidor de seguridad del servidor proxy bloquea algunas redes o páginas Web por diversas razones. En consecuencia, es posible que no pueda descargar el entorno de ejecución de Java (JRE) o ejecutar algunos applets de Java.

Es decir, los servidores proxy funcionan como filtros de contenidos. Y también mejoran el rendimiento: guardan en la memoria caché las páginas Web a las que acceden los sistemas de la red durante un cierto tiempo. Cuando un sistema solicita la misma página web, el servidor proxy utiliza la información guardada en la memoria caché en lugar de recuperarla del proveedor de contenidos. De esta forma, se accede con más rapidez.

Este mismo concepto se intenta llevarlo a cabo a nivel código con el patrón Proxy. Cuando un objeto debe ser controlado de alguna manera, ya sea por simple control de acceso o por estar en un sitio remoto o por ser muy pesado y se quiera limitar su uso, es ideal utilizar este patrón.

6.8.13. Patrones relacionados

Facade: en ciertos casos puede resultar muy similar al facade, pero con mayor inteligencia.

Decorator: ciertas implementaciones se asemejan al decorador.

Singleton: la mayoría de las clases que hacen de proxy son Singletons.

7. Los Anti-Patrones

7.1. Objetivos

7.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

7.2. Anti-patrón

Un antipatrón de diseño es un patrón de diseño que conduce a una mala solución. Buscan evitar que los programadores tomen malas decisiones, partiendo de documentación disponible en lugar de simplemente la intuición.

7.3. Historia

El término nace con el mencionado libro Design Patterns de GOF, que básicamente son los patrones que vimos a lo largo de este curso. Los autores bautizaron dichas soluciones con el nombre de "patrones de diseño" por analogía con el mismo término, usado en arquitectura.

El libro Anti-Patterns (de William Brown, Raphael Malveau, Skip McCormick, Tom Mowbray y Scott Thomas) describe los antipatrones como la contrapartida natural al estudio de los patrones de diseño. El término fue utilizado por primera vez en 1995, por ello es que los antipatrones no se mencionan en el libro original de Design Patterns, puesto que éste es anterior.

7.4. Propósito

Los anti-patrones son ejemplos bien documentados de malas soluciones para problemas. El estudio formal de errores que se repiten permite que el desarrollador pueda reconocerlos más fácilmente y, por ello, podrá encaminarse hacia una mejor solución. Los desarrolladores deben evitar los antipatrones siempre que sea posible.

7.5. Utilización

Dado que el término antipatrón es muy amplio, suele ser utilizado en diversos contextos:

7.5.1. Antipatrones de desarrollo de software

1 Antipatrones de gestión

2 Antipatrones de gestión de proyectos

3 Antipatrones generales de diseño de software

    3.1 Antipatrones de diseño orientado a objetos

4 Antipatrones de programación

5 Antipatrones metodológicos

6 Antipatrones de gestión de la configuración

7.5.2. Antipatrones organizacionales

- Avance del alcance (scope creep): Permitir que el alcance de un proyecto crezca sin el control adecuado.
- Bloqueo del vendedor (vendor lock-in): Construir un sistema que dependa en exceso de un componente proporcionado por un tercero.
- Diseño de comité (design by committee): Contar con muchas opiniones sobre un diseño, pero adolecer de falta de una visión unificada.
- Escalada del compromiso (escalation of commitment): No ser capaz de revocar una decisión a la vista de que no ha sido acertada.
- Funcionalitis acechante (creeping featuritis): Añadir nuevas funcionalidades al sistema en detrimento de su calidad.

- Gestión basada en números (management by numbers): Prestar demasiada atención a criterios de gestión cuantitativos, cuando no son esenciales o difíciles de cumplir.

- Gestión champiñón (mushroom management): Tratar a los empleados sin miramientos, sin informarles de las decisiones que les afectan (manteniéndolos cubiertos y en la oscuridad, como los champiñones).

- Gestión por que lo digo yo (management by perkele): Aplicar una gestión autoritaria con tolerancia nula ante las disensiones.

- Migración de coste (cost migration): Trasladar los gastos de un proyecto a un departamento o socio de negocio vulnerable.

- Obsolescencia continua (continuous obsolescence): Destinar desproporcionados esfuerzos a adaptar un sistema a nuevos entornos.

- Organización de cuerda de violín (violin string organization): Mantener una organización afinada y en buen estado, pero sin ninguna flexibilidad.

- Parálisis del análisis (analysis paralysis): Dedicar esfuerzos desproporcionados a la fase de análisis de un proyecto, eternizando el proceso de diseño iterando sobre la búsqueda de mejores soluciones o variantes.

- Peligro moral (moral hazard): Aislar a quien ha tomado una decisión a raíz de las consecuencias de la misma.

- Sistema de cañerías (stovepipe): Tener una organización estructurada de manera que favorece el flujo de información vertical, pero inhibe la comunicación horizontal.

- Te lo dije (I told you so): Permitir que la atención se centre en que la desoída advertencia de un experto se ha demostrado justificada.

- Vaca del dinero (cash cow): Pecar de autocomplacencia frente a nuevos productos por disponer de un producto legacy muy lucrativo.

7.6. Otros Patrones

7.6.1. Introduccion

Dado el éxito que tuvo el famoso libre de Gof, comenzó una serie de epidemia de patrones. Hoy en día hay patrones de todo tipo que busca explicar las mejores prácticas de cada caso.

Los más conocidos son los siguientes.

7.6.2. Patrones de base de datos

Buscan abstraer y encapsular todos los accesos a la fuente de datos. El más conocido es el DAO. Técnicamente el patrón DAO (Data Access Object) pertenece a los patrones JEE, aunque se centra específicamente en la capa de persistencia de los datos.

7.6.3. Patrones de Arquitectura

Se centran en la construcción del software, sobretodo en la ingeniería del mismo. El más conocido es el MVC, que busca separar el diseño de la lógica del negocio.

7.6.4. Patrones JEE

Es un catálogo de patrones para usar en la tecnología JEE, es decir, en la parte empresarial que ofrece Java. Divide los patrones en 3 categorias: Capa de Presentación, de Negocios y de Integración. En esta última capa se encuentra el DAO.

7.6.5. Patrones de AJAX

Ajax es una tecnología relativamente nueva que trabaja en la capa de diseño de una aplicación web. Pertenece al grupo de tecnologías web 2.0 y se ha convertido en un estándar mundial. Ya existen patrones de diseño de AJAX, como por ejemplo el patrón Process Indicator: su idea es, que cuando se está realizando alguna acción en Ajax y el usuario no percibe ningún tipo de actividad, hay que informarle de una forma alternativa de que se está realizando la petición que ha solicitado.

8. Laboratorios

8.1. Objetivos

8.1.1. Mapa del Curso

1. INTRODUCCIÓN
2. TIPOS DE PATRONES
3. UML REVIEW
4. PATRONES DE COMPORTAMIENTO
5. PATRONES DE CREACIÓN
6. PATRONES DE ESTRUCTURA
7. LOS ANTI-PATRONES
8. LABORATORIOS

8.2. Consignas Generales

Para todos los ejercicios:

- Respete el encapsulamiento.
- Utilice las fuentes del ejercicio anterior

8.3. Laboratorio 1

Realice una clase llamada Banco y aplique a dicha clase el patrón Singleton. Agregue los siguientes atributos: String nombre, String calle, int numero, String telefono.

Instancie dichos atributos en el constructor con los siguientes valores:

```
nombre= "Banco IT";
telefono = "4328-0457";
calle = "Lavalle";
numero= 648;
```

8.4. Laboratorio 2

El banco posee varias sucursales y utilizaremos los patrones de diseño para la creación de las mismas. Comenzaremos por las oficinas que serán parte de las sucursales:

Hay 3 tipos de Oficinas:

a) Oficina Standard, sin ventanas, con 4 escritorios y entran hasta 4 personas.

b) Oficina Gerencial, con 1 ventana, 2 escritorios y entran hasta 2 personas ya que esta pensada para el gerente y una secretaria.

c) Oficina de Seguridad, con 1 ventana, entran hasta 3 personas y tienen 1 escritorio.

Haga una clase llamada Oficina, con los siguientes atributos:

int cantidadEscritorios, boolean tiene ventana, int cantidadMaximaPersonas.

Para crear los tres tipos de Oficina, utilice el patrón Prototype.

8.5. Laboratorio 3

Para crear las sucursales se decide seguir la siguiente estructura. Hay 3 tipos de sucursales:

A) Sucursal Capital:

Oficinas gerenciales: 3.

Oficinas de seguridad: 2.

Oficinas estándar: 6.

Cantidad de mostradores para atención al público: 3.

Cantidad de cajeros automáticos disponibles: 4.

B) Sucursal Gran Bs As:

Oficinas gerenciales: 2.

Oficinas de seguridad: 3.

Oficinas estándar: 5.

Cantidad de mostradores para atención al público: 2.

Cantidad de cajeros automáticos disponibles: 2.

C) Sucursal Interior:

Oficinas gerenciales: 1.

Oficinas de seguridad: 1.

Oficinas estándar: 4.

Cantidad de mostradores para atención al público: 2.

Cantidad de cajeros automáticos disponibles: 1.

Para esto Ud. debe crear la clase Sucursal con los siguientes atributos:

```
private int cantidadMostradores;

private int cantidadCajeros;

private ArrayList <Oficina> oficinasGerenciales;

private ArrayList <Oficina> oficinasSeguridad;

private ArrayList <Oficina> oficinasStandard;
```

En la clase Banco deberá adicionar un atributo: ArrayList <Sucursal> sucursales.

Utilice el patrón Builder para crear los 3 tipos de sucursales.

8.6. Laboratorio 4

Utilice las fuentes del ejercicio Patrones-Ej4-Base para realizar este ejercicio.

Inspeccione las fuentes y verá que un cliente puede tener 2 tipos de cuentas: caja de ahorro o cuenta corriente, ambas pueden ser en pesos o dólares.

Utilice el patrón FactoryMethod para crear de una manera sencilla los distintos tipos de cuentas que puede obtener un cliente.

8.7. Laboratorio 5

El banco se encuentra con el siguiente problema: una tarjeta de crédito puede tener 3 estados posibles: En Rojo, Normal, Inhabilitada.

Cuando un cliente quiera pagar con la tarjeta podrá hacerlo siempre y cuando el estado sea Normal, ya que si el estado fuese En Rojo, el límite es de $ 1000 para retirar. En cambio, si la tarjeta estuviese inhabilitada no puede realizar el retiro del dinero.

Si busca establecer en la clase TarjetaDeCredito un método con la siguiente firma:

public boolean puedeRetirar(int monto)

Resuelva este problema con el patrón State. Ud debe crear la clase TarjetaDeCredito y todas aquellas clases que crea correspondiente.

8.8. Laboratorio 6

El Banco desea resolver el problema con los clientes morosos y para ello quiere informarse cada vez que una tarjeta de crédito es inhabilitada.

El banco debe ser informado con el número de y además se deberá crear un log (un archivo de texto) de manera automática con el numero de tarjeta y la fecha.

Utilice el patrón Observer para resolver este problema.

8.9. Laboratorio 7

Se busca resolver el problema del cálculo de los intereses bancarios tanto para las tarjetas de crédito como para las cuentas que puedan tener los clientes en el Banco.

Para ello el banco cobra un 5% de aumento mensual en el interés de la tarjeta de crédito y descuenta un 1% mensual sobre el monto total de la cuenta corriente y caja de ahorro.

Utilice el patrón Visitor para resolver este problema.

8.10. Laboratorio 8

Busque una clase representativa para realizar el patrón. Este ejercicio será discutido en clase.

8.11. Conclusión

Los patrones de diseño son muy útiles. Es por ello que se los conoce como las mejores prácticas en el desarrollo y construcción de software.

Este curso tiene como objetivo servir como punto de inicio para introducirse en el mundo de los patrones de diseño.

El objetivo práctico es dominar las diferentes estrategias de diseño para lograr diseñar sistemas de forma profesional, maximizando la reutilización y minimizando el mantenimiento.

8.12. Links de referencia

http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o

http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

http://www.programacion.net/java/tutorial/patrones/

http://users.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/

http://www.javacamp.org/designPattern/