

**UNIVERSIDAD VERACRUZANA**

**FACULTAD DE ESTADÍSTICA E INFORMÁTICA**

**XALAPA, VERACRUZ**

**PROGRAMA EDUCATIVO**

**Informática**

**EXPERIENCIA EDUCATIVA**

**Tópicos selectos de computación I**

**DOCENTE**

**Montané Jiménez Luis Gerardo**

**TRABAJO**

**Patrones de diseño de comportamiento**

**INTEGRANTES**

**Hernández Cruz Lázaro**

**Panamá Miranda Luis David**

**Pérez Domínguez Noé Alejandro**

**Rendón Martínez Cristian Alberto**

**Rojas Pérez Daniel Alberto**

**Xalapa, Ver., 15 de febrero de 2017**

Tabla de contenido

[Patrones de diseño de comportamiento 1](#_Toc475433819)

[1. Chain of responsability 2](#_Toc475433820)

[2. Command 9](#_Toc475433821)

[3. Interpreter 18](#_Toc475433822)

[4. Iterator 25](#_Toc475433823)

[5. Mediator 30](#_Toc475433824)

[6. Memento 36](#_Toc475433825)

[7. Observer 40](#_Toc475433826)

[8. State 46](#_Toc475433827)

[9. Strategy 52](#_Toc475433828)

[10. Template method 58](#_Toc475433829)

[11. Visitor 63](#_Toc475433830)

[Bibliografía 68](#_Toc475433831)

# Patrones de diseño de comportamiento

Los patrones de diseño de comportamiento son aquellos que están relacionados con algoritmos y con la asignación de responsabilidades a los objetos.

Describen no solamente patrones de objetos o de clases, sino que también engloban patrones de comunicación entre ellos. Al igual que los otros tipos de patrones, se pueden clasificar en función de que trabajen con clases (Template Method, Interpreter) u objetos (Chain of Responsability, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor).

La variación de la encapsulación es la base de muchos patrones de comportamiento. Cuando un aspecto de un programa cambia frecuentemente, estos patrones trabajan con un objeto que encapsula dicho aspecto, teniendo que definir, por tanto, una clase abstracta que describe la encapsulación del objeto.

## Chain of responsability

Es un patrón de comportamiento ya que se caracteriza principalmente de comportamiento que existe entre clases y objetos en un sistema.

Es un patrón que es utilizado más comúnmente cuando se necesita realizar una cadena de objetos receptores a los cuales se les pasa una petición de un objeto emisor y estos la procesan de manera diferente dependiendo de un criterio antes establecido, esto ayuda a generar un bajo acoplamiento entre clases y objetos lo cual sabemos que es algo bueno para desarrollo.

**Problema que pretende resolver.**

Este patrón pretende solucionar problemas con implementaciones complicadas y condigo rebuscado, como por ejemplo la implementación de if-else largos en el código de manera que se complica la implementación. Por lo que mantienen un bajo acoplamiento ya que mantiene al "cliente" con desconocimiento de lo que pueda generar nuestro "servidor"

**Causas**

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

**Solución.**

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.

Estructura - UML

Cliente 
HandlerConcreto 1 
handleRequest0 
Handler 
handleReguestO 
HandlerConcret02 
handleRequest() 

**Participantes.**

• Manejador: define una interfaz para tratar las peticiones. Opcionalmente, implementa el enlace al sucesor.

• ManejadorConcreto: trata las peticiones de las que es responsable; si el ManejadorConcreto puede manejar la petición, lo hace; en caso contrario la reenvía a su sucesor.

• Cliente: inicializa la petición a un Manejador Concreto de la cadena.

**Consecuencias**

Este patrón se aplica en ocasiones con el patrón Composición. En él, los padres de los componentes pueden actuar como sucesores.

**Patrones relacionados**

Este patrón se aplica en ocasiones con el patrón Composición. En él, los padres de los componentes pueden actuar como sucesores.

**Implementación.**

Esta es la clase de la cual van a heredar e implementar el metodo abstract.

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **Chain**

{

**abstract** **class** **Approver**

{

**protected** Approver successor;

**public** **void** SetSuccessor(Approver successor)

{

**this**.successor = successor;

}

**public** **abstract** **void** ProcessRequest(Purchase p);

}

}

Para el ejemplo que vamos a ocupar, el segundo método ocupa una clase llamada Purchase.

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **Chain**

{

**class** **Purchase**

{

**private** int \_number;

**private** double \_amount;

**private** string \_purpose;

**public** Purchase(int number, double amount, string purpose)

{

**this**.\_amount = amount;

**this**.\_number = number;

**this**.\_purpose = purpose;

}

*//Sets y gets de cada atributo.*

**public** int Number { **get** { **return** \_number; } **set** { \_number = **value**; } }

**public** double Amount{ **get** { **return** \_amount; } **set** { \_amount = **value**; } }

**public** string Purpuse { **get** { **return** \_purpose; } **set** { \_purpose = **value**; } }

}

}

Para formar la cadena, vamos a ocupar algunas clases.

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **Chain**

{

**class** **VicePresident** : Approver

{

**public** **override** **void** ProcessRequest(Purchase purchase)

{

**if** (purchase.Amount < 25000.0)

{

Console.WriteLine("Peticion aceptada ->",

**this**.GetType().Name, purchase.Number);

}

**else** if (successor != **null**)

{

successor.ProcessRequest(purchase);

}

}

}

}

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **Chain**

{

**class** **Director** : Approver

{

**public** **override** **void** ProcessRequest(Purchase purchase)

{

**if** (purchase.Amount < 10000.0)

{

Console.WriteLine("{0} approved request# {1}",

**this**.GetType().Name, purchase.Number);

}

**else** if (successor != **null**)

{

successor.ProcessRequest(purchase);

}

}

}

}

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **Chain**

{

**class** **President** : Approver

{

**public** **override** **void** ProcessRequest(Purchase purchase)

{

**if** (purchase.Amount < 100000.0)

{

Console.WriteLine("{0} approved request# {1}",

**this**.GetType().Name, purchase.Number);

}

**else**

{

Console.WriteLine(

"Request# {0} requires an executive meeting!",

purchase.Number);

}

}

}

}

Las clases anteriores como sabemos, trataban las peticiones de manera diferente. En el main del sistema vamos a probar lo que sucede.

**using** **System**;

**using** **System.Collections.Generic**;

**using** **System.Linq**;

**using** **System.Text**;

**using** **System.Threading.Tasks**;

**namespace** **Chain**

{

**class** **Program**

{

**static** **void** Main(string[] args)

{

Approver larry = **new** Director();

Approver sam = **new** VicePresident();

Approver tammy = **new** President();

larry.SetSuccessor(sam);

sam.SetSuccessor(tammy);

Purchase p = **new** Purchase(2034, 350.00, "Assets");

larry.ProcessRequest(p);

p = **new** Purchase(2035, 32590.10, "Project X");

larry.ProcessRequest(p);

p = **new** Purchase(2036, 122100.00, "Project Y");

larry.ProcessRequest(p);

*// Wait for user*

Console.ReadKey();

}

}

}

## Command

**Sinopsis**:

“Encapsular una petición como un objeto, de modo que puedan parametrizarse otros objetos con distintas peticiones o colas de peticiones y proporcionar soporte para realizar operaciones que puedan deshacerse”.

**Contexto**:

La solución anterior es apropiada cuando lo fundamental en la relación entre una petición y la acción que la satisface es la FLEXIBILIDAD. Este patrón, al encapsular en un objeto la acción, promueve una separación entre dicha acción y la petición que resuelve, lo cual redunda en una mayor flexibilidad en todo lo relativo a la acción.

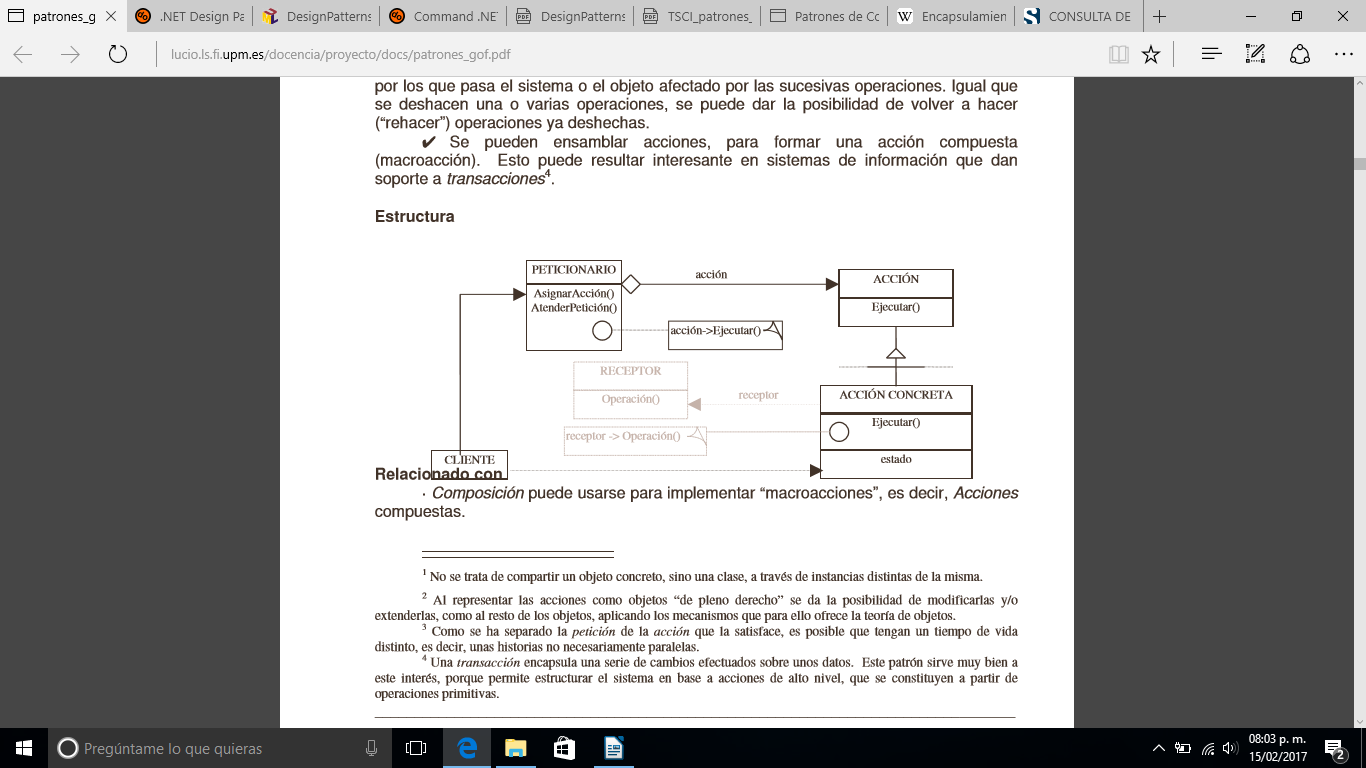
**Causas**:

Pretende principalmente proporcionar soporte para deshacer operaciones. ¿Cómo realizar esto? Dado que estas operaciones se encapsulan como objetos, sería perfectamente posible mantener en memoria una pila con elementos ejecutados y otra que almacene objetos que modelen los distintos estados correspondientes a los instantes anteriores a ejecutar cada uno de los Commands de la primera pila. De este modo, realizar la opción deshacer sería tan sencillo como realizar un pop del primer objeto de cada pila y sobre escribir el estado actual del programa con el contenido del primer objeto almacenado en la pila de estados.

Dado que este patrón nos permite encapsular la ejecución de un método como un objeto, otra de las funcionalidades que nos ofrece este patrón es la de encadenar invocaciones. Si creamos una serie de objetos de este tipo y los almacenamos como objetos, éstos pueden pasarse como parámetros a un método que los vaya ejecutando de forma secuencial.

Asistentes de instalación (cada pantalla implementaría un Command que realizaría una operación Execute() al pulsar el botón “Siguiente”)

**Solución**:

Encapsular en un objeto la acción que satisface una petición.

**Consecuencias:**

* Diferentes objetos pueden ejecutar la misma acción sin necesidad de repetir su declaración e implementación. Basta con que la “compartan”.
* Se puede cambiar con facilidad (incluso dinámicamente) la acción que realiza o está asociada a un objeto.
* Se pueden añadir nuevas acciones sin tocar las clases ya existentes.
* Se puede especificar, encolar y ejecutar una acción en momentos diferentes.
* Se puede contemplar la posibilidad de deshacer una operación. Para ello es necesario disponer de una “lista de históricos” en la que se reflejen los diferentes estados por los que pasa el sistema o el objeto afectado por las sucesivas operaciones. Igual que se deshacen una o varias operaciones, se puede dar la posibilidad de volver a hacer (“rehacer”) operaciones ya deshechas.
* Se pueden ensamblar acciones, para formar una acción compuesta (macroacción). Esto puede resultar interesante en sistemas de información que dan soporte a transacciones.

**Implementación:**

Se nos ha encargado programar parte de la centralita de un nuevo modelo, concretamente de la activación y desactivación de las luces. Sabemos que contaremos con tres tipos de luces: posición, cortas y largas, pero las implementaciones de los métodos encargados de este proceso aún no están definidas, ya que se han subcontratado a una empresa coreana. Sin embargo, nuestro cliente nos exige que, pese a ello, debemos avanzar con el diseño de este módulo.

El patrón Command es perfecto en este escenario: contamos con un entorno en el que debemos ejecutar un método que aún no conocemos o que puede cambiar con el tiempo, por lo que crearemos una interfaz ICommand que exponga un método Execute(). Posteriormente, una vez que los requisitos estén claros y que sepamos qué hay que ejecutar, implementaremos clases que se acoplen entre nuestra centralita y el componente que la empresa coreana nos proporcionará.

Por lo tanto, comenzaremos creando una interfaz ICommand que se encargará de ofrecer una firma para el método que encapsulará el resto de los métodos: Execute():

public interface ICommand

{

    // El método Execute() será aquel que el objeto que reciba la referencia

    // será capaz de ejecutar.

    void Execute();

}

A continuación modelaremos las clases que serán encapsuladas por el objeto Command. O más específicamente, las clases cuyos métodos serán encapsulados por el método Execute() del objecto Command(). Crearemos una clase abstracta de la que heredarán el resto de las clases que serán encapsuladas, aunque en realidad esto no es estrictamente necesario (un objeto Command podría albergar diversos objetos no necesariamente relacionados entre sí).

Esta clase es conocida como Receiver, y se caracteriza porque se encargará de albergar la lógica concreta del método. Si nos ceñimos a la descripción de una interfaz, la clase Receiver sería aquella clase que se encargaría de implementarla.

LucesReceiver.cs

// Clase abstracta de la que heredarán las clases que serán encapsuladas por los

// objectos Command. Por lo tanto, sus métodos serán aquellos que encapsulará el

// método Execute().

public abstract class LucesReceiver

{

    protected bool encendidas;

    protected int distanciaAlumbrado;

    // Propiedad de sólo lectura que devolverá el estado de las luces

    public bool Encendidas

    {

        get {return encendidas;}

    }

    // Método encargado de apagar las luces. Establece el estado del atributo 'encendidas'

    // a 'false'. Será común a todas las clases que hereden de ella.

    public void Apagar()

    {

        this.encendidas = false;

    }

    // El método Encender será distinto en cada una de las clases que hereden de esta clase.

    public abstract int Encender();

}

A continuación implementaremos las clases concretas cuyos métodos serán encapsulados por ICommand.Execute(). Dado que heredan de la clase abstracta LucesReceiver, únicamente tendremos que implementar de forma explícita el método Encender(), ya que el resto de la funcionalidad será común. Comenzaremos por las luces de posición:

LucesPosicion.cs

public class LucesPosicion : LucesReceiver

{

    private const int DISTANCIA = 1;

    public override int Encender()

    {

        this.encendidas = true;

        return DISTANCIA;

    }

}

A continuación hacemos lo mismo con las luces cortas.

LucesCortas.cs

public class LucesCortas : LucesReceiver

{

    private const int DISTANCIA = 40;

    public override int Encender()

    {

        this.encendidas = true;

        return DISTANCIA;

    }

}

Finalmente, realizamos el mismo proceso con las luces largas.

LucesLargas.cs

public class LucesLargas : LucesReceiver

{

    private const int DISTANCIA = 200;

    public override int Encender()

    {

        this.encendidas = true;

        return DISTANCIA;

    }

}

Ya tenemos implementado uno de los elementos de nuestro patrón. Ahora haremos lo propio con la parte cliente, es decir, nuestra centralita. O más concretamente, el módulo encargado de encender y apagar las luces. Este se corresponderá con el elemento Invoker, que es aquel que tendrá como objetivo invocar la acción encapsulada por el objeto Command. Comenzaremos implementando una interfaz que ofrezca dos operaciones:

* Una operación SetCommand(ICommand) que permitirá a nuestro módulo cambiar el ICommand a ejecutar.
* Una operación Invoke() que invoque el método ICommand.Execute() que esté asignado en el momento actual.

IInvoker.cs

public interface IInvoker

{

    // Inyecta un ICommand, permitiendo cambiar la operación encapsulada en

    // tiempo de ejecución

    void SetCommand(ICommand command);

    // Ejecuta el método encapsulado

    void Invoke();

}

A continuación implementaremos la interfaz. Lo haremos a través de la clase ControladorLucesInvoker. Nuestra clase realizará justo aquello que acabamos de definir en la interfaz. Además, incluirá una referencia a un objeto que implemente la interfaz ICommand

public class ControladorLucesInvoker : IInvoker

{

    ICommand command;

    public void SetCommand(ICommand command)

    {

        this.command = command;

    }

    public void Invoke()

    {

        command.Execute();

    }

}

Ya casi hemos completado nuestro patrón. Falta lo más importante: las clases que implementan la clase ICommand y que simbolizan las operaciones que se quieren encapsular. Estas operaciones serán, como podemos imaginar, encender y apagar. Por lo tanto, crearemos las clases EncenderLucesCommand

y ApagarLucesCommand:

EncenderLucesCommand.cs

public class EncenderLucesCommand : ICommand

{

    // Referencia al objeto cuyo método se quiere encapsular

    private LucesReceiver luces;

    // El constructor inyectará el objeto cuyo método se quiere encapsular

    public EncenderLucesCommand(LucesReceiver luces)

    {

        this.luces = luces;

    }

    // El método Execute() ejecutará el método encapsulado

    public void Execute()

    {

        int distancia = luces.Encender();

        Console.WriteLine(string.Format("Encendiendo luces. Alumbrando a una distancia de {0} metros.", distancia));

    }

}

La clase ApagarLucesCommand realizará la operación contraria.

ApagarLucesCommand.cs

public class ApagarLucesCommand : ICommand

{

    private LucesReceiver luces;

    public ApagarLucesCommand(LucesReceiver luces)

    {

        this.luces = luces;

    }

    public void Execute()

    {

        luces.Apagar();

        Console.WriteLine("Apagando las luces");

    }

}

Finalmente, añadimos el código que hará uso del patrón que acabamos de implementar. En él crearemos los tres tipos de luces (posición, cortas y largas) y utilizaremos el método SetCommand() de cada uno de los objetos destinados a encender y a apagar las luces para cambiar en tiempo de ejecución el tipo de luz a manejar.

// Instanciamos los objetos cuyos métodos serán encapsulados dentro de

// objetos que implementan ICommand

LucesReceiver lucesPosicion = new LucesPosicion();

LucesReceiver lucesCortas = new LucesCortas();

LucesReceiver lucesLargas = new LucesLargas();

// Creamos los objetos Command que encapsulan los métodos de las clases anteriores

ICommand encenderPosicion = new EncenderLucesCommand(lucesPosicion);

ICommand apagarPosicion = new ApagarLucesCommand(lucesPosicion);

ICommand encenderCortas = new EncenderLucesCommand(lucesCortas);

ICommand apagarCortas = new ApagarLucesCommand(lucesCortas);

ICommand encenderLargas = new EncenderLucesCommand(lucesLargas);

ICommand apagarLargas = new ApagarLucesCommand(lucesLargas);

// Creamos un nuevo Invoker (el objeto que será desacoplado de las luces)

IInvoker invoker = new ControladorLucesInvoker();

// Le asociamos los objetos Command y los ejecutamos

// El objeto invoker invoca el método 'Execute()' sin saber en ningún momento

// qué es lo que se está ejecutando realmente.

invoker.SetCommand(encenderPosicion);      // Asociamos el ICommand

invoker.Invoke();                          // Hacemos que se ejecute ICommand.Execute()

// Realizamos lo mismo con el resto de instancias que implementan ICommand.

// Como podemos ver, el ICommand puede asignarse en tiempo de ejecucion

invoker.SetCommand(apagarPosicion);

invoker.Invoke();

// Luces cortas

invoker.SetCommand(encenderCortas);

invoker.Invoke();

invoker.SetCommand(apagarCortas);

invoker.Invoke();

// Luces largas

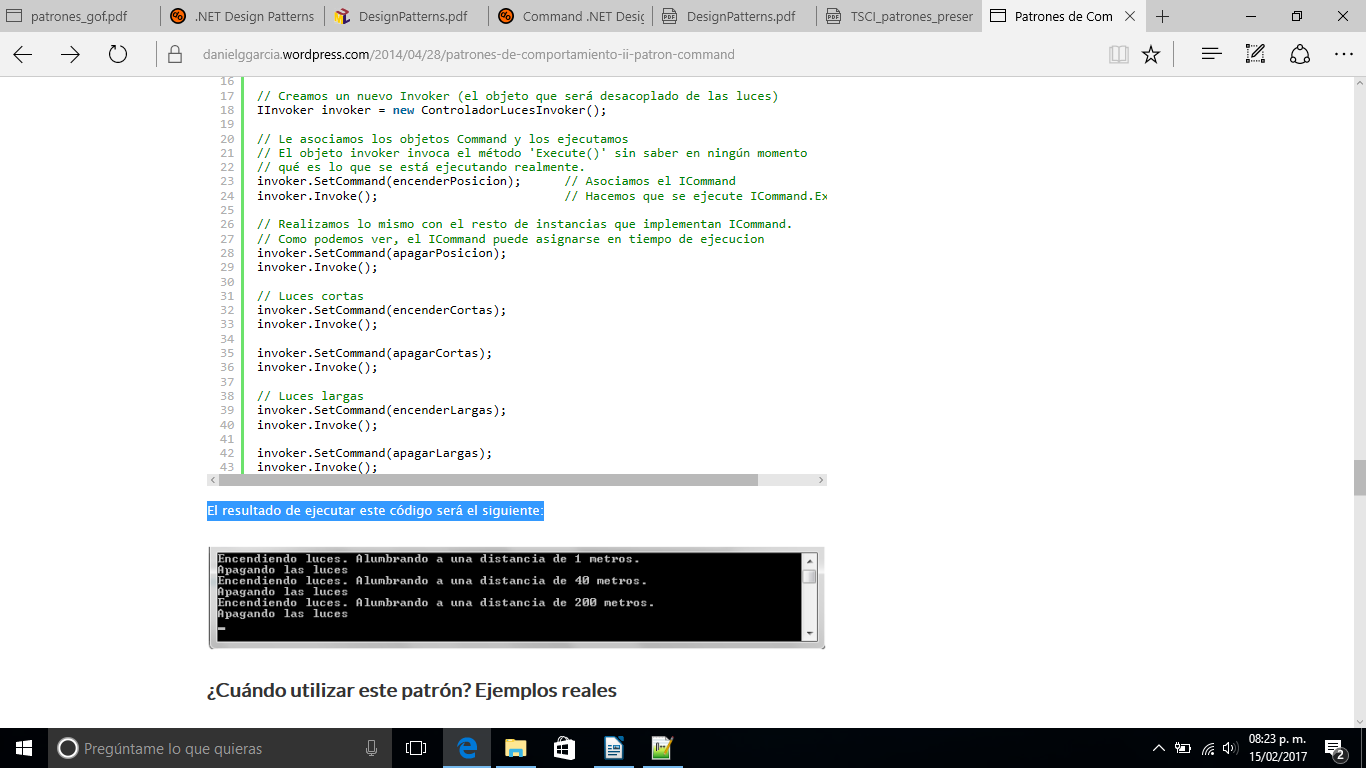
invoker.SetCommand(encenderLargas);

invoker.Invoke();

invoker.SetCommand(apagarLargas);

invoker.Invoke();

El resultado de ejecutar este código será el siguiente:



**Patrones relacionados:**

Memento.

## Interpreter

El “interpreter” es un patrón de diseño que, dado un lenguaje, define una representación para su gramática junto con un intérprete del lenguaje.

**Problema que pretende resolver.**

Si un problema es lo suficientemente común, entonces valdría la pena expresar instancias del problema como sentencias de un lenguaje simple. Después se puede construir in intérprete que resuelve el problema interpretando el lenguaje.

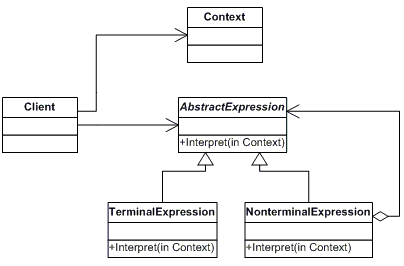
**Causas**

* Principalmente, cuando haya un lenguaje que interpretar y sus diferentes construcciones puedan representarse mediante árboles sintácticos abstractos.

**Solución**

Se describe como definir una gramática para un lenguaje sencillo, representa sentencias del lenguaje e interpreta las sentencias.

**Estructura UML**



**Participantes**

* AbstractExpression (Expression):
  + Daclara una interfaz para ejecutar una operación.
* TerminalExpression:
  + Implementa una operación del intérprete asociada con los símbolos terminales de la gramática.
  + Una instancia es requerida por cada símbolo terminal en la sentencia.
* NonterminalExpression:
  + Implementa una operación del intérprete para símbolos no terminales de la gramática.
  + Mantiene instancias de variables de tipo AbstractExpression por cada símbolo de la gramática.
* Context:
  + Contiene información que es global para el intérprete.
* Client:
  + Construye, o es dado un árbol sintáctico abstracto representando una sentencia en particular del lenguaje que la gramática define. El árbol sintáctico abstracto se construye a partir las expresiones no terminales y las expresiones terminales.
  + Invoca la operación interpret.

**Consecuencias:**

* Permite separar la representación de una computación de la manera en la que se ejecuta.
* Cualquier problema, en última instancia, se puede resolver usando intérpretes.

**Implementación:**

**package** **interpreter**

**import** **java.util**

**import** **scala.io.StdIn**

**trait** **Expression** {

**def** interpret()**:** Int

}

**case** **class** **Add**(leftExpression**:** Expression,

rightExpression**:** Expression) **extends** **Expression** {

**override** **def** interpret()**:** Int = leftExpression.interpret() + rightExpression.interpret()

}

**case** **class** **Product**(leftExpression**:** Expression,

rightExpression**:** Expression) **extends** **Expression** {

**override** **def** interpret()**:** Int = leftExpression.interpret() \* rightExpression.interpret()

}

**case** **class** **Substract**(leftExpression**:** Expression,

rightExpression**:** Expression) **extends** **Expression** {

**override** **def** interpret()**:** Int = leftExpression.interpret() - rightExpression.interpret()

}

**case** **class** **Number**(n**:** Int) **extends** **Expression** {

**override** **def** interpret()**:** Int = n

}

**object** **ExpressionUtils** {

**def** isOperator(s**:** String)**:** Boolean = {

**if** (s == "+" || s == "\*" || s == "-") **true**

**else** **false**

}

**def** getOperator(s**:** String, left**:** Expression, right**:** Expression)**:** Expression = s **match** {

**case** "+" **=>** **Add**(left, right)

**case** "\*" **=>** **Product**(left, right)

**case** "-" **=>** **Substract**(left, right)

}

}

**object** **InterpreterTest** **extends** **App** {

**val** tokenString **=** "7 3 - 2 1 + \*"

**val** stack **=** **new** util.**Stack**[Expression]()

**val** tokenArray **=** tokenString.split(" ")

**for** (s **<-** tokenArray) {

**if** (**ExpressionUtils**.isOperator(s)) {

**val** rightExpression **=** stack.pop()

**val** leftExpression **=** stack.pop()

**val** operator **=** **ExpressionUtils**.getOperator(s, leftExpression, rightExpression)

**val** result **=** operator.interpret()

stack.push(**Number**(result))

} **else** {

**val** i **=** **Number**(**Integer**.parseInt(s.toString))

stack.push(i)

}

}

println(s"($tokenString): ${stack.pop().interpret()}")

}

*//Alternativa con ADT's*

**sealed** **trait** **Option**[A]{

*/\*\**

*\* Equivalente del método interpret()*

*\**

*\* @param z El contexto que se le pasa al intérprete*

*\* @param f El significado que se le dará a cada expresión del lenguaje*

*\* @tparam B El tipo de dato que regresa el intérprete*

*\* @return*

*\*/*

**def** fold[B](z**:** B)(f**:** A => B)**:** B = **this** **match** {

**case** **None**() **=>** z

**case** **Some**(a) **=>** f(a)

}

}

**case** **class** **None**[A]() **extends** **Option**[A]

**case** **class** **Some**[A](a**:** A) **extends** **Option**[A]

**object** **FunInterpreterTest** **extends** **App** {

**val** algunNumero**:** Option[Int] **=** **Some**(5)

**val** ningunNumero**:** Option[Int] **=** **None**()

**def** double(opN**:** Option[Int])**:** Int = opN.fold(0)(**\_** \* 2)

println(s"Algun numero: ${double(algunNumero)}\nNingun numero: ${double(ningunNumero)}")

}

*//Alternativa con el Free Monad pattern*

**import** **cats.\_**

**import** **cats.free.\_**

**import** **freasymonad.cats.free**

@free **trait** **Interaction** {

**type** InteractionF[A] **=** **Free**[GrammarADT, A]

**sealed** **trait** **GrammarADT**[A]

**def** ask(a**:** String)**:** InteractionF[String]

**def** tell(msg**:** String)**:** InteractionF[Unit]

}

**object** **InteractionProgram** {

**import** **Interaction.ops.\_**

**def** program**:** InteractionF[Unit] **=**

**for** {

answer **<-** ask("How are you doing?")

**\_** **<-** tell(s"It's goot that you are doing $answer")

} **yield** ()

}

**object** **ConsoleTest** **extends** **App** {

**val** consoleInterpreter **=** **new** **Interaction**.**Interp**[Id] {

**override** **def** ask(a**:** String)**:** Id[String] **=** {

println(a)

**StdIn**.readLine()

}

**override** **def** tell(msg**:** String)**:** Id[Unit] **=** println(msg)

}

consoleInterpreter.run(**InteractionProgram**.program)

}

**object** **SpeechTest** **extends** **App** {

**import** **edu.cmu.sphinx.api.\_**

**import** **scala.util.control.Breaks.\_**

**val** configuration **=** **new** **Configuration**

configuration.setAcousticModelPath("resource:/edu/cmu/sphinx/models/en-us/en-us")

configuration.setDictionaryPath("resource:/edu/cmu/sphinx/models/en-us/cmudict-en-us.dict")

configuration.setLanguageModelPath("resource:/edu/cmu/sphinx/models/en-us/en-us.lm.bin")

**val** speechInterpreter **=** **new** **Interaction**.**Interp**[Id] {

**override** **def** ask(a**:** String)**:** Id[String] **=** {

println(a)

**val** speechRecognizer **=** **new** **LiveSpeechRecognizer**(configuration)

speechRecognizer.startRecognition(**true**)

**var** result **=** speechRecognizer.getResult

**var** answer**:** String = ""

breakable { **while**(**true**) {

**val** word **=** speechRecognizer.getResult.getHypothesis

**if** (word.endsWith("change")) break()

**else** {

println(word)

answer **=** answer + " " + word

}

}}

answer

}

**override** **def** tell(msg**:** String)**:** Id[Unit] **=** println(msg)

}

speechInterpreter.run(**InteractionProgram**.program)

}

**Patrones relacionados:**

Composite, Visitor y State

## Iterator

**Problemas que pretende resolver.**

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 está recorriendo.

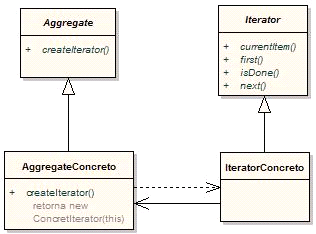
**Causas**

* 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.

**Solución.**

Provee un mecanismo estándar para acceder secuencialmente a los elementos de una colección; define una interface que declara métodos para acceder secuencialmente a los objetos de una colección. Una clase accede a una colección a través de dicha interface.

**Estructura UML.**



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

**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

**Ejemplo de código:**

**using** System**;**

**using** System**.**Collections**;**

**namespace** DoFactory**.**GangOfFour**.**Iterator**.**Structural

**{**

/// <summary>

/// MainApp startup class for Structural

/// Iterator Design Pattern.

/// </summary>

class MainApp

**{**

/// <summary>

/// Entry point into console application.

/// </summary>

static void Main**()**

**{**

ConcreteAggregate a **=** **new** ConcreteAggregate**();**

a**[**0**]** **=** "Item A"**;**

a**[**1**]** **=** "Item B"**;**

a**[**2**]** **=** "Item C"**;**

a**[**3**]** **=** "Item D"**;**

// Create Iterator and provide aggregate

Iterator i **=** a**.**CreateIterator**();**

Console**.**WriteLine**(**"Iterating over collection:"**);**

**object** item **=** i**.**First**();**

**while** **(**item **!=** **null)**

**{**

Console**.**WriteLine**(**item**);**

item **=** i**.**Next**();**

**}**

// Wait for user

Console**.**ReadKey**();**

**}**

**}**

/// <summary>

/// The 'Aggregate' abstract class

/// </summary>

**abstract** class Aggregate

**{**

**public** **abstract** Iterator CreateIterator**();**

**}**

/// <summary>

/// The 'ConcreteAggregate' class

/// </summary>

class ConcreteAggregate **:** Aggregate

**{**

**private** ArrayList \_items **=** **new** ArrayList**();**

**public** **override** Iterator CreateIterator**()**

**{**

**return** **new** ConcreteIterator**(this);**

**}**

// Gets item count

**public** int Count

**{**

**get** **{** **return** \_items**.**Count**;** **}**

**}**

// Indexer

**public** **object** **this[**int index**]**

**{**

**get** **{** **return** \_items**[**index**];** **}**

**set** **{** \_items**.**Insert**(**index**,** **value);** **}**

**}**

**}**

/// <summary>

/// The 'Iterator' abstract class

/// </summary>

**abstract** class Iterator

**{**

**public** **abstract** **object** First**();**

**public** **abstract** **object** Next**();**

**public** **abstract** bool IsDone**();**

**public** **abstract** **object** CurrentItem**();**

**}**

/// <summary>

/// The 'ConcreteIterator' class

/// </summary>

class ConcreteIterator **:** Iterator

**{**

**private** ConcreteAggregate \_aggregate**;**

**private** int \_current **=** 0**;**

// Constructor

**public** ConcreteIterator**(**ConcreteAggregate aggregate**)**

**{**

**this.**\_aggregate **=** aggregate**;**

**}**

// Gets first iteration item

**public** **override** **object** First**()**

**{**

**return** \_aggregate**[**0**];**

**}**

// Gets next iteration item

**public** **override** **object** Next**()**

**{**

**object** ret **=** **null;**

**if** **(**\_current **<** \_aggregate**.**Count **-** 1**)**

**{**

ret **=** \_aggregate**[++**\_current**];**

**}**

**return** ret**;**

**}**

// Gets current iteration item

**public** **override** **object** CurrentItem**()**

**{**

**return** \_aggregate**[**\_current**];**

**}**

// Gets whether iterations are complete

**public** **override** bool IsDone**()**

**{**

**return** \_current **>=** \_aggregate**.**Count**;**

**}**

**}**

**}**

## Mediator

**Tipo:**

Comportamiento, a nivel de objetos.

**Sinopsis:**

Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento (dependencia) al evitar que los objetos se relacionen unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

**Contexto:**

Cuando muchos objetos interactúan con otros objetos, se puede formar una estructura muy compleja, con objetos con muchas conexiones con otros objetos. En un caso extremo cada objeto puede conocer a todos los demás objetos.

**Causas:**

* Cuando nuestro sistema tiene gran número de objetos que se comunican de forma activa, y dicha comunicación es compleja y está bien definida.
* Si la reutilización de un objeto es difícil ya que es dependiente de muchos otros.
* Si nuestro sistema es excesivamente complejo, deberemos subdividirlo en varios Mediator.

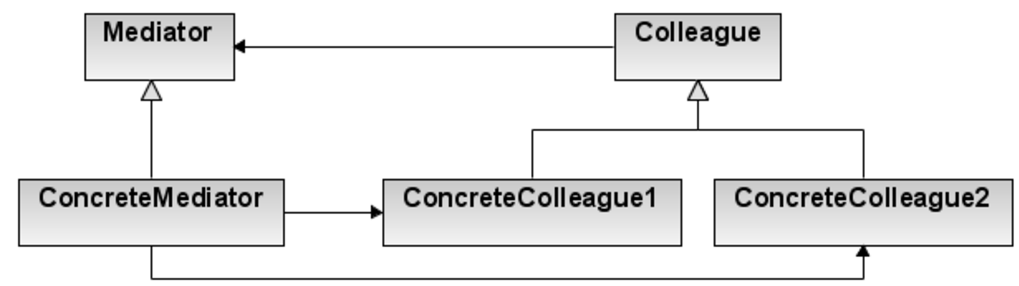
Para evitar esto el patrón Mediator encapsula el comportamiento de todo un conjunto de objetos en un solo objeto.

**Solución:**

La solución consiste en crear una entidad intermediaria que se encargue de gestionar la comunicación entre objetos. En primer lugar, definiremos una interfaz para exponer las operaciones que un intermediario puede realizar, la cual llamaremos Mediator. Como es evidente debemos implementar dicha interfaz mediante una clase ConcreteMediator para dotar a éste de funcionalidad.

El siguiente paso es definir la interfaz de los integrantes del sistema, la cual llamaremos Colleague. Aquí se expondrán todas las operaciones que un objeto perteneciente al sistema puede realizar para comunicarse. Podemos crear diferentes tipos de colegas siempre y cuando respeten la interfaz.

**Estructura:**



Donde:

* Mediator: Establece las operaciones que pueden realizarse sobre el sistema encapsulado.
* ConcreteMediator: Implementa las operaciones anteriores y define la interacción de los elementos del sistema. Actúa de procesador central.
* Colleague: Establece las operaciones comunes que todo integrante del sistema debe implementar.
* ConcreteColleague: Es el integrante real del sistema que implementa todas las operaciones establecidas por la interfaz Colleague y las suyas propias.

**Consecuencias:**

POSITIVAS:

* Desacopla a los integrantes del sistema de sus interacciones. De esta manera podrían reutilizarse o modificar los Colleague y los Mediator de manera independiente.
* Simplifica la comunicación entre objetos: los objetos que se comunican de "muchos a muchos" ahora lo harán de "uno a muchos", disminuyendo su complejidad.
* Si se plantea correctamente, la interacción entre los integrantes es más fácil de entender.

NEGATIVAS:

* Si los sistemas son demasiado grandes, las ventajas en materia de simplicidad y facilidad de modificación desaparecen.

**Implementación:**

* Primero, simplifica el mantenimiento, al localizar su posible comportamiento, en caso contrario se esparce entre diferentes clases en un mismo objeto.
* Segundo, evita la necesidad de acoplar los objetos.
* Tercero, esto simplifica la lógica al reemplazar las interacciones "Muchos a muchos" entre controles individuales con interacciones "uno a muchos" entre el mediador y los colegas.

**Ejemplo del código:** ejemploMediator.cs

**using** **System**;

**namespace** **DoFactory.GangOfFour.Mediator.Structural**

{

*/// <summary>*

*/// MainApp startup class for Structural*

*/// Mediator Design Pattern.*

*/// </summary>*

**class** **MainApp**

{

*/// <summary>*

*/// Entry point into console application.*

*/// </summary>*

**static** **void** Main()

{

ConcreteMediator m = **new** ConcreteMediator();

ConcreteColleague1 c1 = **new** ConcreteColleague1(m);

ConcreteColleague2 c2 = **new** ConcreteColleague2(m);

m.Colleague1 = c1;

m.Colleague2 = c2;

c1.Send("How are you?");

c2.Send("Fine, thanks");

*// Wait for user*

Console.ReadKey();

}

}

*/// <summary>*

*/// The 'Mediator' abstract class*

*/// </summary>*

**abstract** **class** **Mediator**

{

**public** **abstract** **void** Send(string message,

Colleague colleague);

}

*/// <summary>*

*/// The 'ConcreteMediator' class*

*/// </summary>*

**class** **ConcreteMediator** : Mediator

{

**private** ConcreteColleague1 \_colleague1;

**private** ConcreteColleague2 \_colleague2;

**public** ConcreteColleague1 Colleague1

{

**set** { \_colleague1 = **value**; }

}

**public** ConcreteColleague2 Colleague2

{

**set** { \_colleague2 = **value**; }

}

**public** **override** **void** Send(string message,

Colleague colleague)

{

**if** (colleague == \_colleague1)

{

\_colleague2.Notify(message);

}

**else**

{

\_colleague1.Notify(message);

}

}

}

*/// <summary>*

*/// The 'Colleague' abstract class*

*/// </summary>*

**abstract** **class** **Colleague**

{

**protected** Mediator mediator;

*// Constructor*

**public** Colleague(Mediator mediator)

{

**this**.mediator = mediator;

}

}

*/// <summary>*

*/// A 'ConcreteColleague' class*

*/// </summary>*

**class** **ConcreteColleague1** : Colleague

{

*// Constructor*

**public** ConcreteColleague1(Mediator mediator)

: **base**(mediator)

{

}

**public** **void** Send(string message)

{

mediator.Send(message, **this**);

}

**public** **void** Notify(string message)

{

Console.WriteLine("Colleague1 gets message: "

+ message);

}

}

*/// <summary>*

*/// A 'ConcreteColleague' class*

*/// </summary>*

**class** **ConcreteColleague2** : Colleague

{

*// Constructor*

**public** ConcreteColleague2(Mediator mediator)

: **base**(mediator)

{

}

**public** **void** Send(string message)

{

mediator.Send(message, **this**);

}

**public** **void** Notify(string message)

{

Console.WriteLine("Colleague2 gets message: "

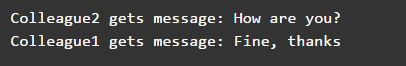
+ message);

}

}

}

**Salida:**



**Patrones relacionados:**

* Facade, la diferencia es que el protocolo de mediator es multidireccional mientras que el de facade es unidireccional (solo hacía los elementos del sistema)
* Observer, ya que los colegas (collegues) pueden comunicarse con el mediator mediante el patrón observer.

## Memento

Este patrón de diseño permite capturar y exportar el estado interno de un objeto para que luego se pueda restaurar, sin romper la encapsulación.

Su finalidad es almacenar el estado de un objeto (o del sistema completo) en un momento dado, de manera que se pueda restaurar posteriormente si fuese necesario. 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.

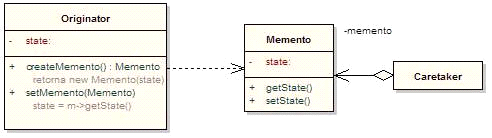
**Se usa cuando:**

* 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).

**Solucion.**

Se usa este patrón cuando se quiere poder restaurar el sistema desde estados pasados y por otra parte, es usado cuando se desea 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).

**Estructura UML**



**Participantes**

* Caretaker: es responsable por mantener a salvo a Memento. No opera o examina su contenido.
* Memento: almacena el estado interno de un objeto Originator. El Memento puede
* almacenar todo o parte del estado.
* Originator: crea un objeto Memento conteniendo una fotografía de su estado interno.

**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.

**Ejemplo de código:**

**using** System**;**

**namespace** DoFactory**.**GangOfFour**.**Memento**.**Structural

**{**

/// <summary>

/// MainApp startup class for Structural

/// Memento Design Pattern.

/// </summary>

class MainApp

**{**

/// <summary>

/// Entry point into console application.

/// </summary>

static void Main**()**

**{**

Originator o **=** **new** Originator**();**

o**.**State **=** "On"**;**

// Store internal state

Caretaker c **=** **new** Caretaker**();**

c**.**Memento **=** o**.**CreateMemento**();**

// Continue changing originator

o**.**State **=** "Off"**;**

// Restore saved state

o**.**SetMemento**(**c**.**Memento**);**

// Wait for user

Console**.**ReadKey**();**

**}**

**}**

/// <summary>

/// The 'Originator' class

/// </summary>

class Originator

**{**

**private** string \_state**;**

// Property

**public** string State

**{**

**get** **{** **return** \_state**;** **}**

**set**

**{**

\_state **=** **value;**

Console**.**WriteLine**(**"State = " **+** \_state**);**

**}**

**}**

// Creates memento

**public** Memento CreateMemento**()**

**{**

**return** **(new** Memento**(**\_state**));**

**}**

// Restores original state

**public** void SetMemento**(**Memento memento**)**

**{**

Console**.**WriteLine**(**"Restoring state..."**);**

State **=** memento**.**State**;**

**}**

**}**

/// <summary>

/// The 'Memento' class

/// </summary>

class Memento

**{**

**private** string \_state**;**

// Constructor

**public** Memento**(**string state**)**

**{**

**this.**\_state **=** state**;**

**}**

// Gets or sets state

**public** string State

**{**

**get** **{** **return** \_state**;** **}**

**}**

**}**

/// <summary>

/// The 'Caretaker' class

/// </summary>

class Caretaker

**{**

**private** Memento \_memento**;**

// Gets or sets memento

**public** Memento Memento

**{**

**set** **{** \_memento **=** **value;** **}**

**get** **{** **return** \_memento**;** **}**

**}**

**}**

**}**

## Observer

El patrón Observer puede ser utilizado cuando hay objetos que dependen de otro, necesitando ser notificados en caso de que se produzca algún cambio en él.

Define una dependencia de uno-a-muchos entre objetos de forma que, cuando un objeto cambia de estado, se notifica a los objetos dependientes para que se actualicen automáticamente. También conocido como dependents, publish subscribe.

**Causas**.

* Cuando una abstracción tiene dos aspectos y uno depende del otro. Encapsularlos permite cambiarlos y reutilizarlos
* Cuando cambiar un objeto implicar cambiar un numero n de otros objetos, por lo que no sabe cuántos debemos de cambiar.
* Cuando un objeto requiere notificar o avisar algo a otros objetos, sin hacer suposiciones sobre quienes son dichos objetos, por lo que se habla de un bajo acoplamiento.

**Solución**.

Mantener la consistencia entre objetos relacionados, sin aumentar el acoplamiento entre clases

Subject 
+ attach(Observer) 
+ detach (Observer) 
+ notify O o- 
- forall o in observers: 
o.update(); 
subject 
ConcreteSubject 
- subjectState 
+ getState() o- 
+ setState() 
return subjectState; 
observers 
Observer 
+ update() 
ConcreteObserver 
- observerState 
+ update() o 
observerState = subject.getState(); 

**Participantes.**

Subject:

* Conoce a sus observadores, que pueden ser un número arbitrario
* proporciona una interfaz para añadir y quitar objetos observadores

Observer:

* Define la interfaz de los objetos a los que se deben notificar cambios en un sujeto.

ConcreteSubject:

* Almacena el estado de interés para sus observadores
* Envía notificaciones a sus observadores cuando su estado cambia

ConcreteObserver:

* Mantiene una referencia a un ConcreteSubject
* Almacena el estado del sujeto que le resulta de interés
* Implementa la interfaz de Observer para mantener su estado consistente con el del sujeto.

Diagrama de secuencia de un sujeto con 2 observadores.

setState() 
:ConcreteSubject 
noti 
col 
:ConcreteObserver 
u ate 
etState 
u date 
etState 
c02 
:ConcreteObserver 

**Consecuencias**

* Permite modificar sujetos y observadores de manera independiente
* Permite reutilizar un sujeto sin reutilizar sus observadores, y viceversa
* Permite añadir observadores sin tener que cambiar el sujeto ni los

demás observadores

* Acoplamiento abstracto entre el sujeto y el observador.
* El sujeto no sabe la clase concreta de sus observadores (acoplamiento mínimo).
* Soporte para broadcast. El sujeto envía la notificación a todos los

observadores suscritos.

* Se pueden añadir/quitar observadores.
* Una operación en el sujeto puede desencadenar una cascada de cambios en sus observadores.
* El protocolo no ofrece detalles sobre lo que ha cambiado.

**Implementación.**

La clase observable es la siguiente.

**package** observer;

**import** **java.util.Observable**;

*/\*\**

*\**

*\* @author Cristian*

*\*/*

**public** **class** **Observado** **extends** Observable{

*//Esta clase representa a un objeto que puede ser observado por varios observadores.*

String mensaje;

**public** Observado(){

mensaje = "Obervando al objeto";

}

**public** void mensaje(String m){

mensaje = m;

*//REalizamos un cambio al objeto observado*

setChanged();

notifyObservers(mensaje);

*//EL metodo avisa que hubo cambio al observador en su propuedad mensaje y lo notifica.*

}

}

La clase que observará.

**package** observer;

**import** **java.util.Observable**;

**import** **java.util.Observer**;

**public** **class** **Observador** **implements** Observer{

*//Esta clase solo se encarga de observar y recibir las notificaciones del objeto observado.*

@Override

**public** void update(Observable o, Object arg){

System.out.println("Actualizacion: "+ o.toString()+arg.toString()+"\n");

}

}

En el main podemos ver como se crean los objetos y los cambios que estos generaran.

*/\*\**

*\**

*\* @author Cristian*

*\*/*

**public** **class** **Observer** {

*/\*\**

*\* @param args the command line arguments*

*\*/*

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

*//Se instanciaron un objeto observador y un observable.*

Observado observado = **new** Observado();

Observador observador = **new** Observador();

*//Se agrega al observador.*

observado.addObserver(observador);

observado.mensaje("hola");

observado.mensaje("cambio de mensaje");

observado.mensaje("mensaje nuevo");

}

}

## State

**Sinopsis:**

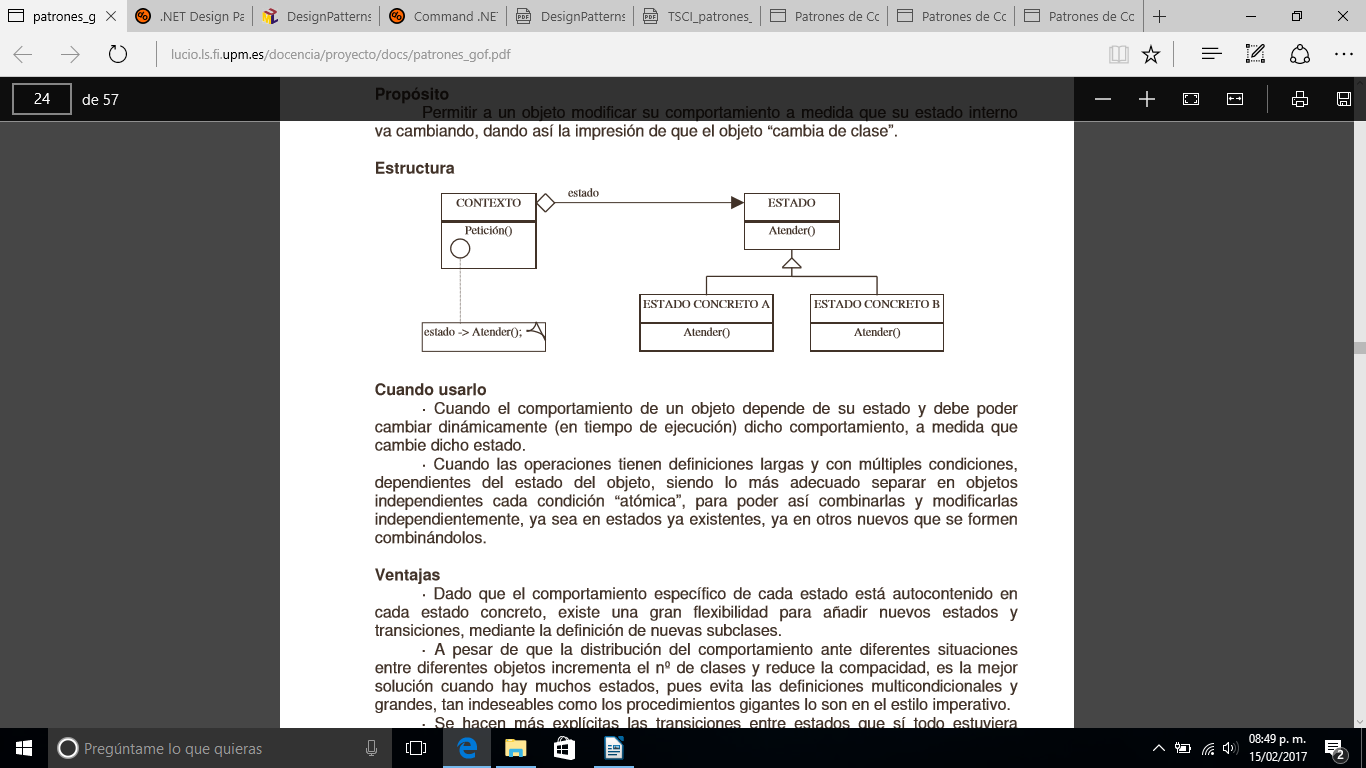
Permitir a un objeto modificar su comportamiento a medida que su estado interno va cambiando, dando así la impresión de que el objeto “cambia de clase”.

**Contexto:**

La aplicación de este patrón será adecuada en entornos en los que se identifiquen “estados” con operaciones comunes que varíen su comportamiento dependiendo de en cuál de ellos se encuentren activos. Un buen ejemplo de este comportamiento sería la implementación de un Workflow de un gestor de defectos en el que un elemento de trabajo pueda encontrarse en distintos estados (nuevo, en curso, corregido…) y en el que cada uno de los estados implica un conjunto de posibles operaciones y transiciones que difieren entre sí.

**Causas:**

* Cuando el comportamiento de un objeto depende de su estado y debe poder cambiar dinámicamente (en tiempo de ejecución) dicho comportamiento, a medida que cambie dicho estado.
* Cuando las operaciones tienen definiciones largas y con múltiples condiciones, dependientes del estado del objeto, siendo lo más adecuado separar en objetos independientes cada condición “atómica”, para poder así combinarlas y modificarlas independientemente, ya sea en estados ya existentes, ya en otros nuevos que se formen combinándolos.

**Solución:**

**Consecuencias:**

Dado que el comportamiento específico de cada estado está autocontenido en cada estado concreto, existe una gran flexibilidad para añadir nuevos estados y transiciones, mediante la definición de nuevas subclases.

A pesar de que la distribución del comportamiento ante diferentes situaciones entre diferentes objetos incrementa el nº de clases y reduce la compacidad, es la mejor solución cuando hay muchos estados, pues evita las definiciones multicondicionales y grandes, tan indeseables como los procedimientos gigantes lo son en el estilo imperativo.

Se hacen más explícitas las transiciones entre estados que sí todo estuviera concentrado en una sola clase (esto hace más inteligible el diseño), impidiendo además los estados internos inconsistentes (desde el punto de vista del contexto, los cambios de estado son ATÓMICOS).

Si los objetos que representan los estados no tienen variables de instancia (esto es, el estado está totalmente definido en su propio tipo) entonces diferentes contextos pueden compartir estados, como PesoMosca sin estado intrínseco, sólo comportamiento.

**Implementación:**

Modelaremos este ejemplo con una clase a la que llamaremos “VehiculoBasico”. Su funcionamiento será sencillo: contará con una variable entera que almacenará el estado actual, otra que almacenará la velocidad, y un conjunto de métodos que serán los encargados de realizar las transiciones. ¿Cómo lo controlamos? Pues a priori, no tenemos mucho donde elegir: a través de una serie de sentencias switch o if…else, que comprueben el estado actual y actúen de forma distinta dependiendo del estado en el que nos encontremos. Comencemos con la declaración de la clase, un conjunto de enteros que modelen los posibles estados y sus atributos:

public class VehiculoBasico

{

    // Estados

    private const int APAGADO = 0;

    private const int PARADO = 1;

    private const int EN\_MARCHA = 2;

    // Atributos

    private const int VELOCIDAD\_MAXIMA = 200;

    private int estado = APAGADO;

    private int velocidadActual = 0;

}

A continuación modelaremos cada uno de los métodos que pueden realizar transiciones. Comenzaremos con el método Contacto:

public void Contacto()

{

    switch (estado)

    {

        case APAGADO:

            {

                // El vehiculo arranca

                estado = PARADO;

                velocidadActual = 0;

                Console.WriteLine("El vehiculo se encuentra ahora PARADO");

                break;

            }

        case PARADO:

            {

                // El vehiculo se apaga

                estado = APAGADO;

                Console.WriteLine("El vehiculo se encuentra ahora APAGADO");

                break;

            }

        case EN\_MARCHA:

            {

                // No se puede detener el vehiculo en marcha!

                Console.WriteLine("ERROR: No se puede cortar el contacto en marcha!");

                break;

            }

        default:

            break;

    }

}

Como vemos, lo primero que haremos en el método será comprobar el estado actual y/o otros atributos. Acorde a esta información, realizaremos las operaciones oportunas. Seguiremos con el método Acelerar

public void Acelerar()

{

    switch (estado)

    {

        case APAGADO:

            {

                // Acelerar con el vehiculo apagado no sirve de mucho <img class="emoji" draggable="false" alt="" src="https://s0.wp.com/wp-content/mu-plugins/wpcom-smileys/twemoji/2/svg/1f642.svg">

                Console.WriteLine("ERROR: El vehiculo esta apagado. Efectue el contacto para iniciar");

                break;

            }

        case PARADO:

            {

                // El vehiculo se pone en marcha. Aumenta la velocidad y cambiamos de estado

                velocidadActual += 10;

                estado = EN\_MARCHA;

                Console.WriteLine("El vehiculo se encuentra ahora EN\_MARCHA");

                Console.WriteLine("Velocidad actual: " + velocidadActual);

                break;

            }

        case EN\_MARCHA:

            {

                // Aumentamos la velocidad, permaneciendo en el mismo estado

                if (velocidadActual >= VELOCIDAD\_MAXIMA)

                    Console.WriteLine("ERROR: El coche ha alcanzado su velocidad maxima");

                else

                    velocidadActual += 10;

                break;

            }

        default:

            break;

    }

}

Finalmente, codificaremos el último de los métodos: Frenar.

public void Frenar()

{

    switch (estado)

    {

        case APAGADO:

            {

                // Frenar con el vehiculo parado tampoco sirve de mucho...

                Console.WriteLine("ERROR: El vehiculo esta apagado. Efectue el contacto para iniciar");

                break;

            }

        case PARADO:

            {

                // No ocurre nada. Si el vehiculo ya se encuentra detenido, no habra efecto alguno

                Console.WriteLine("ERROR: El vehiculo ya se encuentra detenido");

                break;

            }

        case EN\_MARCHA:

            {

                // Reducimos la velocidad. Si esta llega a 0, cambiaremos a estado "PARADO"

                velocidadActual -= 10;

                if (velocidadActual <= 0)

                {

                    estado = PARADO;

                    Console.WriteLine("El vehiculo se encuentra ahora PARADO");

                }

                Console.WriteLine("Velocidad actual: " + velocidadActual);

                break;

            }

        default:

            break;

    }

}

Ahora codificaremos un pequeño programita que haga uso de nuestra clase:

VehiculoBasico vb = new VehiculoBasico();

vb.Acelerar();

vb.Contacto();

vb.Acelerar();

vb.Acelerar();

vb.Acelerar();

vb.Frenar();

vb.Frenar();

vb.Frenar();

vb.Frenar();

El resultado es bastante previsible: al acelerar sin arrancar obtenemos un error. Damos contacto, aceleramos un poco y vamos frenando hasta que el vehículo se detiene. E incluso pisamos el freno estando ya parados, por lo que el vehículo nos informa de ello.

**Patrones relacionados:**

Strategy

## Strategy

**Tipo:**

Comportamiento, a nivel de objetos.

**Sinopsis:**

Definir una familia de algoritmos relacionados, encapsularlos y hacerlos intercambiables.

**Contexto:**

En la programación orientada a objetos muchas clases relacionadas difieren sólo en su comportamiento, se necesitan distintas variantes del mismo algoritmo y además una clase define muchos comportamientos.

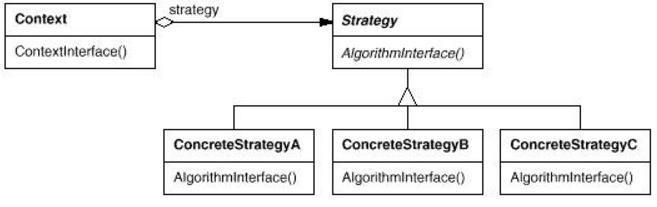
**Causas:**

* Cuando muchas clases relacionadas sólo se diferencian en su comportamiento.
* Cuando se necesitan diferentes variantes de un algoritmo.
* Cuando un algoritmo usa datos que los clientes no tienen por qué conocer.
* Cuando una clase define muchos comportamientos, los cuales se manifiestan como definiciones condicionales múltiples de sus operaciones.

**Solución:**

El patrón Strategy encapsulará un algoritmo completo ignorando los detalles de su implementación, permitiendo intercambiarlo en tiempo de ejecución para permitir actuar a la clase cliente con un comportamiento distinto.

**Estructura:**



Donde:

* Strategy: Declara una interfaz común a todos los algoritmos soportados
* ConcreteStrategy: Implementa el algoritmo que que usa la interfaz strategy.
* Context: Se configuran con un objeto ConcreteStrategy, mantiene una referencia a un objeto Strategy, puede definir una interfaz que permite acceder a sus datos

**Consecuencias:**

POSITIVAS:

* Las familias jerárquicas de estrategias definen algoritmos y comportamientos que enfatizan la reutilización. La herencia puede ayudar a sacar factor común a la funcionalidad de los algoritmos.
* El encapsulamiento de algoritmos en clases separadas ofrece una ventajosa alternativa a la especialización por herencia del contexto para obtener un comportamiento diferente, que promueve la independencia, la facilidad de entender el diseño y la posibilidad de futuras extensiones.
* Se eliminan las costosas definiciones de comportamiento multicondicionales.
* Se posibilita ofrecer diferentes implementaciones del mismo comportamiento, en función de restricciones como el espacio en memoria o el tiempo de respuesta.

NEGATIVAS:

* Los clientes deben tener un cierto conocimiento de cada estrategia, para así poder elegir en cada situación cual es la más apropiada.
* Dado que todas las estrategias comportan una interfaz común, si las diferencias entre ellas es grande, es probable que mucha de la información que se les pasa no sea de utilidad más que a las más complejas.
* Puede producirse una gran explosión en el número de objetos del sistema.

**Implementación:**

Las posibles estrategias se ejecutan dentro de un objeto de contexto que se encarga de recuperar la estrategia apropiada para el cliente. Cada una de las estrategias implementa una interfaz que define la firma del método de la estrategia.

**Ejemplo del código:** ejemploStrategy.cs

**using** **System**;

**using** **System.Collections.Generic**;

**namespace** **DoFactory.GangOfFour.Strategy.RealWorld**

{

*/// <summary>*

*/// MainApp startup class for Real-World*

*/// Strategy Design Pattern.*

*/// </summary>*

**class** **MainApp**

{

*/// <summary>*

*/// Entry point into console application.*

*/// </summary>*

**static** **void** Main()

{

*// Two contexts following different strategies*

SortedList studentRecords = **new** SortedList();

studentRecords.Add("Samual");

studentRecords.Add("Jimmy");

studentRecords.Add("Sandra");

studentRecords.Add("Vivek");

studentRecords.Add("Anna");

studentRecords.SetSortStrategy(**new** QuickSort());

studentRecords.Sort();

studentRecords.SetSortStrategy(**new** ShellSort());

studentRecords.Sort();

studentRecords.SetSortStrategy(**new** MergeSort());

studentRecords.Sort();

*// Wait for user*

Console.ReadKey();

}

}

*/// <summary>*

*/// The 'Strategy' abstract class*

*/// </summary>*

**abstract** **class** **SortStrategy**

{

**public** **abstract** **void** Sort(List<string> list);

}

*/// <summary>*

*/// A 'ConcreteStrategy' class*

*/// </summary>*

**class** **QuickSort** : SortStrategy

{

**public** **override** **void** Sort(List<string> list)

{

list.Sort(); *// Default is Quicksort*

Console.WriteLine("QuickSorted list ");

}

}

*/// <summary>*

*/// A 'ConcreteStrategy' class*

*/// </summary>*

**class** **ShellSort** : SortStrategy

{

**public** **override** **void** Sort(List<string> list)

{

*//list.ShellSort(); not-implemented*

Console.WriteLine("ShellSorted list ");

}

}

*/// <summary>*

*/// A 'ConcreteStrategy' class*

*/// </summary>*

**class** **MergeSort** : SortStrategy

{

**public** **override** **void** Sort(List<string> list)

{

*//list.MergeSort(); not-implemented*

Console.WriteLine("MergeSorted list ");

}

}

*/// <summary>*

*/// The 'Context' class*

*/// </summary>*

**class** **SortedList**

{

**private** List<string> \_list = **new** List<string>();

**private** SortStrategy \_sortstrategy;

**public** **void** SetSortStrategy(SortStrategy sortstrategy)

{

**this**.\_sortstrategy = sortstrategy;

}

**public** **void** Add(string name)

{

\_list.Add(name);

}

**public** **void** Sort()

{

\_sortstrategy.Sort(\_list);

*// Iterate over list and display results*

**foreach** (string name **in** \_list)

{

Console.WriteLine(" " + name);

}

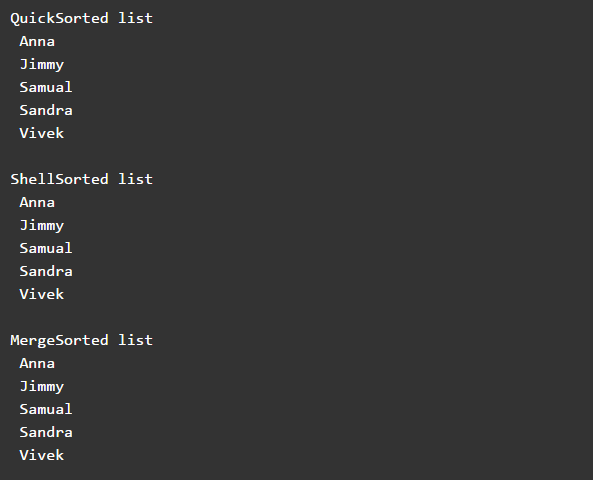
Console.WriteLine();

}

}

}

**Salida:**



**Patrones relacionados:**

* Es parecido al patrón State (en el que un objeto puede cambiar su estado interno durante la ejecución del programa pudiendo comportarse de forma diferente entonces), mientras que con Strategy se crean varios objetos de un mismo tipo, pero pudiendo tener distintos comportamientos cada uno de ellos.

## Template method

**Sinopsis**:

“Permitir que ciertos pasos de un algoritmo definidos en un método de una clase sean redefinidos en sus clases derivadas sin necesidad de sobrecargar la operación entera”.

**Contexto**:

Si el patrón Command nos permite encapsular una invocación a un método, el patrón Template Method o Método Modelo establece una forma de encapsular algoritmos. Este patrón se basa en un principio muy sencillo: si un algoritmo puede aplicarse a varios supuestos en los que únicamente cambie un pequeño número de operaciones, la idea será utilizar una clase para modelarlo a través de sus operaciones. Esta clase base se encargará de definir los pasos comunes del algoritmo, mientras que las clases que hereden de ella implementarán los detalles propios de cada caso concreto, es decir, el código específico para cada caso.

**Causas**:

* Cuando se pretende implementar una sola vez las partes invariantes de un algoritmo, permitiendo a las subclases implementar el comportamiento que puede variar.
* Cuando se quiere “sacar factor común” del comportamiento compartido por varias clases para localizarlo en una única clase, que evita la duplicación de código. A esta técnica se la conoce comúnmente como REFACTORIZAR PARA GENERALIZAR y sigue el esquema:

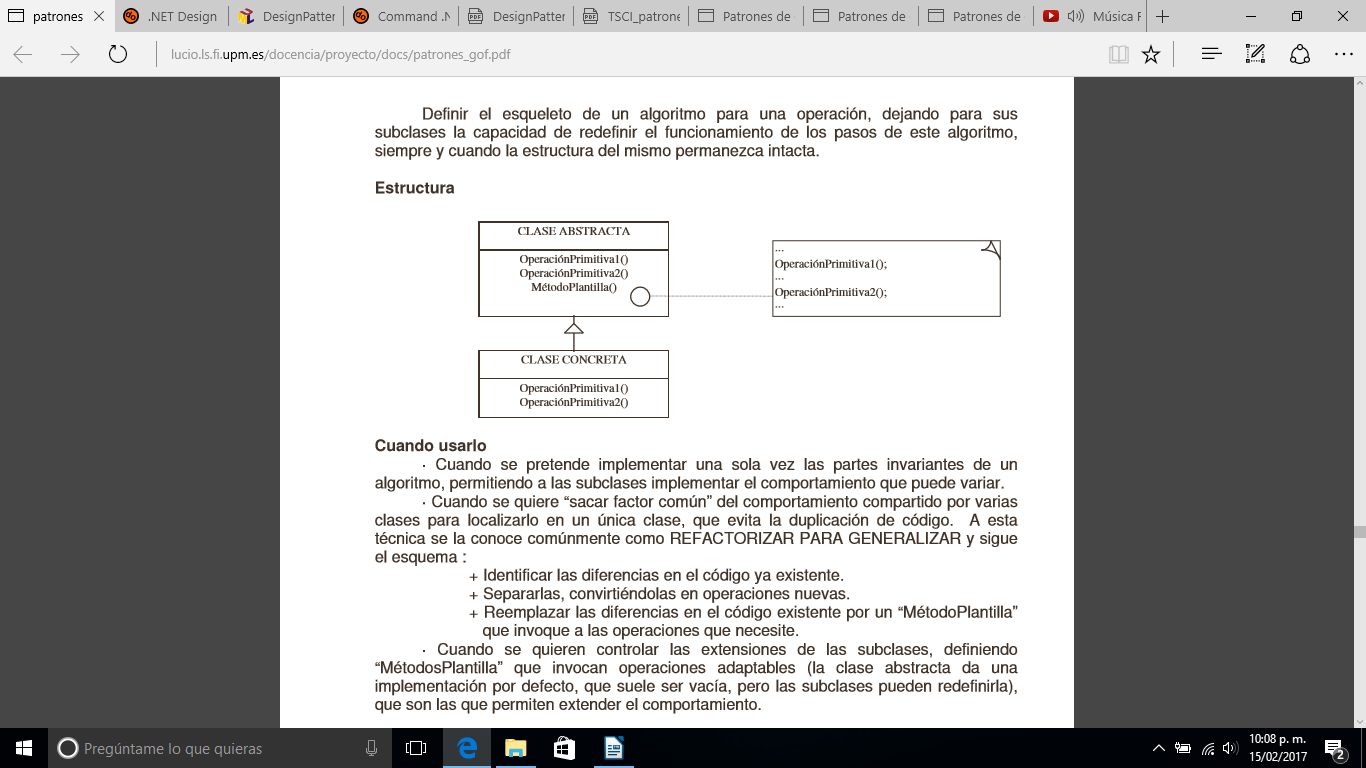
+ Identificar las diferencias en el código ya existente.

+ Separarlas, convirtiéndolas en operaciones nuevas.

+ Reemplazar las diferencias en el código existente por un “MétodoPlantilla” que invoque a las operaciones que necesite.

* Cuando se quieren controlar las extensiones de las subclases, definiendo “MétodosPlantilla” que invocan operaciones adaptables (la clase abstracta da una implementación por defecto, que suele ser vacía, pero las subclases pueden redefinirla), que son las que permiten extender el comportamiento.

**Solución**:

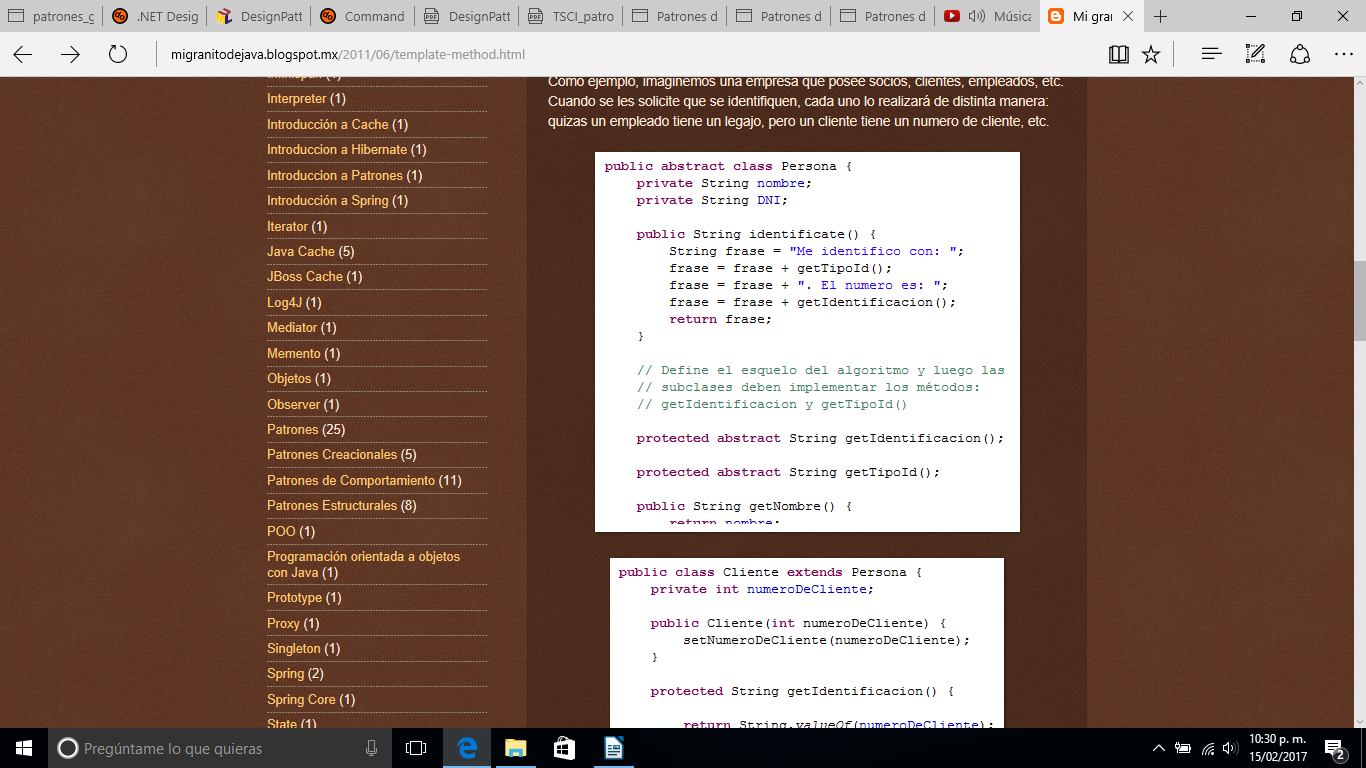
Definir el esqueleto de un algoritmo para una operación, dejando para sus subclases la capacidad de redefinir el funcionamiento de los pasos de este algoritmo, siempre y cuando la estructura del mismo permanezca intacta.

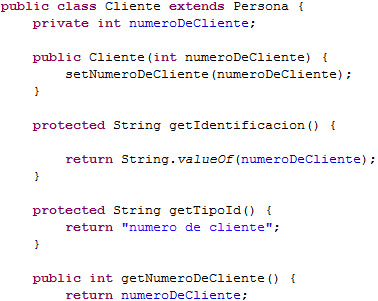
**Consecuencias**:

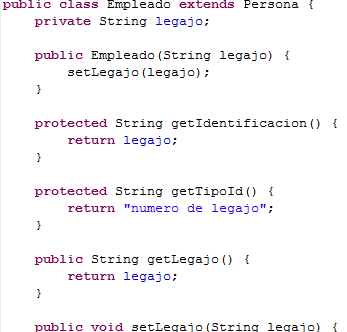
Los “MetodosPlantilla” son una técnica fundamental para reutilizar código, especialmente en las librerías de clases, donde son la razón de ser de la “factorización de comportamiento común”. Llevan a una estructura de control invertido, cuyo paradigma es el “Principio de Hollywood” : <<NO NOS LLAME, YA LE LLAMAREMOS>>, esto es, la clase padre invoca la implementación que la clase hija hace de operaciones primitivas, pero no al revés

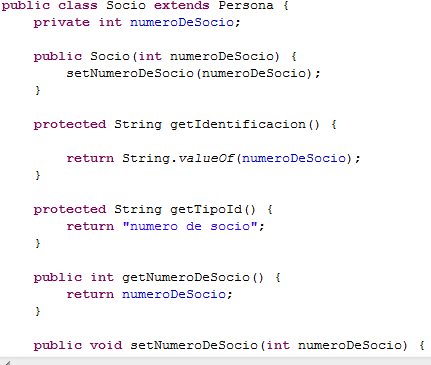
**Implementación**:

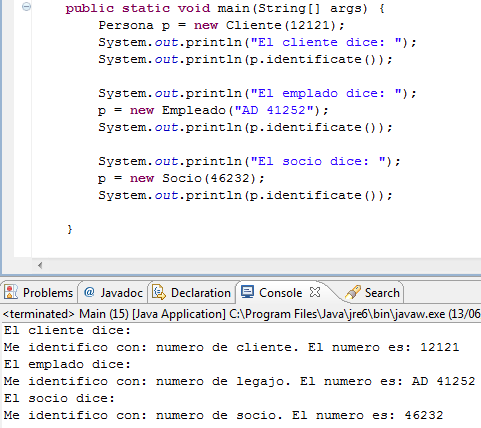
Como ejemplo, imaginemos una empresa que posee socios, clientes, empleados, etc. Cuando se les solicite que se identifiquen, cada uno lo realizará de distinta manera: quizas un empleado tiene un legajo, pero un cliente tiene un numero de cliente, etc.











**Patrones relacionados:**

Strategy

Factory method

## Visitor

El patrón Visitor es una forma de separar el algoritmo de la estructura de un objeto.

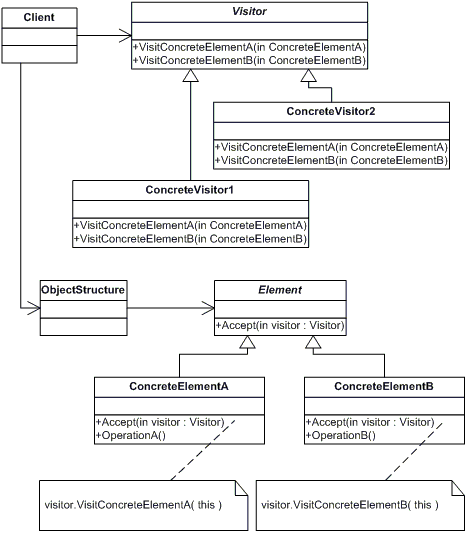
Permite representar una operación que está pensada para ser aplicada sobre los elementos de una estructura de objetos, permitiendo así definir y añadir un nuevo comportamiento sin necesidad de cambiar las clases de los elementos de la estructura de objetos.

**Causas**

* Cuando una “Estructura de Objetos” contiene muchas clases con diferentes interfaces y se quieren añadir operaciones a dichos objetos en función de la interfaz.
* Cuando se quieren añadir a los objetos de una estructura operaciones muy diferentes y sin relación alguna entre ellas, sin necesidad de hacerlo a nivel individual.
* Cuando las clases que definen la “Estructura de Objetos” casi nunca cambian, pero frecuentemente queremos añadir nuevas operaciones a ciertos objetos de la estructura.

**Solución**

La idea básica es que se tiene un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método aceptar (accept()) que recibe al objeto visitante (visitor) como argumento. El visitante es una interfaz que tiene un método visit diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz visitor de la forma: visitorClase1, visitorClase2... visitorClaseN. El método accept de una clase elemento llama al método visit de su clase. Clases concretas de un visitante pueden entonces ser escritas para hacer una operación en particular.



**Participantes**

* Visitor:
  + Declara una operación de visita para cada clase de ConcreteElement en la estructura de objeto. El nombre y la firma de la operación identifican a la clase que envía la solicitud de visita al visitante. Eso permite al visitante determinar la clase concreta del elemento que se está visitando. Entonces el visitante puede acceder a los elementos directamente a través de su interfaz particular.
* ConcreteVisitor
  + Implementa cada operación declarada por el Visitante. Cada operación implementa un fragmento del algoritmo definido para la clase u objeto correspondiente en la estructura. ConcreteVisitor proporciona el contexto para el algoritmo y almacena su estado local. Este estado a menudo acumula resultados durante el recorrido de la estructura.
* Element
  + Define una operación Accept que toma un Visitor como argumento
* ConcreteElement
  + Implemente una operación Accept que toma un Visitor como argumento.
* ObjectStructure
  + Puede enumerar sus elementos
  + Puede proveer de una interfaz de alto nivel para permitir que el Visitor visite a sus elementos
  + Puede ser un patrón Composite o una colección tal como una lista.

**Consecuencias**

* Es fácil añadir nuevas operaciones a la “Estructura De Objetos”; sólo hay que crear un nuevo “Visitante”, en vez de cambiar todas las clases a las que queremos que afecte.
* Se juntan las operaciones relacionadas entre sí, separándose las que nada tienen que ver. Esto simplifica tanto las clases de los “Elementos” como los algoritmos definidos en los “Visitantes”.
* Se pueden recorrer jerarquías formadas por objetos de distintos tipos (distinta clase padre), mientras que un Iterador no puede hacer esto.
* Se puede ir acumulando el estado a medida que se recorre la estructura, en vez de tener que pasarlo como parámetro o guardarlo en variables globales.

**Implementación**

**package** **visitor**

**trait** **Visitable** {

**def** accept(v**:** Visitor)**:** Unit

}

**sealed** **trait** **Articulo** {

**val** precio**:** BigDecimal

}

**case** **class** **CocaCola**() **extends** **Articulo** **with** **Visitable** {

**val** nombre**:** String = "Coca-Cola"

**val** descripcion**:** String = "Deliciosa bebida de Coca-Cola co."

**override** **val** precio**:** BigDecimal = **BigDecimal**(9.99)

**override** **def** accept(v**:** Visitor)**:** Unit = v.visit(**this**)

}

**case** **class** **Guitarra**() **extends** **Articulo** **with** **Visitable** {

**val** nombre**:** String = "Guitarra Fender"

**val** descripcion**:** String = "Guitarra Fender XS-9000"

**override** **val** precio**:** BigDecimal = **BigDecimal**(25000.00)

**override** **def** accept(v**:** Visitor)**:** Unit = v.visit(**this**)

}

**case** **class** **Lapiz**() **extends** **Articulo** **with** **Visitable** {

**val** nombre**:** String = "Lapiz Mirado"

**val** descripcion**:** String = "Lapiz Mirado 2BH"

**override** **val** precio**:** BigDecimal = **BigDecimal**(6.50)

**override** **def** accept(v**:** Visitor)**:** Unit = v.visit(**this**)

}

**trait** **Visitor** {

**def** visit(c**:** CocaCola)**:** Unit

**def** visit(g**:** Guitarra)**:** Unit

**def** visit(lapiz**:** Lapiz)**:** Unit

}

**object** **ImpuestoVisitor** **extends** **Visitor** {

**override** **def** visit(c**:** CocaCola)**:** Unit = println(s"El impuesto de la coca es de: ${c.precio \* .16}")

**override** **def** visit(g**:** Guitarra)**:** Unit = println(s"El impuesto de la guitarra es de: ${g.precio \* .15}")

**override** **def** visit(l**:** Lapiz)**:** Unit = println(s"El impuesto del lapiz es de: ${l.precio \* .20}")

}

**object** **Tienda** **extends** **App** {

*//Usando el patrón Visitor*

**val** articulos **=** **List**(**CocaCola**(), **Guitarra**(), **Lapiz**())

articulos foreach { **\_**.accept(**ImpuestoVisitor**) }

*//Alternativa al patrón Visitor en la programación funcional - Pattern matching*

**def** calcularImpuesto(a**:** Articulo)**:** Unit = a **match** {

**case** c@CocaCola() **=>** println(s"El impuesto de la coca es de: ${c.precio \* .16}")

**case** g@Guitarra() **=>** println(s"El impuesto de la guitarra es de: ${g.precio \* .15}")

**case** l@Lapiz() **=>** println(s"El impuesto del lapiz es de: ${l.precio \* .20}")

}

articulos foreach { articulo **=>** calcularImpuesto(articulo) }

}

**Patrones relacionados:**

Command y con Iterator

# Bibliografía

<http://patronesdediseno.net16.net/comportamiento.html>

<http://lucio.ls.fi.upm.es/docencia/proyecto/docs/patrones_gof.pdf>

<https://www.ecured.cu/Mediator_(Patr%C3%B3n_de_dise%C3%B1o)>

<http://www.dofactory.com/net/mediator-design-pattern>

<http://programacion.net/articulo/patrones_de_diseno_xix_patrones_de_comportamiento_mediator_1022>

<http://informaticapc.com/patrones-de-diseno/mediator.php>

<https://www.ecured.cu/Strategy>