# Patrones de diseño (JAVA)

Jhisbieth Samara Monroy Meléndez

Tutor:

Johann Latorre

Universidad Autónoma de Bucaramanga – UNAD

Desarrollo Web

2021

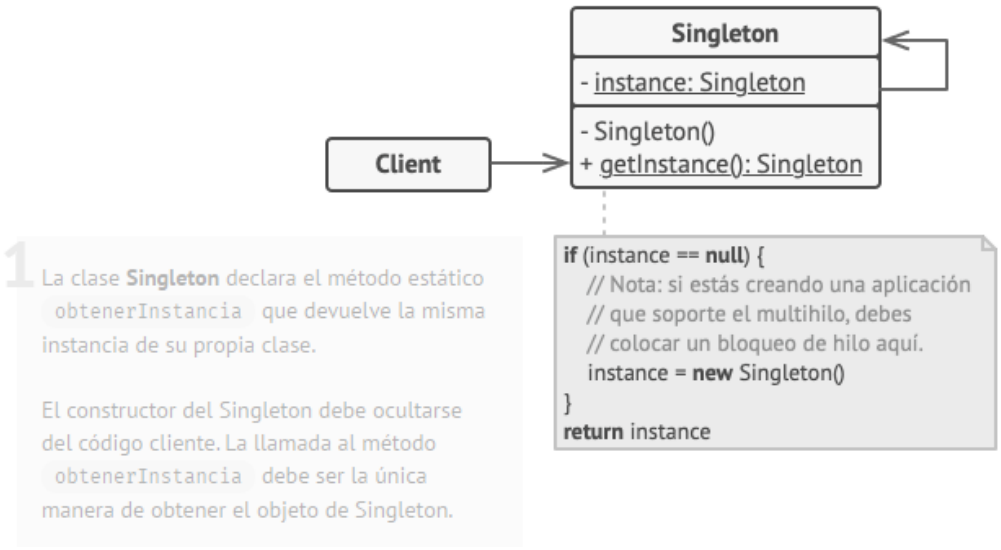
**Patrón singleton:**

Ejemplo de la página refactoring:

**Analogía en el mundo real**

El gobierno es un ejemplo excelente del patrón Singleton. Un país sólo puede tener un gobierno oficial. Independientemente de las identidades personales de los individuos que forman el gobierno, el título "Gobierno de X" es un punto de acceso global que identifica al grupo de personas a cargo.

Utiliza una clase llamada Singleton, en este caso usa una variable de tipo String: value para poder comprobar que el patrón se esté implementando de manera correcta; el Thread.sleep(1000) sirve para dormir el hilo por 1 segundo. Al usar las palabras FOO y VAR cuando se llama dos veces al objeto en el main, se comprueba que se esté reusando el patrón porque sale la misma palabra



```java
package sprint1;
public final class Singleton {
    private static Singleton instance;
    public String value;

    private Singleton(String value) {
        // The following code emulates slow initialization.
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        this.value = value;
    }

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

Main:

```java
package sprint1;

public class DemoSingleThread {
    public static void main(String[] args) {
        System.out.println("If you see the same value, then singleton was reused (yay!)" + "\n" +
                "If you see different values, then 2 singletons were created (booo!!)" + "\n\n" +
                "RESULT:" + "\n");
        Singleton singleton = Singleton.getInstance("FOO");
        Singleton anotherSingleton = Singleton.getInstance("BAR");
        System.out.println(singleton.value);
        System.out.println(anotherSingleton.value);
    }
}
```

```
it - Sprint1 (run)
run:
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (booo!!)

RESULT:

FOO
FOO
BUILD SUCCESSFUL (total time: 1 second)
```

Ejemplo propuesto:

Lista de países: Supongamos que en el registro de la página se debe poner el país al que pertenece ese usuario, para no asignarle un espacio de memoria a cada usuario con la lista de los países (y ya que los nombres de los países es poco probable que cambien), se utiliza el patrón singleton, con este patrón se tendría un único objeto instanciado una vez y cada usuario que necesite esta información pide el único objeto creado con la lista de los países.

**Patrón:**

```
2   package sprint1;
3
4
5   public final class ListaPaises {
6       private static ListaPaises instancia;
7       private ListaPaises(){
8       }
9       public static ListaPaises getInstancia(){
10          if(instancia==null){
11              instancia = new ListaPaises();
12          }
13          return instancia;
14      }
15
16
17  }
18
```

Output - Sprint1 (run)

```
run:
Si ves el mismo valor ListaPaises fue rehusado (yay!)
Si ves dos valores distintos entonces ListaPais fue creado dos veces (booo!!)

RESULT:

1057941451
1057941451
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Main:**

```
2   package sprint1;
3
4   public class Sprint1 {
5       public static void main(String[] args) {
6           System.out.println("Si ves el mismo valor ListaPaises fue rehusado (yay!)" + "\n" +
                    "Si ves dos valores distintos entonces ListaPais fue creado dos veces (booo!!)" + "\n\n" +
8                   "RESULT:" + "\n");
9           ListaPaises p = ListaPaises.getInstancia();
10          ListaPaises p2= ListaPaises.getInstancia();
11
12          System.out.println(p.hashCode());
13          System.out.println(p2.hashCode());
14  }
15  }
```

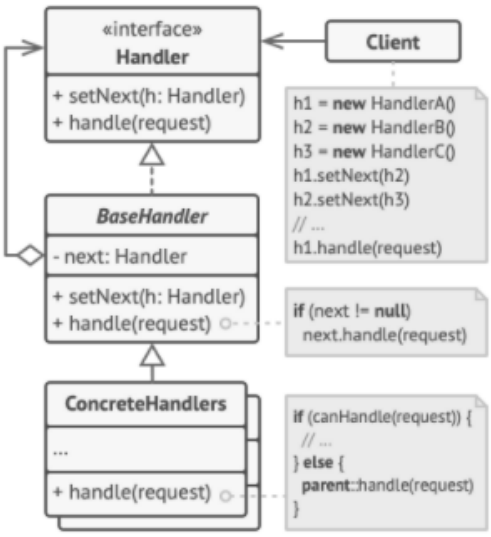
**Patrón Chain of Responsibility**

## Estructura

1 La clase **Manejadora** declara la interfaz común a todos los manejadores concretos. Normalmente contiene un único método para manejar solicitudes, pero en ocasiones también puede contar con otro método para establecer el siguiente manejador de la cadena.

2 La clase **Manejadora Base** es opcional y es donde puedes colocar el código boilerplate (segmentos de código que suelen no alterarse) común para todas las clases manejadoras.

Normalmente, esta clase define un campo para almacenar una referencia al siguiente manejador. Los clientes pueden crear una cadena pasando un manejador al constructor o modificador (*setter*) del manejador previo. La clase también puede implementar el comportamiento de gestión por defecto: puede pasar la ejecución al siguiente manejador después de comprobar su existencia.




4 El **Cliente** puede componer cadenas una sola vez o componerlas dinámicamente, dependiendo de la lógica de la aplicación. Observa que se puede enviar una solicitud a cualquier manejador de la cadena; no tiene por qué ser al primero.

3 Los **Manejadores Concretos** contienen el código para procesar las solicitudes. Al recibir una solicitud, cada manejador debe decidir si procesarla y, además, si la pasa a lo largo de la cadena.

Habitualmente los manejadores son autónomos e inmutables, y aceptan toda la información necesaria únicamente a través del constructor.

Ejemplo de la pagína Refactoring:

Este ejemplo muestra cómo una solicitud que contiene información de usuario pasa una cadena secuencial de manejadores que realizan varias acciones, como la autenticación, autorización y validación.

Este ejemplo es un poco diferente de la versión estándar del patrón establecida por varios autores. La mayoría de ejemplos del patrón se basan en la noción de buscar el manejador adecuado, lanzarlo y salir de la cadena a continuación. Pero aquí ejecutamos todos los manejadores hasta que hay uno que no puede gestionar una solicitud. Ten en cuenta que éste sigue siendo el patrón Chain of Responsibility, aunque el flujo es un poco distinto.

## middleware

### middleware/Middleware.java: Interfaz de validación básica

```java
package refactoring_guru.chain_of_responsibility.example.middleware;

/**
 * Base middleware class.
 */
public abstract class Middleware {
    private Middleware next;

    /**
     * Builds chains of middleware objects.
     */
    public Middleware linkWith(Middleware next) {
        this.next = next;
        return next;
    }

    /**
     * Subclasses will implement this method with concrete checks.
     */
    public abstract boolean check(String email, String password);

    /**
     * Runs check on the next object in chain or ends traversing if we're in
     * last object in chain.
     */
    protected boolean checkNext(String email, String password) {
        if (next == null) {
            return true;
        }
        return next.check(email, password);
    }
```

### middleware/ThrottlingMiddleware.java: Comprueba el límite de cantidad de solicitudes

```
package refactoring_guru.chain_of_responsibility.example.middleware;

/**
 * ConcreteHandler. Checks whether there are too many failed login requests.
 */
public class ThrottlingMiddleware extends Middleware {
    private int requestPerMinute;
    private int request;
    private long currentTime;

    public ThrottlingMiddleware(int requestPerMinute) {
        this.requestPerMinute = requestPerMinute;
        this.currentTime = System.currentTimeMillis();
    }

    /**
     * Please, not that checkNext() call can be inserted both in the beginning
     * of this method and in the end.
     *
     * This gives much more flexibility than a simple loop over all middleware
     * objects. For instance, an element of a chain can change the order of
     * checks by running its check after all other checks.
     */
    public boolean check(String email, String password) {
        if (System.currentTimeMillis() > currentTime + 60_000) {
            request = 0;
            currentTime = System.currentTimeMillis();
        }

        request++;

          if (request > requestPerMinute) {
              System.out.println("Request limit exceeded!");
              Thread.currentThread().stop();
          }
          return checkNext(email, password);
     }
}
```

### middleware/UserExistsMiddleware.java: Comprueba las credenciales del usuario

```
package refactoring_guru.chain_of_responsibility.example.middleware;

import refactoring_guru.chain_of_responsibility.example.server.Server;

/**
 * ConcreteHandler. Checks whether a user with the given credentials exists.
 */
public class UserExistsMiddleware extends Middleware {
    private Server server;

    public UserExistsMiddleware(Server server) {
        this.server = server;
    }

    public boolean check(String email, String password) {
        if (!server.hasEmail(email)) {
            System.out.println("This email is not registered!");
            return false;
        }
        if (!server.isValidPassword(email, password)) {
            System.out.println("Wrong password!");
            return false;
        }
        return checkNext(email, password);
    }
}
```

### middleware/RoleCheckMiddleware.java: Comprueba el papel del usuario

```
package refactoring_guru.chain_of_responsibility.example.middleware;

/**
 * ConcreteHandler. Checks a user's role.
 */
public class RoleCheckMiddleware extends Middleware {
    public boolean check(String email, String password) {
        if (email.equals("admin@example.com")) {
            System.out.println("Hello, admin!");
            return true;
        }
        System.out.println("Hello, user!");
        return checkNext(email, password);
    }
}
```

## server

### server/Server.java: Objetivo de la autorización

```java
package refactoring_guru.chain_of_responsibility.example.server;

import refactoring_guru.chain_of_responsibility.example.middleware.Middleware;

import java.util.HashMap;
import java.util.Map;

/**
 * Server class.
 */
public class Server {
    private Map<String, String> users = new HashMap<>();
    private Middleware middleware;

    /**
     * Client passes a chain of object to server. This improves flexibility and
     * makes testing the server class easier.
     */
    public void setMiddleware(Middleware middleware) {
        this.middleware = middleware;
    }

    /**
     * Server gets email and password from client and sends the authorization
     * request to the chain.
     */
    public boolean logIn(String email, String password) {
        if (middleware.check(email, password)) {
            System.out.println("Authorization have been successful!");
              // Do something useful here for authorized users.

              return true;
          }
          return false;
     }

     public void register(String email, String password) {
         users.put(email, password);
     }

     public boolean hasEmail(String email) {
         return users.containsKey(email);
     }

     public boolean isValidPassword(String email, String password) {
         return users.get(email).equals(password);
     }
 }
```

### Demo.java: Código cliente

```
package refactoring_guru.chain_of_responsibility.example;

import refactoring_guru.chain_of_responsibility.example.middleware.Middleware;
import refactoring_guru.chain_of_responsibility.example.middleware.RoleCheckMiddleware;
import refactoring_guru.chain_of_responsibility.example.middleware.ThrottlingMiddleware;
import refactoring_guru.chain_of_responsibility.example.middleware.UserExistsMiddleware;
import refactoring_guru.chain_of_responsibility.example.server.Server;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * Demo class. Everything comes together here.
 */
public class Demo {
    private static BufferedReader reader = new BufferedReader(new InputStreamReader(Syst
    private static Server server;

    private static void init() {
        server = new Server();
        server.register("admin@example.com", "admin_pass");
        server.register("user@example.com", "user_pass");

        // All checks are linked. Client can build various chains using the same
        // components.
        Middleware middleware = new ThrottlingMiddleware(2);
        middleware.linkWith(new UserExistsMiddleware(server))
                .linkWith(new RoleCheckMiddleware());

        // Server gets a chain from client code.
        server.setMiddleware(middleware);
    }
     public static void main(String[] args) throws IOException {
         init();

         boolean success;
         do {
             System.out.print("Enter email: ");
             String email = reader.readLine();
             System.out.print("Input password: ");
             String password = reader.readLine();
             success = server.logIn(email, password);
         } while (!success);
     }
}
```

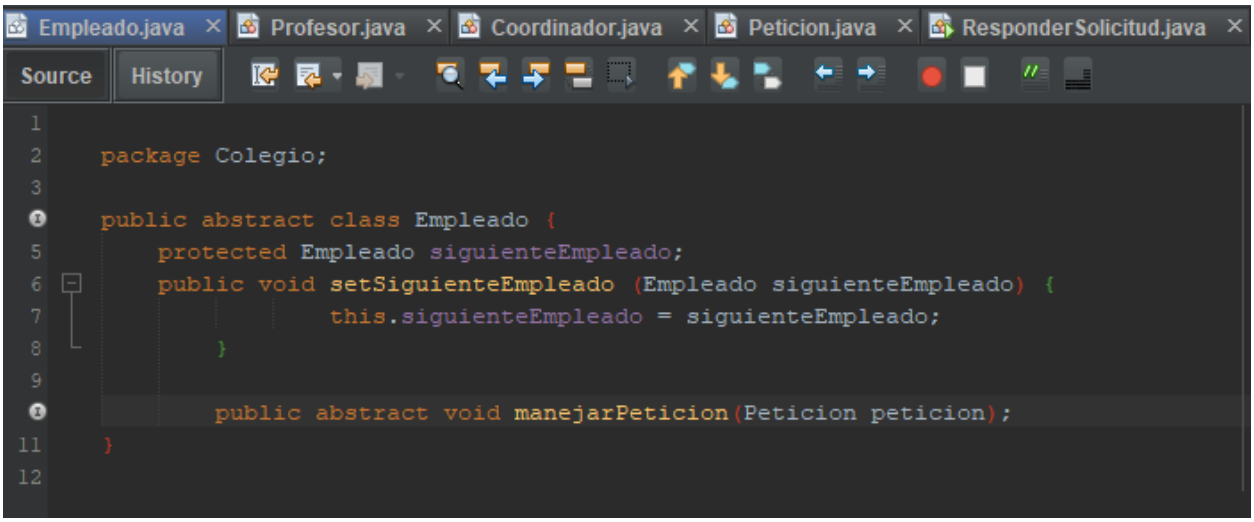
**OutputDemo.txt: Resultado de la ejecución**

```
Enter email: admin@example.com
Input password: admin_pass
Hello, admin!
Authorization have been successful!


Enter email: user@example.com
Input password: user_pass
Hello, user!
Authorization have been successful!
```

Ejemplo propuesto:

Se tiene un colegio que debe procesar solicitudes de los estudiantes, para definir quién resuelve cada solicitud se usa un sistema numérico, el 0 significa que la solicitud la puede responder el profesor, el 1 significa que la solicitud excede la responsabilidad del profesor y debe resolverla el coordinador. Se crea una clase llamada Empleado de la cuál harán extensión las clases profesor y coordinador, también se crea la clase que recibirá la petición.



Empleado.java | Profesor.java | Coordinador.java | Peticion.java | ResponderSolicitud.java

Source | History

```java
package Colegio;

public class Profesor extends Empleado{

    @Override
        public void manejarPeticion(Peticion peticion){

                if (peticion.getValue() == 0) {
                        System.out.println(peticion.getDescripcion() + peticion.getValue());
                        System.out.println("La solicitud del valor 0 es atendida por el profesor");
                }
                else
                {
                        siguienteEmpleado.manejarPeticion(peticion);
                }
        }
}
```

Empleado.java | Profesor.java | Coordinador.java | Peticion.java | ResponderSolicitud.java

Source | History

```java
package Colegio;

public class Coordinador extends Empleado {
    @Override
        public void manejarPeticion(Peticion peticion) {
                if (peticion.getValue() == 1) {
                        System.out.println("Valor: "+ peticion.getDescripcion() + peticion.getValue());
                        System.out.println("La solicitud del valor 1 es atendida por: el coordinador");
                } else {
                        siguienteEmpleado.manejarPeticion(peticion);
                }
        }
}
```

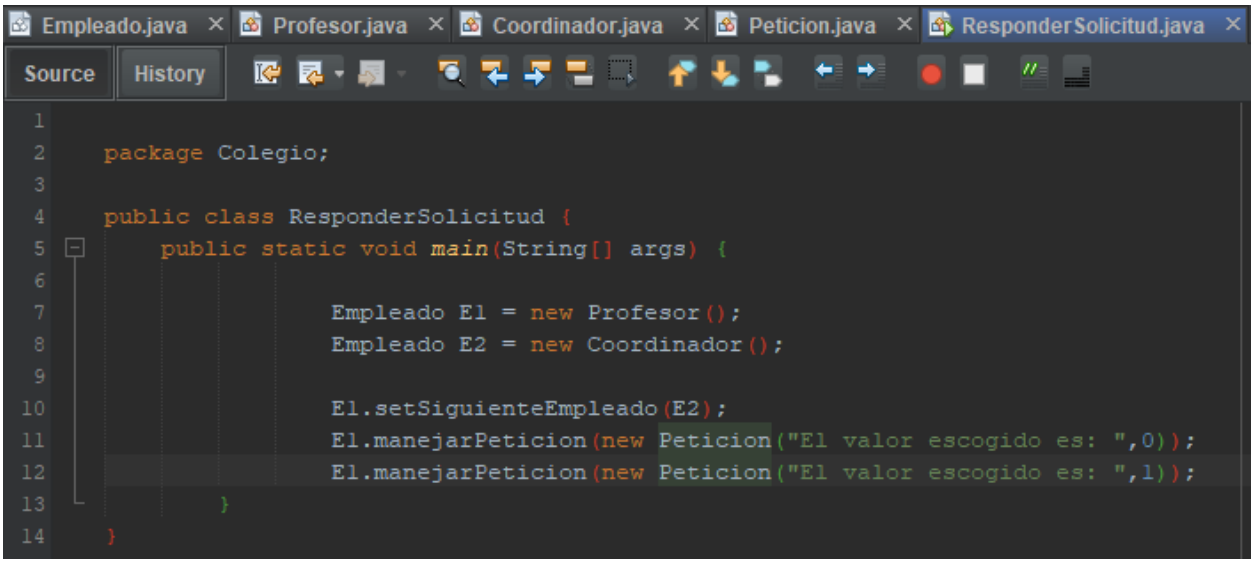
Empleado.java | Profesor.java | Coordinador.java | Peticion.java | ResponderSolicitud.java

Source | History

```java
package Colegio;


public class Peticion {
    private int Value;
    private String descripcion;

    public Peticion (String descripcion , int Value){
        this.Value = Value;
        this.descripcion = descripcion;
    }
    public int getValue() {
        return Value;
    }
    public String getDescripcion() {
        return descripcion;
    }
```


Empleado.java
Profesor.java
Coordinador.java
Peticion.java
ResponderSolicitud.java
Source
History
1
2 package Colegio;
3
4 public class ResponderSolicitud {
5 public static void main(String[] args) {
6
7 Empleado E1 = new Profesor();
8 Empleado E2 = new Coordinador();
9
10 E1.setSiguienteEmpleado(E2);
11 E1.manejarPeticion(new Peticion("El valor escogido es: ",0));
12 E1.manejarPeticion(new Peticion("El valor escogido es: ",1));
13 }
14 }