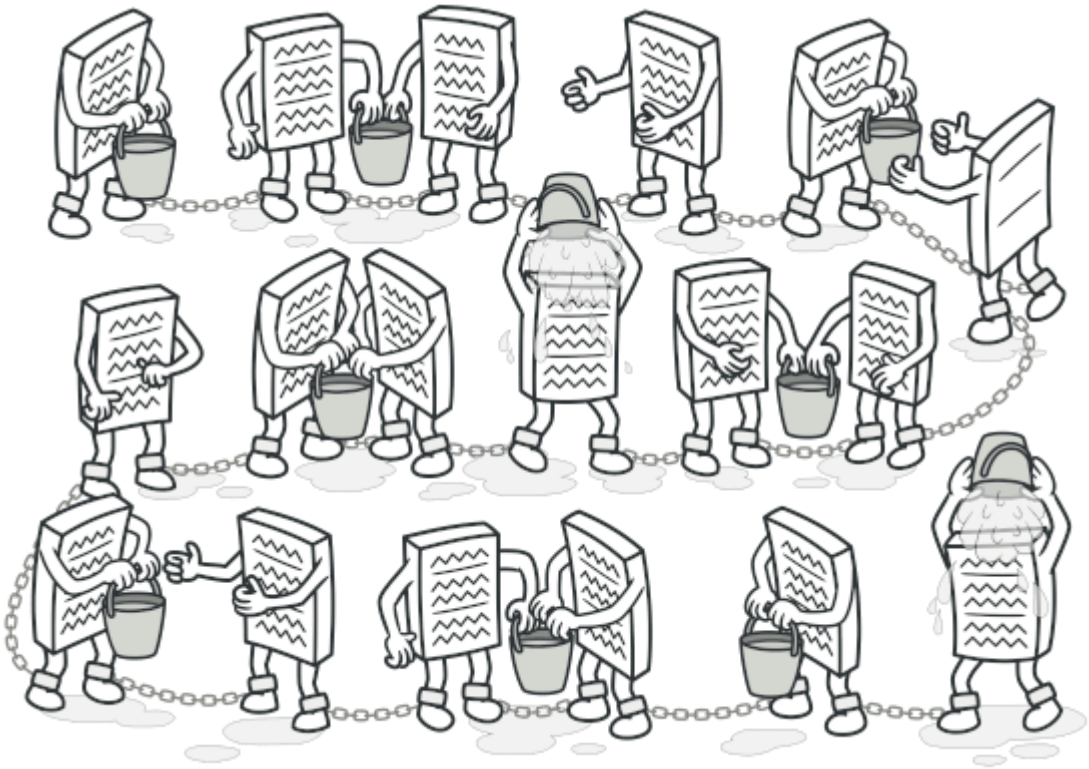
Patrones de diseño

1. Chain of Responsibility

También llamado: Cadena de responsabilidad, CoR, Chain of Command

Propósito

Chain of Responsibility es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.



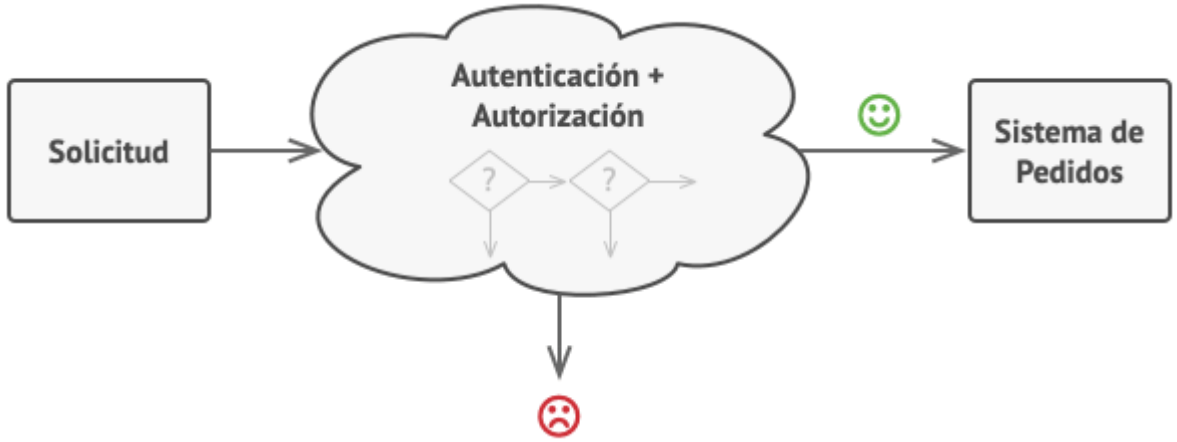
1.1 Ejemplo del libro

Problema

Imagina que estás trabajando en un sistema de pedidos online. Quieres restringir el acceso al sistema de forma que únicamente los usuarios autenticados puedan generar pedidos. Además, los usuarios que tengan permisos administrativos deben tener pleno acceso a todos los pedidos.

Tras planificar un poco, te das cuenta de que estas comprobaciones deben realizarse secuencialmente. La aplicación puede intentar autenticar a un usuario en el sistema

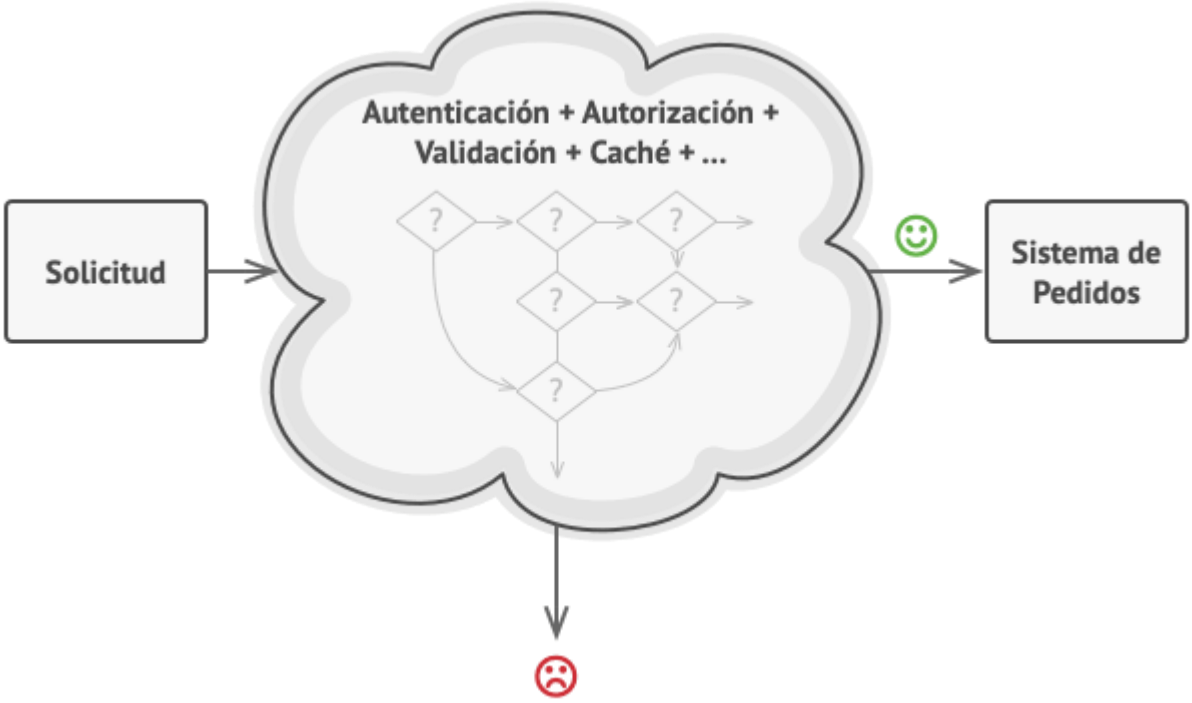
cuando reciba una solicitud que contenga las credenciales del usuario. Sin embargo, si esas credenciales no son correctas y la autenticación falla, no hay razón para proceder con otras comprobaciones.



La solicitud debe pasar una serie de comprobaciones antes de que el propio sistema de pedidos pueda gestionarla.

Durante los meses siguientes, implementas varias de esas comprobaciones secuenciales.

- Uno de tus colegas sugiere que no es seguro pasar datos sin procesar directamente al sistema de pedidos. De modo que añades un paso adicional de validación para sanear los datos de una solicitud.
- Más tarde, alguien se da cuenta de que el sistema es vulnerable al desciframiento de contraseñas por la fuerza. Para evitarlo, añades rápidamente una comprobación que filtra las solicitudes fallidas repetidas que vengan de la misma dirección IP.
- Otra persona sugiere que podrías acelerar el sistema devolviendo los resultados en caché en solicitudes repetidas que contengan los mismos datos, de modo que añades otra comprobación que permite a la solicitud pasar por el sistema únicamente cuando no hay una respuesta adecuada en caché.



Cuanto más crece el código, más se complica.

El código de las comprobaciones, que ya se veía desordenado, se vuelve más y más abotargado cada vez que añades una nueva función. En ocasiones, un cambio en una comprobación afecta a las demás. Y lo peor de todo es que, cuando intentas reutilizar las comprobaciones para proteger otros componentes del sistema, tienes que duplicar parte del código, ya que esos componentes necesitan parte de las comprobaciones, pero no todas ellas.

El sistema se vuelve muy difícil de comprender y costoso de mantener. Luchas con el código durante un tiempo hasta que un día decides refactorizarlo todo.

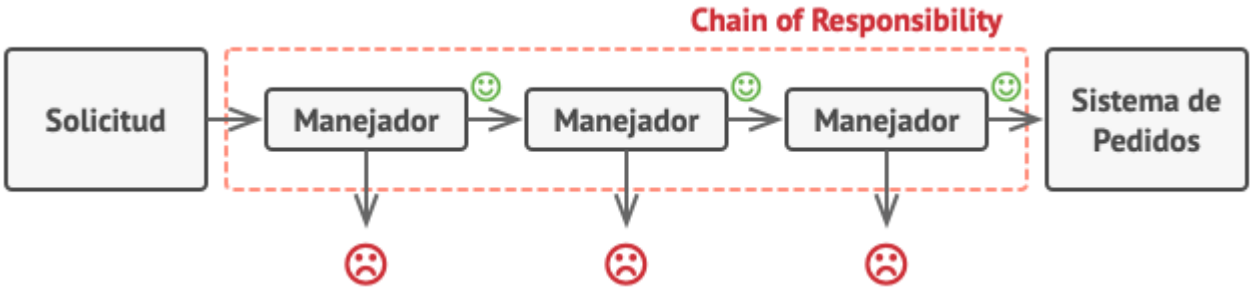
Solución

Al igual que muchos otros patrones de diseño de comportamiento, el **Chain of Responsibility** se basa en transformar comportamientos particulares en objetos autónomos llamados *manejadores*. En nuestro caso, cada comprobación debe ponerse dentro de su propia clase con un único método que realice la comprobación. La solicitud, junto con su información, se pasa a este método como argumento.

El patrón sugiere que vincules esos manejadores en una cadena. Cada manejador vinculado tiene un campo para almacenar una referencia al siguiente manejador de la cadena. Además de procesar una solicitud, los manejadores la pasan a lo largo de la cadena. La solicitud viaja por la cadena hasta que todos los manejadores han tenido la oportunidad de procesarla.

Y ésta es la mejor parte: un manejador puede decidir no pasar la solicitud más allá por la cadena y detener con ello el procesamiento.

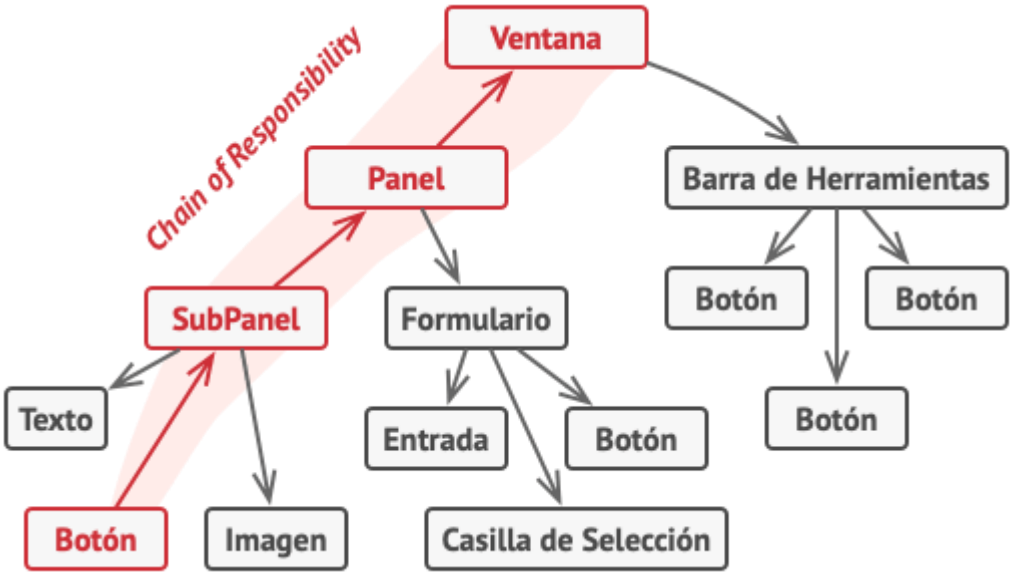
En nuestro ejemplo de los sistemas de pedidos, un manejador realiza el procesamiento y después decide si pasa la solicitud al siguiente eslabón de la cadena. Asumiendo que la solicitud contiene la información correcta, todos los manejadores pueden ejecutar su comportamiento principal, ya sean comprobaciones de autenticación o almacenamiento en la memoria caché.



Los manejadores se alinean uno tras otro, formando una cadena.

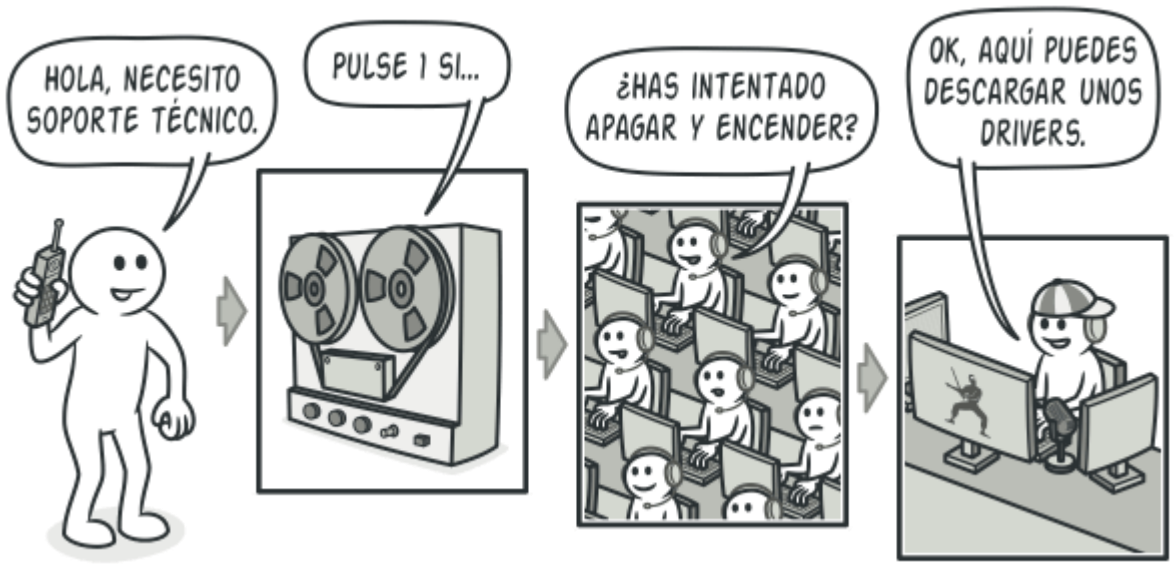
No obstante, hay una solución ligeramente diferente (y un poco más estandarizada) en la que, al recibir una solicitud, un manejador decide si puede procesarla. Si puede, no pasa la solicitud más allá. De modo que un único manejador procesa la solicitud o no lo hace ninguno en absoluto. Esta solución es muy habitual cuando tratamos con eventos en pilas de elementos dentro de una interfaz gráfica de usuario (GUI).

Por ejemplo, cuando un usuario hace clic en un botón, el evento se propaga por la cadena de elementos GUI que comienza en el botón, recorre sus contenedores (como formularios o paneles) y acaba en la ventana principal de la aplicación. El evento es procesado por el primer elemento de la cadena que es capaz de gestionarlo. Este ejemplo también es destacable porque muestra que siempre se puede extraer una cadena de un árbol de objetos.



Una cadena puede formarse a partir de una rama de un árbol de objetos.
Es fundamental que todas las clases manejadoras implementen la misma interfaz. Cada manejadora concreta solo debe preocuparse por la siguiente que cuente con el método `ejecutar`. De esta forma puedes componer cadenas durante el tiempo de ejecución, utilizando varios manejadores sin acoplar tu código a sus clases concretas.

Analogía en el mundo real



Una llamada al soporte técnico puede pasar por muchos operadores.
Acabas de comprar e instalar una nueva pieza de hardware en tu computadora. Como eres un fanático de la informática, la computadora tiene varios sistemas operativos instalados. Intentas arrancarlos todos para ver si soportan el hardware. Windows detecta y habilita el hardware automáticamente. Sin embargo, tu querido Linux se niega a funcionar con el nuevo hardware. Ligeramente esperanzado, decides llamar al número de teléfono de soporte técnico escrito en la caja.

Lo primero que oyes es la voz robótica del contestador automático. Te sugiere nueve soluciones populares a varios problemas, pero ninguna de ellas es relevante a tu caso. Después de un rato, el robot te conecta con un operador humano.

Por desgracia, el operador tampoco consigue sugerirte nada específico. Se dedica a recitar largos pasajes del manual, negándose a escuchar tus comentarios. Cuando escuchas por enésima vez la frase “¿has intentado apagar y encender la computadora?”, exiges que te pasen con un ingeniero de verdad.

Por fin, el operador pasa tu llamada a unos de los ingenieros, que probablemente ansiaba una conversación humana desde hacía tiempo, sentado en la solitaria sala del servidor del oscuro sótano de un edificio de oficinas. El ingeniero te indica dónde descargar los drivers adecuados para tu nuevo hardware y cómo instalarlos en Linux. Por fin, ¡la solución! Acabas la llamada dando saltos de alegría.

1.2 Ejemplo de la vida real

La cadena de responsabilidad es una notificación de forma serial y es el objeto el que determina si se realiza la acción.

En un banco se requiere notificar si el monto de las operaciones realizadas supera un limite durante un tiempo determinado.

El cliente puede tener diferentes tipos de operaciones y vínculos con el banco por lo tanto se realiza una proceso que vaya preguntando a cada sistema si el monto de ese tipo de operación sumado con el monto de la operación superan el monto que se esta solicitando

El modulo de notificación esta en cada área de operaciones del banco e interactúan recibiendo el monto de notificación, el monto del proceso anterior.

Una vez calculado el monto del área y sumado al monto recibido se determina si supera el monto notificado y de ser así se notifica y para la respuesta.

1.3 Código

Uso del patrón en Java

Complejidad:

Popularidad:

Ejemplos de uso: El patrón Chain of Responsibility no es un invitado habitual en el programa Java, ya que tan solo es relevante cuando el código opera con cadenas de objetos.

Uno de los casos de uso más populares para el patrón es la propagación de eventos a los componentes padre (*bubbling*) de las clases GUI. Otro caso de uso notable son los filtros de acceso secuencial.

Aquí tienes algunos ejemplos del patrón en las principales bibliotecas Java:

- `javax.servlet.Filter#doFilter()`
- `java.util.logging.Logger#log()`

Identificación: El patrón es reconocible porque los métodos de comportamiento de un grupo de objetos invocan indirectamente los mismos métodos en otros objetos, mientras que todos los objetos siguen la interfaz común.

Acceso filtrado

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

```
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

```
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.RoleCheckMiddle
ware;
import
refactoring_guru.chain_of_responsibility.example.middleware.ThrottlingMiddl
eware;
import
refactoring_guru.chain_of_responsibility.example.middleware.UserExistsMiddl
eware;
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(System.in));
    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!
```

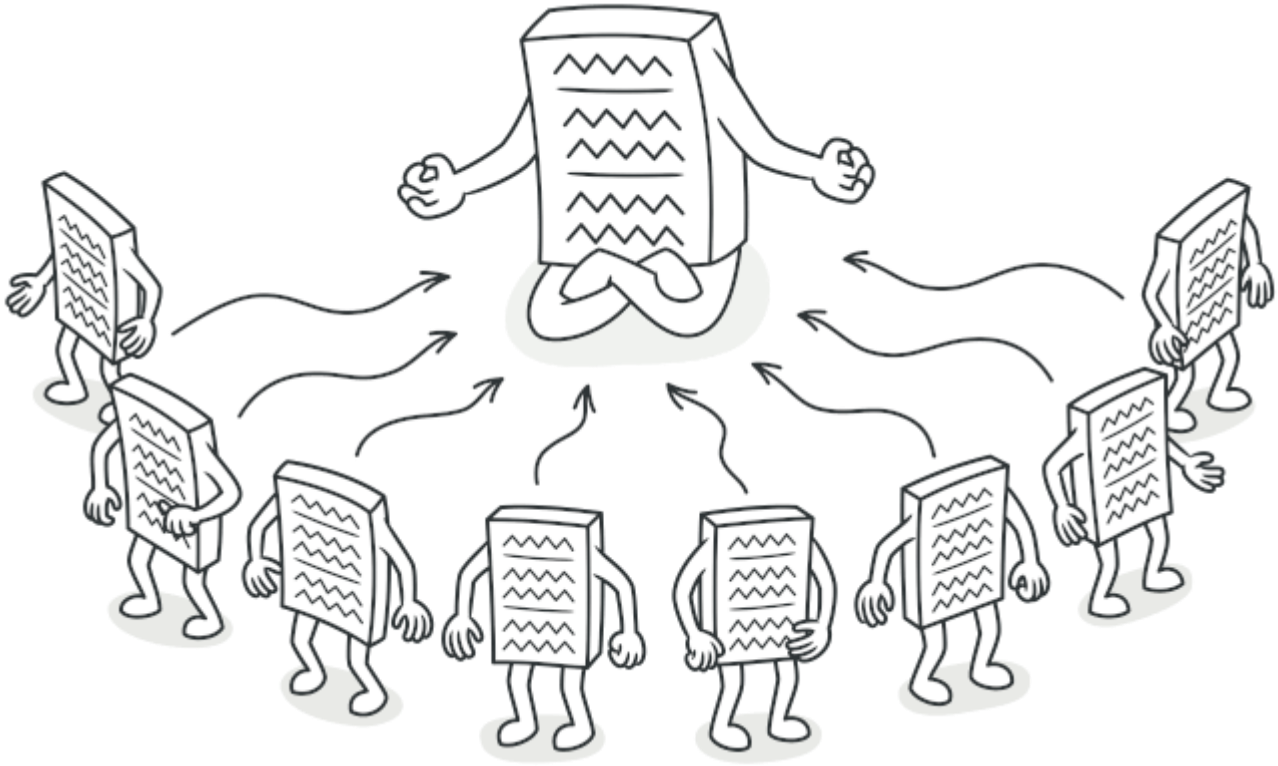
2. Patron Singleton

Singleton

También llamado: Instancia única

Propósito

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.



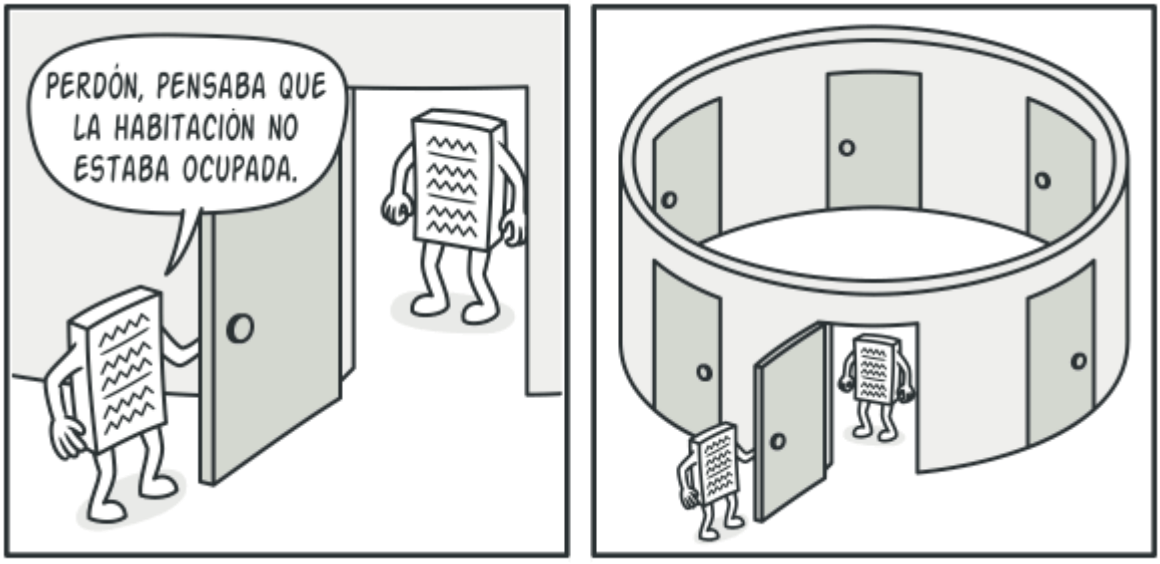
Problema

El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el *Principio de responsabilidad única*:

1. **Garantizar que una clase tenga una única instancia**. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.

   Funciona así: imagina que has creado un objeto y al cabo de un tiempo decides crear otro nuevo. En lugar de recibir un objeto nuevo, obtendrás el que ya habías creado.

Ten en cuenta que este comportamiento es imposible de implementar con un constructor normal, ya que una llamada al constructor siempre **debe** devolver un nuevo objeto por diseño.



Puede ser que los clientes ni siquiera se den cuenta de que trabajan con el mismo objeto todo el tiempo.

2. **Proporcionar un punto de acceso global a dicha instancia**. ¿Recuerdas esas variables globales que utilizaste (bueno, sí, fui yo) para almacenar objetos esenciales? Aunque son muy útiles, también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.

   Al igual que una variable global, el patrón Singleton nos permite acceder a un objeto desde cualquier parte del programa. No obstante, también evita que otro código sobreescriba esa instancia.

   Este problema tiene otra cara: no queremos que el código que resuelve el primer problema se encuentre disperso por todo el programa. Es mucho más conveniente tenerlo dentro de una clase, sobre todo si el resto del código ya depende de ella.

Hoy en día el patrón Singleton se ha popularizado tanto que la gente suele llamar *singleton* a cualquier patrón, incluso si solo resuelve uno de los problemas antes mencionados.

Solución

Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador `new` con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

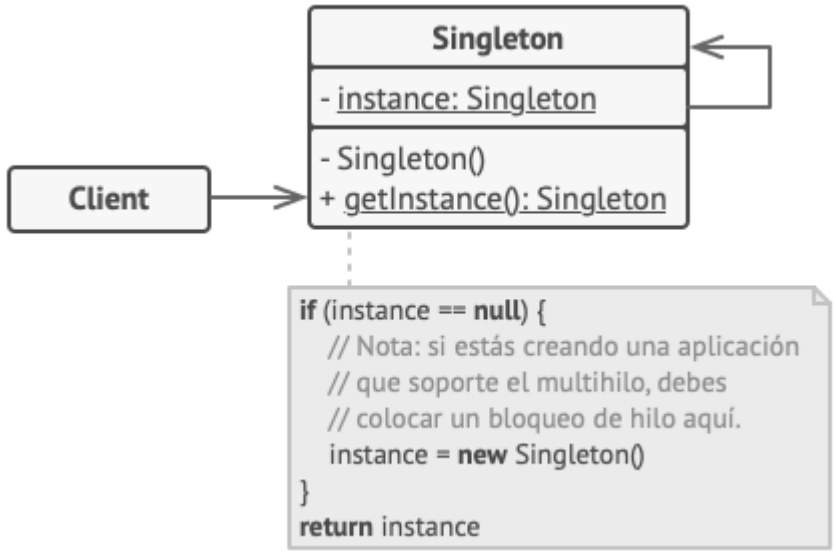
Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

Analogía en el mundo real

2.1 Ejemplo del Libro

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.

Estructura



1. La clase **Singleton** declara el método estático `obtenerInstancia` que devuelve la misma instancia de su propia clase.

   El constructor del Singleton debe ocultarse del código cliente. La llamada al método `obtenerInstancia` debe ser la única manera de obtener el objeto de Singleton.

Pseudocódigo

En este ejemplo, la clase de conexión de la base de datos actúa como **Singleton**. Esta clase no tiene un constructor público, por lo que la única manera de obtener su objeto

es invocando el método `obtenerInstancia`. Este método almacena en caché el primer objeto creado y lo devuelve en todas las llamadas siguientes.

```
// La clase Base de datos define el método `obtenerInstancia`
// que permite a los clientes acceder a la misma instancia de
// una conexión de la base de datos a través del programa.
class Database is
    // El campo para almacenar la instancia singleton debe
    // declararse estático.
    private static field instance: Database

    // El constructor del singleton siempre debe ser privado
    // para evitar llamadas de construcción directas con el
    // operador `new`.
    private constructor Database() is
        // Algún código de inicialización, como la propia
        // conexión al servidor de una base de datos.
        // ...

    // El método estático que controla el acceso a la instancia
    // singleton.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                // Garantiza que la instancia aún no se ha
                // inicializado por otro hilo mientras ésta ha
                // estado esperando el desbloqueo.
                if (Database.instance == null) then
                    Database.instance = new Database()
        return Database.instance

    // Por último, cualquier singleton debe definir cierta
    // lógica de negocio que pueda ejecutarse en su instancia.
    public method query(sql) is
        // Por ejemplo, todas las consultas a la base de datos
        // de una aplicación pasan por este método. Por lo
        // tanto, aquí puedes colocar lógica de regularización
        // (throttling) o de envío a la memoria caché.
        // ...

class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // La variable `bar` contendrá el mismo objeto que la
        // variable `foo`.
```

2.2 Ejemplo de la vida real.

En las conexiones a base de datos, siempre se pide la creación de una instancia de base de datos, en cambio de que se cree una nueva instancia se puede tener una sola si se tiene un patrón que pregunte si ya se tiene una instancia.