

2024

# PATRONES DE DISEÑO

ANDREU ORENGA RAMON

ENTORNOS DE DESARROLLO

1ºDAW

IES BEnigasló

## Contenido

Patrón Builder.....	3
EJEMPLO DE IMPLEMENTACIÓN DEL PATRON BUILDER.....	3
PASO 1: DEFINIR LA INTERFAZ AUTOMOVILBUILDER .....	3
PASO 2: CREAR UN CONCRETEBUILDER.....	3
PASO 3: DEFINIR EL DIRECTOR .....	4
PASO 4: DEFINIR EL PRODUCTO (AUTOMOVIL) .....	4
PASO 5: USAR EL PATRÓN BUILDER PARA CONSTRUIR EL AUTOMÓVIL .....	5
EXPANSIÓN DEL EJEMPLO DEL AUTOMÓVIL .....	6
PASO 6: AÑADIR MÁS CARACTERÍSTICAS AL AUTOMOVILBUILDER .....	6
PASO 7: IMPLEMENTAR LAS NUEVAS CARACTERÍSTICAS EN SEDANBUILDER .....	6
PASO 8: MODIFICAR CONCESIONARIO PARA USAR LAS NUEVAS OPCIONES .....	7
PASO 9: ACTUALIZAR LA CLASE AUTOMOVIL.....	7
PASO 10: DEMOSTRACIÓN EN MAIN .....	8
Patrón Callback.....	9
Definir la interfaz de Callback .....	9
Crear una clase que usa la interfaz de Callback.....	9
Uso del patrón Callback en la clase principal .....	10
Funcionamiento del ejemplo .....	10
Patrón Factory .....	11
Definir la interfaz de Factory .....	11
Crear una clase que usa la interfaz de Factory.....	11
Uso del patrón Factory en la clase principal .....	12
Funcionamiento del ejemplo .....	12
Patrón Observer .....	13
Definir la interfaz de Observer.....	13
Crear una clase que usa la interfaz de Observer .....	13
Uso del patrón Observer en la clase principal.....	14
Funcionamiento del ejemplo .....	14
Patrón Singleton .....	15
Definir la clase Singleton .....	15
Crear una clase que usa la instancia Singleton.....	15
Uso del patrón Singleton en la clase principal .....	15
Funcionamiento del ejemplo .....	16
Patrón Adapter .....	16

Definir la interfaz de Adapter.....	16
Crear una clase que usa la interfaz de Adapter .....	17
Uso del patrón Adapter en la clase principal.....	17
Funcionamiento del ejemplo .....	18

# Patrones de diseño

## Patrón Builder

### EJEMPLO DE IMPLEMENTACIÓN DEL PATRON BUILDER

#### PASO 1: DEFINIR LA INTERFAZ AUTOMOVILBUILDER

Primero, definiremos una interfaz AutomovilBuilder que especifica los métodos para construir las diferentes partes del automóvil.

```
Actividades_PD > Actividad_PatronBuilder > src > AutomovilBuilder.java
1  interface AutomovilBuilder {
2      void buildMotor();
3      void buildAsientos(int numeroAsientos);
4      void buildColor(String color);
5      Automovil obtenerAutomovil();
6  }
7
```

#### PASO 2: CREAR UN CONCRETEBUILDER

Ahora, crearemos una clase concreta que implementa AutomovilBuilder. Esta clase será responsable de construir un tipo específico de automóvil, por ejemplo, un Sedán.

```
Actividades_PD > Actividad_PatronBuilder > src > SedanBuilder.java > ...
1  class SedanBuilder implements AutomovilBuilder {
2      private Automovil automovil;
3
4      public SedanBuilder() {
5          this.automovil = new Automovil();
6      }
7
8      @Override
9      public void buildMotor() {
10         automovil.setMotor("Motor de 4 cilindros");
11     }
12
13     @Override
14     public void buildAsientos(int numeroAsientos) {
15         automovil.setAsientos(numeroAsientos);
16     }
17
18     @Override
19     public void buildColor(String color) {
20         automovil.setColor(color);
21     }
22
23     @Override
24     public Automovil obtenerAutomovil() {
25         return automovil;
26     }
27 }
28
```

### PASO 3: DEFINIR EL DIRECTOR

El Director se encarga de construir un automóvil usando el builder proporcionado, asegurándose de que los pasos se ejecuten en orden.

```
Actividades_PD > Actividad_PatronBuilder > src > Concesionario.java > ...
1  class Concesionario {
2      public Automovil construirAutomovil(AutomovilBuilder builder) {
3          builder.buildMotor();
4          builder.buildAsientos(numeroAsientos:5); // Número estándar de asientos para un Sedán
5          builder.buildColor(color:"Rojo");
6          return builder.obtenerAutomovil();
7      }
8  }
9  |
```

### PASO 4: DEFINIR EL PRODUCTO (AUTOMOVIL)

La clase Automovil representa el producto final que está siendo construido.

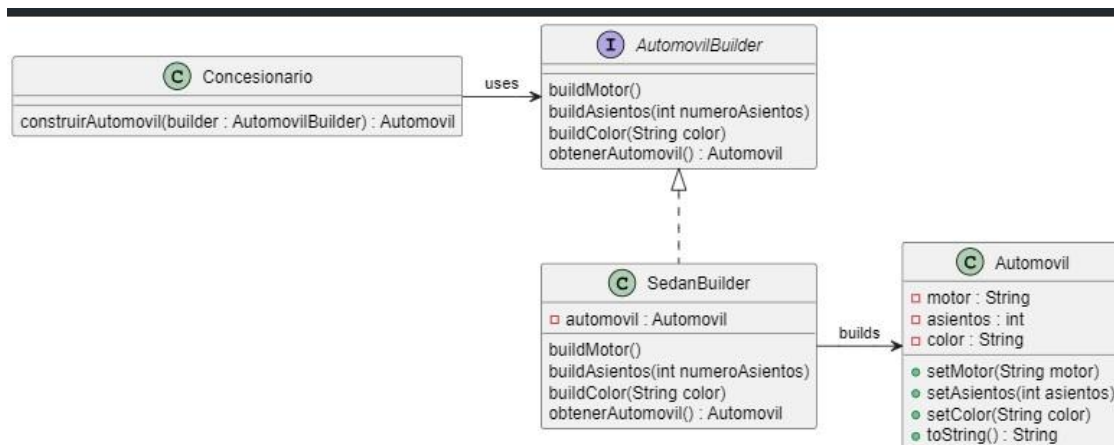
```
Actividades_PD > Actividad_PatronBuilder > src > Automovil.java > ...
1  class Automovil {
2      private String motor;
3      private int asientos;
4      private String color;
5
6      public void setMotor(String motor) {
7          this.motor = motor;
8      }
9
10     public void setAsientos(int asientos) {
11         this.asientos = asientos;
12     }
13
14     public void setColor(String color) {
15         this.color = color;
16     }
17
18     @Override
19     public String toString() {
20         return "Automovil{" +
21             "motor='" + motor + '\'' +
22             ", asientos=" + asientos +
23             ", color='" + color + '\'' +
24             '}';
25     }
26 }
27
```

## PASO 5: USAR EL PATRÓN BUILDER PARA CONSTRUIR EL AUTOMÓVIL

Finalmente, utilizamos nuestro patrón Builder en la aplicación principal para crear un automóvil.

```
Actividades_PD > Actividad_PatronBuilder > src > Main.java > ...
1  public class Main {
2      public static void main(String[] args) {
3          Concesionario concesionario = new Concesionario();
4          AutomovilBuilder builder = new SedanBuilder();
5          Automovil miAutomovil = concesionario.construirAutomovil(builder);
6          System.out.println(miAutomovil);
7      }
8  }
9
10
11
```

### DIAGRAMA DEL PATRON BUILDER DE AUTOMOVIL



## EXPANSIÓN DEL EJEMPLO DEL AUTOMÓVIL

Podemos añadir más funcionalidades al patrón Builder para hacerlo más flexible y poderoso. Por ejemplo, podríamos querer permitir más personalización en los tipos de motor o incluso añadir nuevas características como un sistema de entretenimiento o características de seguridad. Vamos a añadir estas opciones a nuestro patrón Builder.

### PASO 6: AÑADIR MÁS CARACTERÍSTICAS AL AUTOMOVILBUILDER

Ampliemos la interfaz AutomovilBuilder para incluir métodos adicionales que permitan más personalización:

```
dades_PD > Actividad_PatronBuilder > src > AutomovilBuilder.java > ...  
interface AutomovilBuilder {  
    void buildMotor(String tipoMotor);  
    void buildAsientos(int numeroAsientos);  
    void buildColor(String color);  
    void buildSistemaEntretenimiento(boolean tiene);  
    void buildPaqueteSeguridad(String paquete);  
    Automovil obtenerAutomovil();  
}
```

### PASO 7: IMPLEMENTAR LAS NUEVAS CARACTERÍSTICAS EN SEDANBUILDER

Actualizamos nuestra clase concreta SedanBuilder para manejar estas nuevas opciones:

```
dades_PD > Actividad_PatronBuilder > src > SedanBuilder.java > ...  
class SedanBuilder implements AutomovilBuilder {  
    private Automovil automovil;  
  
    public SedanBuilder() {  
        this.automovil = new Automovil();  
    }  
  
    @Override  
    public void buildMotor() {  
        automovil.setMotor("Motor de 4 cilindros");  
    }  
  
    @Override  
    public void buildAsientos(int numeroAsientos) {  
        automovil.setAsientos(numeroAsientos);  
    }  
  
    @Override  
    public void buildColor(String color) {  
        automovil.setColor(color);  
    }  
  
    @Override  
    public Automovil obtenerAutomovil() {  
        return automovil;  
    }  
}
```

## PASO 8: MODIFICAR CONCESIONARIO PARA USAR LAS NUEVAS OPCIONES

Ajustamos el Director para que pueda construir automóviles utilizando las nuevas características disponibles:

```
class Concesionario {
    AutomovilBuilder builder = new AutomovilBuilder();

    public Automovil construirAutomovil() {
        builder.buildMotor(tipoMotor: "Motor V6");
        builder.buildAsientos(numeroAsientos: 5);
        builder.buildColor(color: "Azul Metálico");
        builder.buildSistemaEntretenimiento(tiene: true);
        builder.buildPaqueteSeguridad(paquete: "Avanzado");
        return builder.obtenerAutomovil();
    }
}
```

## PASO 9: ACTUALIZAR LA CLASE AUTOMOVIL

Asegurémonos de que la clase Automovil puede manejar estas nuevas características:

```
class Automovil {
    private String motor;
    private int asientos;
    private String color;
    private boolean sistemaEntretenimiento;
    private String paqueteSeguridad;

    // Setters
    public void setMotor(String motor) {
        this.motor = motor;
    }

    public void setAsientos(int asientos) {
        this.asientos = asientos;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void setSistemaEntretenimiento(boolean sistemaEntretenimiento) {
        this.sistemaEntretenimiento = sistemaEntretenimiento;
    }

    public void setPaqueteSeguridad(String paqueteSeguridad) {
        this.paqueteSeguridad = paqueteSeguridad;
    }

    // Getters
    public String getMotor() {
        return motor;
    }

    public int getAsientos() {
        return asientos;
    }

    public String getColor() {
        return color;
    }

    public boolean isSistemaEntretenimiento() {
        return sistemaEntretenimiento;
    }

    public String getPaqueteSeguridad() {
        return paqueteSeguridad;
    }
}
```



## PASO 10: DEMOSTRACIÓN EN MAIN

Finalmente, actualizamos el método main para demostrar las capacidades expandidas:

```
des_PD > Actividad_PatronBuilder > src > Main.java
public class Main {
    public static void main(String[] args) {
        Concesionario concesionario = new Concesionario();
        AutomovilBuilder builder = new SedanBuilder();
        Automovil miAutomovil = concesionario.construirAutomovil(builder);
        System.out.println(miAutomovil);
    }
}
```

### DIAGRAMA UML CON LAS MODIFICACIONES



## Patrón Callback

### Definir la interfaz de Callback

Primero, se define una interfaz Retrollamada que contendrá el método llamar(). Este método será llamado una vez que una operación específica haya terminado.

```
package src;  
public interface Retrollamada {  
    void llamar();  
}
```

### Crear una clase que usa la interfaz de Callback

Luego, se crea una clase que utilizará esta interfaz. MiClase realiza alguna operación que puede llevar tiempo y utiliza el método llamar de la interfaz Retrollamada para notificar cuando ha finalizado la operación.

```
package src;  
public class MiClase {  
    private Retrollamada retrollamada;  
  
    public MiClase(Retrollamada retrollamada) {  
        this.retrollamada = retrollamada;  
    }  
  
    public void ejecutar() {  
        // Simula una operación que toma tiempo  
        try {  
            Thread.sleep(1000); // Simula tiempo de espera  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        }  
  
        // Llama al método de callback una vez completada la operación  
        retrollamada.llamar();  
    }  
}
```

## Uso del patrón Callback en la clase principal

Finalmente, en la clase principal, se instancia 'MiClase' pasándole una implementación de la interfaz 'Retrollamada', que en este caso, imprime un mensaje al ser llamada.

```
package src;

public class Principal {
    public static void main(String[] args) {
        MiClase miClase = new MiClase(new Retrollamada() {
            @Override
            public void llamar() {
                System.out.println("¡La operación ha terminado!");
            }
        });

        miClase.ejecutar();
    }
}
```

## Funcionamiento del ejemplo

**Definición de Retrollamada:** Esta interfaz es crucial porque define cómo se comunicarán las clases de manera asíncrona.

**MiClase implementa la operación:** Esta clase realiza la operación que puede demorar y luego llama al método `llamar()` del objeto `Retrollamada` que se le pasa en el constructor. Esto desacopla el tiempo de ejecución de la operación de la respuesta a esa operación.

**Uso en Principal:** Aquí se muestra cómo instanciar `MiClase` y cómo manejar el callback sin bloquear otras operaciones, permitiendo que el hilo principal continúe con otras tareas o maneje otros eventos.

Este ejemplo ilustra cómo el patrón Callback permite una mayor flexibilidad y mejor respuesta en aplicaciones donde no se desea bloquear el flujo principal de ejecución, como en interfaces de usuario o en operaciones de I/O.

## Patrón Factory

### Definir la interfaz de Factory

Primero, se define una interfaz Producto que contendrá métodos comunes para todos los productos que se van a crear. Luego, se define una interfaz Factory que contendrá el método crearProducto(). Este método será llamado para crear instancias de los productos.

```
1 // Interfaz Producto
2 public interface Producto {
3
4     void operacion();
5 }
6 // Interfaz Factory
7 public interface Factory {
8     Producto crearProducto();
9 }
10
```

### Crear una clase que usa la interfaz de Factory

Luego, se crean clases concretas que implementarán la interfaz Producto. También se crean fábricas concretas que implementarán la interfaz Factory y crearán instancias de los productos concretos.

```
1 // Clase concreta que implementa Producto
2 public class ProductoConcretoA implements Producto {
3     @Override
4     public void operacion() {
5         System.out.println("Operación de Producto Concreto A");
6     }
7 }
8
9 // Clase concreta que implementa Producto
10 public class ProductoConcretoB implements Producto {
11     @Override
12     public void operacion() {
13         System.out.println("Operación de Producto Concreto B");
14     }
15 }
16
17 // Fábrica concreta que crea ProductoConcretoA
18 public class FactoryConcretoA implements Factory {
19     @Override
20     public Producto crearProducto() {
21         return new ProductoConcretoA();
22     }
23 }
24
25 // Fábrica concreta que crea ProductoConcretoB
26 public class FactoryConcretoB implements Factory {
27     @Override
28     public Producto crearProducto() {
29         return new ProductoConcretoB();
30     }
31 }
```

## Uso del patrón Factory en la clase principal

Finalmente, en la clase principal, se instancia la fábrica concreta correspondiente y se usa para crear productos. Luego, se llama al método `operacion()` de los productos creados.



```
1
2 public class Cliente {
3     public static void main(String[] args) {
4         // Usar FactoryConcretoA para crear ProductoConcretoA
5         Factory factoryA = new FactoryConcretoA();
6         Producto productoA = factoryA.crearProducto();
7         productoA.operacion();
8
9         // Usar FactoryConcretoB para crear ProductoConcretoB
10        Factory factoryB = new FactoryConcretoB();
11        Producto productoB = factoryB.crearProducto();
12        productoB.operacion();
13    }
14 }
15
```

## Funcionamiento del ejemplo

**Definición de Producto y Factory:** Estas interfaces son cruciales porque definen los métodos comunes que deben implementar los productos y las fábricas.

**Clases concretas de Producto y Factory:** Estas clases implementan las interfaces definidas y proporcionan las funcionalidades específicas de los productos y las fábricas.

**Uso en Cliente:** Aquí se muestra cómo instanciar las fábricas concretas y cómo crear productos a través de ellas sin necesidad de conocer las clases concretas de los productos. Esto permite desacoplar el código de creación de objetos del código de uso de los mismos.

Este ejemplo ilustra cómo el patrón Factory permite una mayor flexibilidad y mejor organización en aplicaciones donde se requiere crear instancias de varias clases concretas derivadas de una misma interfaz. Es especialmente útil en escenarios donde el proceso de creación de objetos es complejo o varía dependiendo de la situación.

## Patrón Observer

### Definir la interfaz de Observer

Primero, se define una interfaz Observer que contendrá el método actualizar(). Esta interfaz será implementada por todas las clases que deben ser notificadas de cambios en el estado de otro objeto. Luego, se define una interfaz Observable que contendrá métodos para agregar, remover y notificar observadores.

```
1 // Interfaz Observer
2 public interface Observer {
3     void actualizar();
4 }
5
6 // Interfaz Observable
7 public interface Observable {
8     void agregarObservador(Observer o);
9     void removerObservador(Observer o);
10    void notificarObservadores();
11 }
12
```

### Crear una clase que usa la interfaz de Observer

Luego, se crea una clase concreta que implementa la interfaz Observable. Esta clase gestionará una lista de observadores y los notificará cuando ocurra un cambio en su estado. También se crean clases concretas que implementan la interfaz Observer y reaccionan a las notificaciones del Observable.

```
1 package src_Patron_Observer;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class SujetoConcreto {
7
8     private List<Observer> observadores;
9     private String estado;
10
11     public SujetoConcreto() {
12         this.observadores = new ArrayList<>();
13     }
14
15     public void setEstado(String estado) {
16         this.estado = estado;
17         notificarObservadores();
18     }
19
20     public String getEstado() {
21         return estado;
22     }
23
24     @Override
25     public void agregarObservador(Observer o) {
26         observadores.add(o);
27     }
28
29     @Override
30     public void removerObservador(Observer o) {
31         observadores.remove(o);
32     }
33
34     @Override
35     public void notificarObservadores() {
36         for (Observer observador : observadores) {
37             observador.actualizar();
38         }
39     }
40 }
41
42 // Clase concreta que implementa Observer
43 public class ObservadorConcreto implements Observer {
44     private SujetoConcreto sujeto;
45
46     public ObservadorConcreto(SujetoConcreto sujeto) {
47         this.sujeto = sujeto;
48         this.sujeto.agregarObservador(this);
49     }
50
51     @Override
52     public void actualizar() {
53         System.out.println("El estado del sujeto ha cambiado a: " + sujeto.getEstado());
54     }
55 }
```

## Uso del patrón Observer en la clase principal

Finalmente, en la clase principal, se instancia el sujeto concreto y varios observadores. Luego, se cambia el estado del sujeto y se observa cómo los observadores reaccionan a este cambio.



```
1 public class Cliente {
2     public static void main(String[] args) {
3         SujetoConcreto sujeto = new SujetoConcreto();
4
5         ObservadorConcreto observador1 = new ObservadorConcreto(sujeto);
6         ObservadorConcreto observador2 = new ObservadorConcreto(sujeto);
7
8         sujeto.setEstado("Estado 1");
9         sujeto.setEstado("Estado 2");
10    }
11 }
```

## Funcionamiento del ejemplo

**Definición de Observer y Observable:** Estas interfaces son cruciales porque definen cómo los observadores se registran y son notificados de los cambios en el estado del sujeto.

**Clase concreta de Observable:** SujetoConcreto gestiona una lista de observadores y los notifica cuando su estado cambia.

**Clase concreta de Observer:** ObservadorConcreto reacciona a los cambios en el estado del sujeto cuando es notificado.

**Uso en Cliente:** Aquí se muestra cómo instanciar el sujeto y los observadores, cómo registrar los observadores y cómo cambiar el estado del sujeto para ver cómo los observadores son notificados y reaccionan.

Este ejemplo ilustra cómo el patrón Observer permite la comunicación entre objetos en un sistema de manera que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente. Esto es útil en aplicaciones donde es necesario mantener la consistencia entre objetos relacionados sin acoplar fuertemente sus clases.

# Patrón Singleton

## Definir la clase Singleton

Primero, se define una clase Singleton que asegura que solo una instancia de la clase puede ser creada. Esto se logra mediante un constructor privado y un método estático que devuelve la única instancia de la clase.

```
1 public class Singleton {
2     // La única instancia de la clase Singleton
3     private static Singleton instancia;
4
5     // Constructor privado para evitar instanciación externa
6     private Singleton() {}
7
8     // Método estático para obtener la única instancia de la clase
9     public static Singleton getInstance() {
10         if (instancia == null) {
11             instancia = new Singleton();
12         }
13         return instancia;
14     }
15
16     // Ejemplo de un método de la clase Singleton
17     public void operacion() {
18         System.out.println("Operación de la instancia Singleton");
19     }
20 }
21
```

## Crear una clase que usa la instancia Singleton

En este patrón, generalmente no se necesita una clase específica que use la instancia Singleton, ya que cualquier clase puede utilizarla directamente llamando al método `getInstance()`.

## Uso del patrón Singleton en la clase principal

Finalmente, en la clase principal, se accede a la instancia Singleton y se llama a sus métodos. Esto demuestra cómo se garantiza que solo una instancia de la clase es utilizada en toda la aplicación.

```
1 public class Cliente {
2     public static void main(String[] args) {
3         // Obtener la única instancia de Singleton
4         Singleton singleton1 = Singleton.getInstance();
5         singleton1.operacion();
6
7         // Intentar obtener otra instancia de Singleton
8         Singleton singleton2 = Singleton.getInstance();
9         singleton2.operacion();
10
11         // Verificar que ambas referencias apuntan al mismo objeto
12         if (singleton1 == singleton2) {
13             System.out.println("Ambas referencias apuntan a la misma instancia Singleton");
14         } else {
15             System.out.println("Las referencias apuntan a diferentes instancias Singleton");
16         }
17     }
18 }
19
```



## Funcionamiento del ejemplo

Definición de Singleton: La clase Singleton contiene un constructor privado y un método estático `getInstancia()` que asegura que solo una instancia de la clase puede ser creada. La instancia se crea solo cuando es necesaria y se guarda en una variable estática.

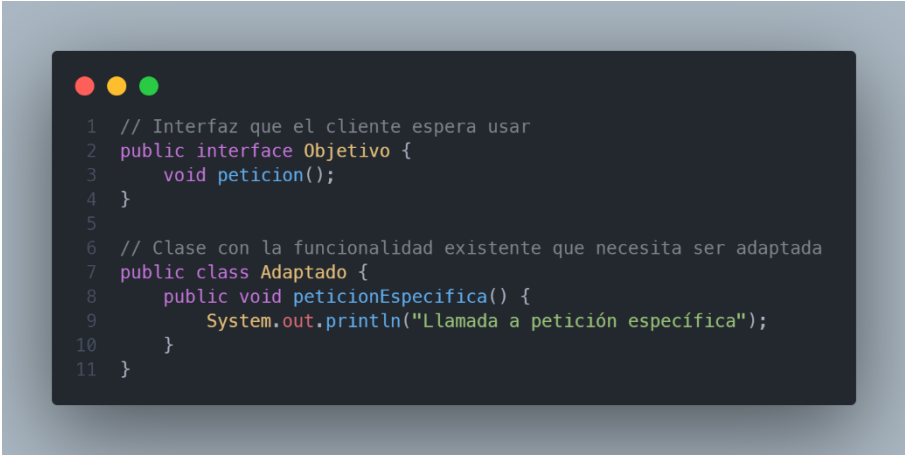
Uso en Cliente: Aquí se muestra cómo acceder a la instancia Singleton y cómo asegurarse de que solo una instancia es utilizada en toda la aplicación. Se llama al método `operacion()` de la instancia Singleton y se verifica que todas las referencias apuntan al mismo objeto.

Este ejemplo ilustra cómo el patrón Singleton garantiza que solo una instancia de una clase exista y proporcione un punto global de acceso a ella. Es útil en escenarios donde se necesita un control estricto sobre la instancia única, como en la configuración global de una aplicación o en la gestión de recursos compartidos.

## Patrón Adapter

### Definir la interfaz de Adapter

Primero, se define una interfaz `Objetivo` que declara el método que el cliente espera usar. Luego, se define una clase `Adaptado` que contiene la funcionalidad existente que necesita ser adaptada.



```
1 // Interfaz que el cliente espera usar
2 public interface Objetivo {
3     void peticion();
4 }
5
6 // Clase con la funcionalidad existente que necesita ser adaptada
7 public class Adaptado {
8     public void peticionEspecifica() {
9         System.out.println("Llamada a petición específica");
10    }
11 }
```

## Crear una clase que usa la interfaz de Adapter

Luego, se crea una clase Adaptador que implementa la interfaz Objetivo y traduce las llamadas al método del Adaptado.

```
1 // Clase Adaptador que implementa la interfaz Objetivo
2 public class Adaptador implements Objetivo {
3     private Adaptado adaptado;
4
5     public Adaptador(Adaptado adaptado) {
6         this.adaptado = adaptado;
7     }
8
9     @Override
10    public void peticion() {
11        // Traduce la llamada a una llamada al método del Adaptado
12        adaptado.peticionEspecifica();
13    }
14 }
```

## Uso del patrón Adapter en la clase principal

Finalmente, en la clase principal, se instancia el Adaptado y el Adaptador, y se usa el Adaptador para llamar al método esperado por el cliente.

```
1
2 public class Cliente {
3     public static void main(String[] args) {
4         // Instanciar la clase Adaptado
5         Adaptado adaptado = new Adaptado();
6
7         // Instanciar la clase Adaptador pasándole la instancia de Adaptado
8         Objetivo adaptador = new Adaptador(adaptado);
9
10        // Usar el método del Adaptador que el cliente espera
11        adaptador.peticion();
12    }
13 }
14
```

## Funcionamiento del ejemplo

**Definición de Objetivo y Adaptado:** La interfaz Objetivo declara el método `peticion()` que el cliente espera usar. La clase Adaptado contiene la funcionalidad existente que necesita ser adaptada.

**Clase Adaptador:** La clase Adaptador implementa la interfaz Objetivo y traduce las llamadas al método `peticion()` en llamadas al método `peticionEspecifica()` de la clase Adaptado.

**Uso en Cliente:** Aquí se muestra cómo instanciar la clase Adaptado y la clase Adaptador, y cómo usar el Adaptador para llamar al método `peticion()` esperado por el cliente. Esto permite que el cliente use la funcionalidad del Adaptado sin cambiar su código.

Este ejemplo ilustra cómo el patrón Adapter permite que clases con interfaces incompatibles trabajen juntas. Es útil en aplicaciones donde se necesita integrar clases antiguas con nuevas sin modificar el código existente.

Códigos de los Ejemplos en las carpetas `src` de cada patrón en el proyecto