
ANALYZE
DEVELOPMENT
DESIGN
IMPLEMENT

Contenido

Patrones de diseño

A la hora de desarrollar software, hay **escenarios muy comunes que se repiten de forma similar**, tanto en la **arquitectura** como en el **diseño** del software. Los patrones son soluciones a estos problemas comunes, y conocerlos es importante para utilizar mecanismos sólidos que mejoren la calidad del software.

Los patrones de diseño **son unas técnicas para resolver problemas comunes en el desarrollo de software** y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su **efectividad resolviendo problemas similares en ocasiones anteriores**. Otra es que **debe ser reutilizable**, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Las **ventajas** que proporciona el utilizar patrones de diseño en el desarrollo del software son:

- ✓ **Proporcionar catálogos de elementos reusables** en el diseño de sistemas software.
- ✓ **Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos** y solucionados anteriormente.
- ✓ Formalizar un **vocabulario común** entre diseñadores.
- ✓ **Estandarizar** el modo en que se realiza el diseño.
- ✓ **Facilitar el aprendizaje de las nuevas generaciones de diseñadores** condensando conocimiento ya existente.

Según la escala o nivel de abstracción, se clasifican en:

- ✓ **Patrones de arquitectura:** Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software. Algunos de los patrones de arquitectura son:

  - *MVC (Modelo-Vista-Controlador)*
  - *MVVM (Modelo Vista VistaModelo)*
  - *MVP (Modelo Vista Presentación)*
  - *Capas*
  - *Cliente-Servidor*
  - *Maestro-Esclavo*
  - *Tubería*
  - …

- ✓ **Patrones de diseño:** Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software. Algunos de los patrones de diseño son:

  - *Observador*
  - *Singleton*
  - *Factoría*
  - *Abstracta*
  - *Fachada*
  - *Iterador*
  - *Adaptador*
  - *Comando*
  - …

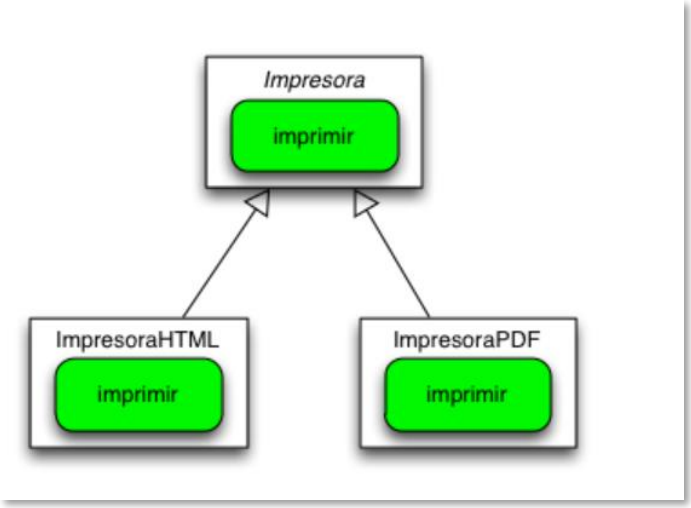
Para describir un patrón se **usan plantillas más o menos estandarizadas**, de forma que se expresen uniformemente y puedan constituir efectivamente un medio de comunicación uniforme entre diseñadores. Varios autores eminentes en esta área han propuesto plantillas ligeramente distintas, si bien la mayoría definen los mismos conceptos básicos (por ejemplo **GoF (Gang Of Four:** *"describen soluciones simples y elegantes a problemas específicos en el diseño de software orientado a objetos" (Gamma et al, 1994").*

Hay patrones de muchos tipos, pero los principales son los de arquitectura: DDD, microservicios, orientado a eventos, CQRS, por capas, hexagonal y el clásico MVC. También se utilizan en la actualidad algunos de los patrones de diseño más comunes: Adapter, Front Controller, Composite View, DAO, DTO...

PATRÓN ADAPTADOR: EJEMPLO PRÁCTICO

El patrón Adaptador **adapta una interfaz** para que pueda ser utilizada por una clase que de otro modo no podría utilizarla. Está dentro de los patrones estructurales (son los patrones de diseño software que solucionan problemas de composición (agregación) de clases y objetos).

Supongamos que tenemos la siguiente jerarquía de clases que definen varias impresoras que imprimen documentos en distintos formatos.



El código fuente de estas clases es el siguiente:

- ✓ Clase abstracta Impresora:

```
public abstract class Impresora {

    private String texto;

    public String getTexto() {
        return texto;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }

    public abstract void  imprimir();
```

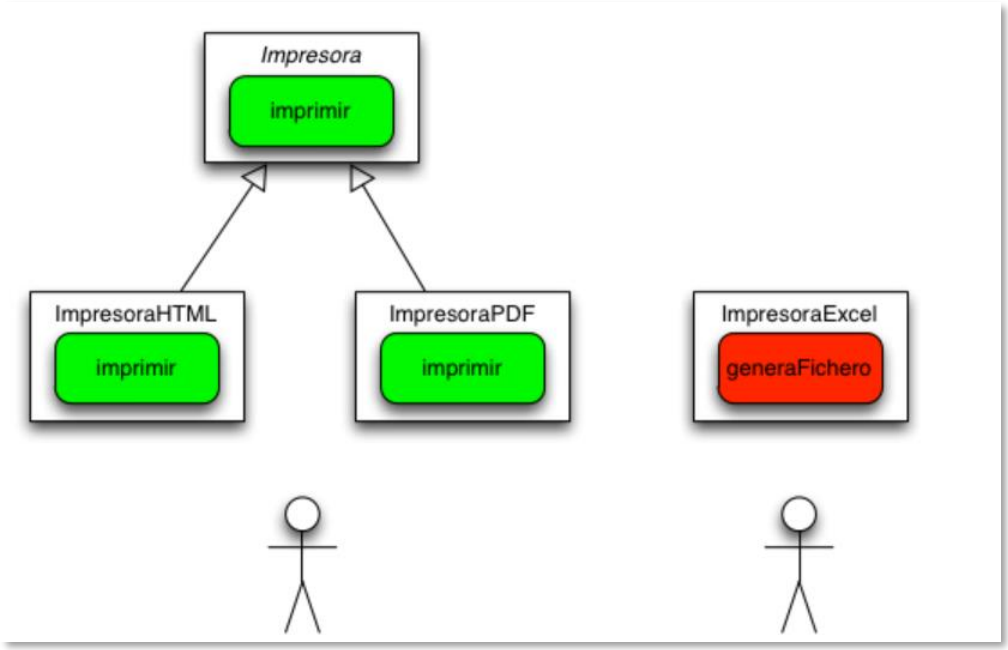
✓ Clase ImpresoraTexto que extiende de Impresora:

```
public class ImpresoraTexto extends Impresora {

    public ImpresoraTexto(String texto) {
        super();
        setTexto(texto);

    }

    @Override
    public void imprimir() {

        System.out.println("fichero texto con " + getTexto());

    }
}
```

✓ Clase ImpresoraPDF que extiende de Impresora:

```
public class ImpresoraPDF extends Impresora {

public ImpresoraPDF(String texto) {
 super();
 setTexto(texto);
 }
@Override
 public void imprimir() {

System.out.println("fichero pdf con " + getTexto());

}

}
```

Hemos creado una jerarquía de clases en la que el método imprimir se encarga de generar los distintos tipos de documentos. ¿Qué ocurriría si otro desarrollador ya hubiera construido una clase para ficheros excel y quisiéramos usarla dentro de nuestra jerarquía?

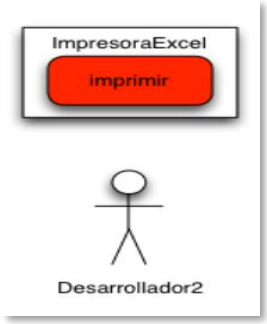


- ✓ Clase ImpresoraExcel

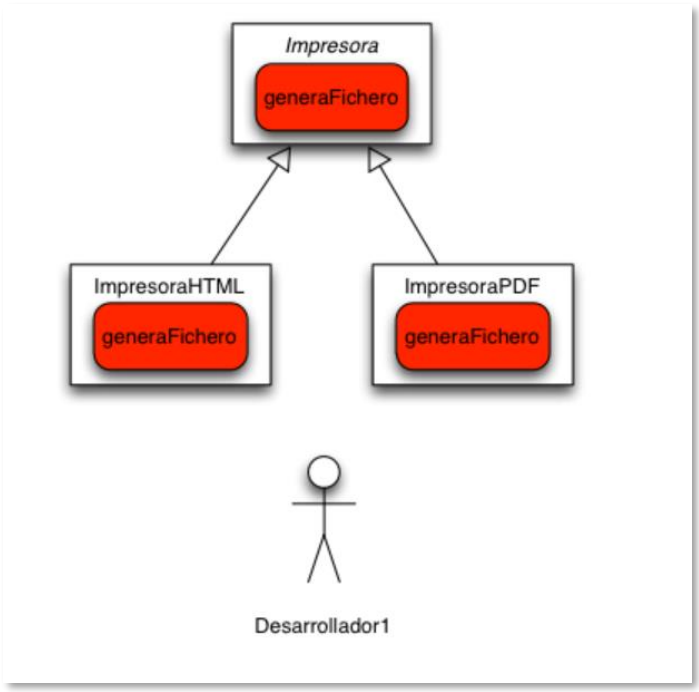
```
package adaptador;

public class ImpresoraExcel {

    public void generarFichero(String texto) {

        System.out.println("fichero excel con " + texto);

    }
}
```

Nos podemos dar cuenta de que la clase no encaja en nuestra jerarquía actual ya que no hereda de la clase Impresora y no comparte el método imprimir. Existen dos caminos erróneos básicos que se suelen tomar en estos casos:

1. Modificar la clase ImpresoraExcel para que encaje en nuestra jerarquía.

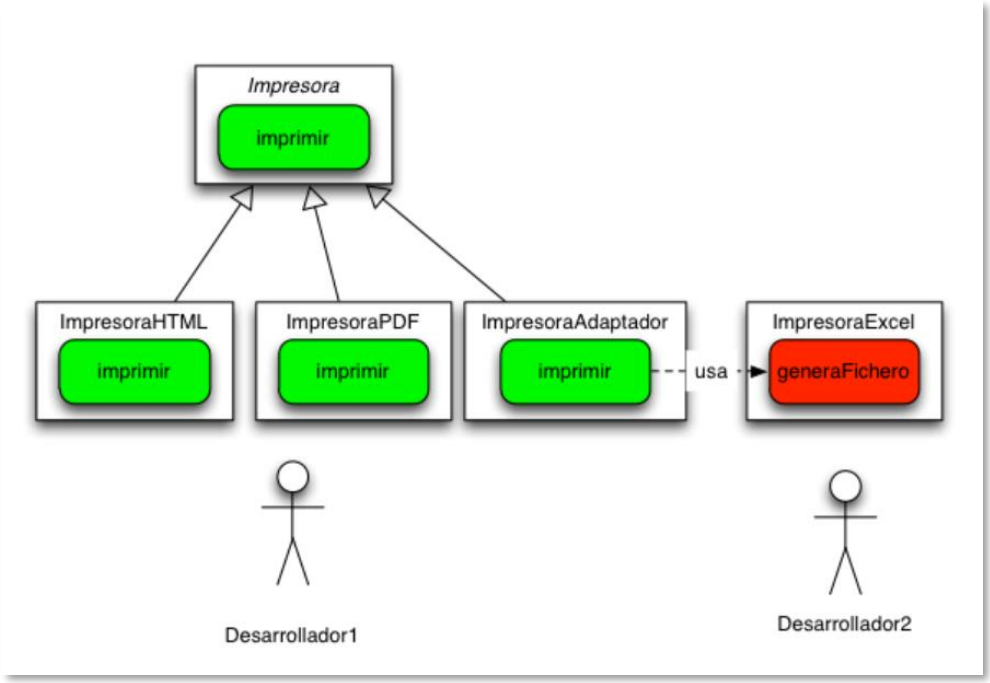


2. Modificar nuestra jerarquía de clases para que encaje con la clase ImpresoraExcel lo que supone un mayor esfuerzo.



Sin embargo, ambas soluciones rompen con **el principio OCP (Principio Abierto-Cerrado, una clase debe estar abierta a la extensibilidad pero cerrada a las modificaciones)**

Entonces, si queremos encajar de una manera correcta la clase ImpresoraExcel en nuestra jerarquía de clases deberemos hacer uso de una **clase Adaptador**. Esta clase hereda de la clase impresora y se encarga de adaptar los métodos de la clase ImpresoraExcel a la jerarquía que ya poseemos.



El código fuente de la clase Adaptadora quedaría de la siguiente manera:

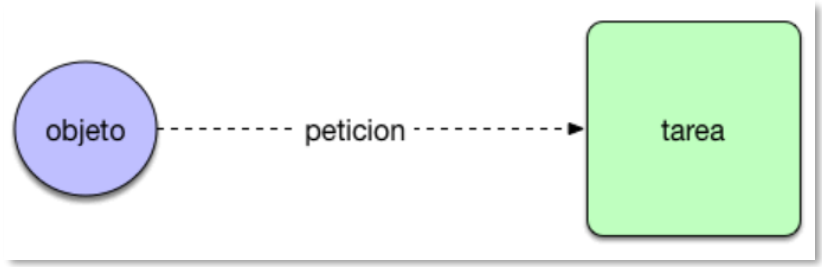
```
public class ImpresoraAdaptador extends Impresora {

    private final ImpresoraExcel impresoraExcel;

    public ImpresoraAdaptador() {
        super();
        impresoraExcel = new ImpresoraExcel();

    }

    @Override
    public void imprimir() {

        impresoraExcel.generarFichero(getTexto());

    }

}
```

De esta forma conseguimos cumplir con el principio OCP y no tener que modificar el código previamente construido. Viendo la relación existente entre los patrones de diseño y los principios de ingeniería del software.

PATRÓN COMANDO: EJEMPLO PRÁCTICO

El patrón comando encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma. Está dentro de los patrones de comportamiento (son patrones de diseño software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan).

En programación nos podemos encontrar en muchas situaciones en las que tenemos que gestionar tareas que reciben algún tipo de objeto como parámetro.



Una vez recibido este objeto deberemos procesarle. En principio es una tarea que parece muy sencilla y nos bastaría con tener una clase que implemente las diferentes tareas para el objeto.

Supongamos que disponemos de una clase Producto:

```java
package Comando;

public class Producto {

    private int id;
    private String nombre;
    private double precio;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

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

    public double getPrecio() {
        return precio;
    }

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

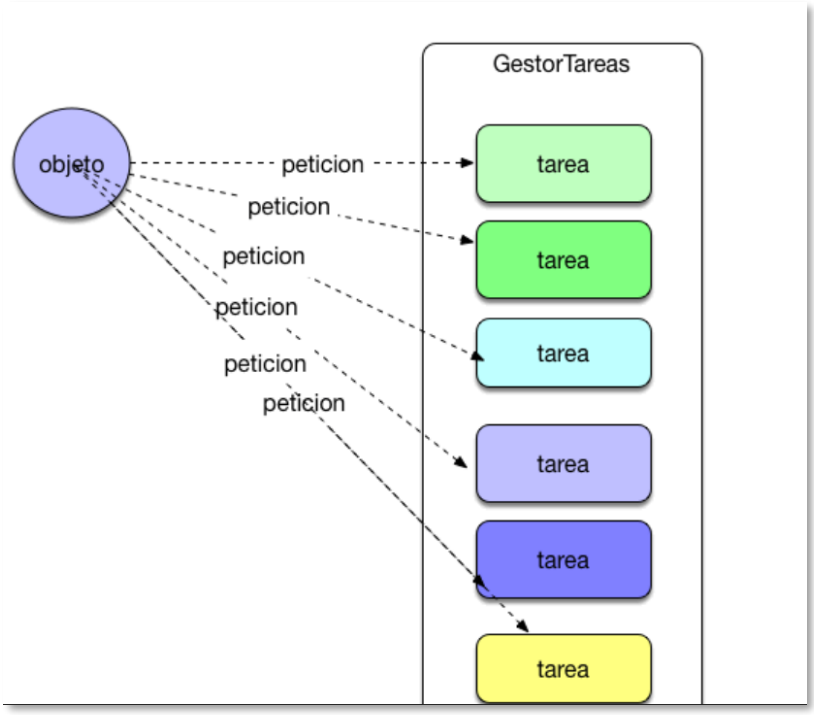
Deberemos realizar varias tareas:

- ✓ ValidarProducto
- ✓ EnviarPorCorreo
- ✓ Imprimir

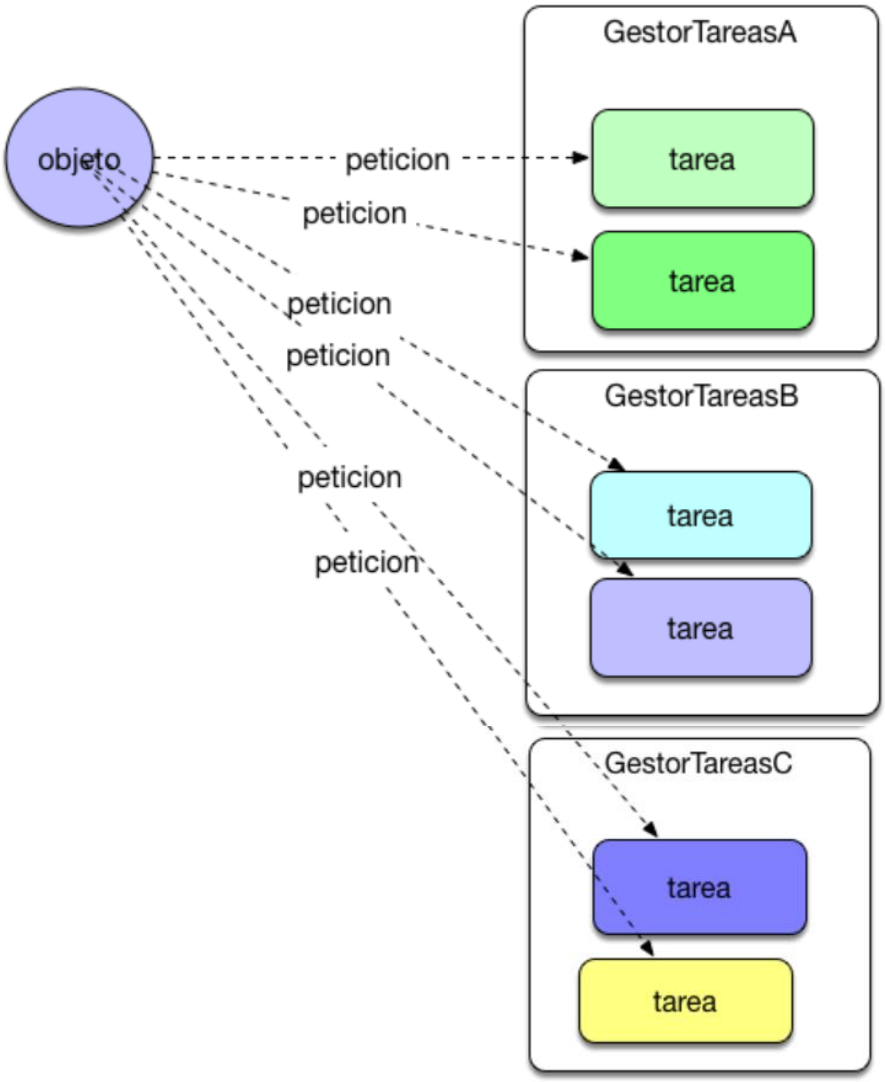
Estamos ante una situación muy sencilla, es suficiente crear una clase que disponga de tres métodos:

```java
package Comando;

public class GestorProductos {

    public void validarProducto(Producto producto) {

        if (producto.getPrecio() < 100) {

            System.out.println("producto valido");
        } else {

            System.out.println("producto invalido");
        }
    }

    public void imprimirProducto(Producto producto) {

        System.out.println(producto.getNombre());
        System.out.println(producto.getId());
        System.out.println(producto.getPrecio());
    }

    public void enviarPorCorreo(Producto producto) {

        System.out.println(producto.getNombre() + "enviado por correo");
    }
}
```

Los problemas surgen cuando las tareas crecen de forma exponencial.

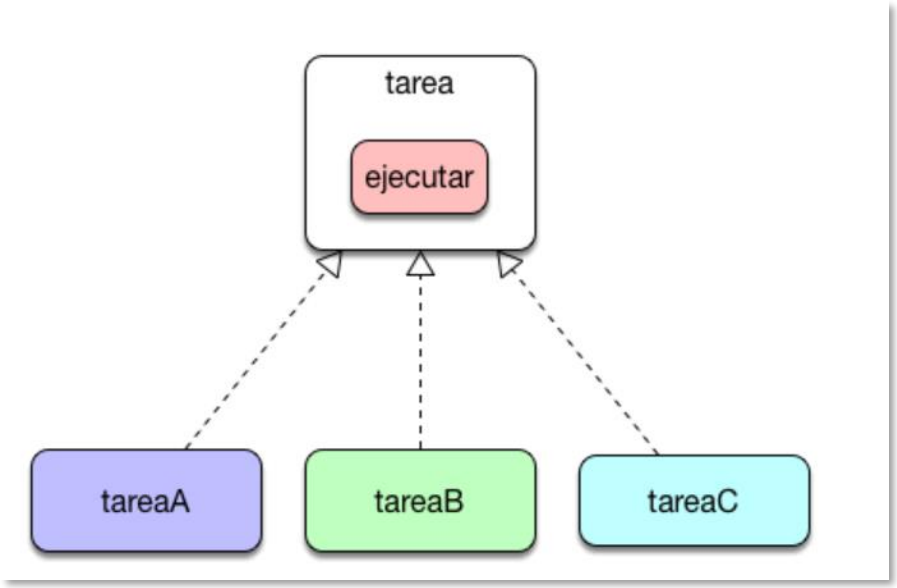


Una solución sencilla pasaría por construir más clases que almacenen los diferentes métodos.



Sin embargo, no siempre es la mejor solución ya que genera un fuerte acoplamiento entre el cliente y los diferentes componentes. Por otro lado, no siempre es sencillo decidir qué tareas van en cada clase ya que con el paso del tiempo las tareas y la relación entre ellas varía en un negocio.

Utilizando el patrón Comando, resolvemos el problema de una forma más correcta. Se define el concepto abstracto de tarea y de construir varias clases que lo implementen.



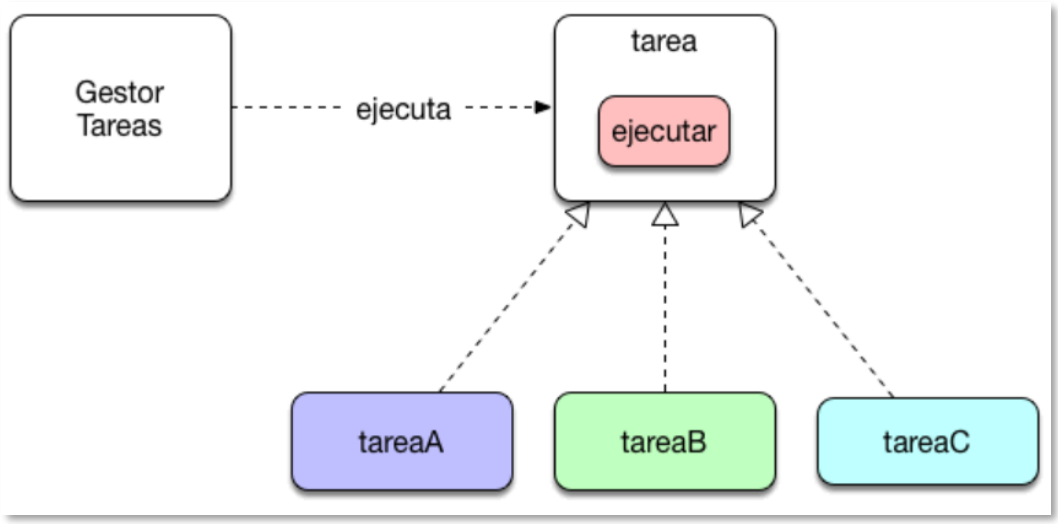

```java
package Comando;

public interface TareaProducto {

    public abstract void ejecutar(Producto producto);
}
```

```java
package Comando;

public class TareaEnvioCorreo implements TareaProducto {

    @Override
    public void ejecutar(Producto producto) {
        System.out.println(producto.getNombre() + "enviado por correo");

    }

}
```

```java
package Comando;

public class TareaImprimirProducto implements TareaProducto {

    @Override
    public void ejecutar(Producto producto) {
        System.out.println(producto.getNombre());
        System.out.println(producto.getId());
        System.out.println(producto.getPrecio());

    }
```

```java
package Comando;

public class ValidarProducto implements TareaProducto {

    @Override
    public void ejecutar(Producto producto) {
        if (producto.getPrecio() < 100) {

            System.out.println("producto valido");
        } else {

            System.out.println("producto invalido");
        }

    }

}
```

Una vez hemos construido la jerarquía de clases nos queda definir el GestorTareas que se encarga de ejecutar cada una de las tareas.




```java
package Comando;

public class GestorTareas {

    public void ejecutar(TareaProducto tarea, Producto p) {

        tarea.ejecutar(p);
    }
}
```

Este diseño nos permite tener una mayor flexibilidad ya que cada tarea es independiente. El añadir nuevas tareas no afecta al resto de tareas. Por otro lado es muy sencillo generar nuevas clases que por ejemplo agrupen tareas.

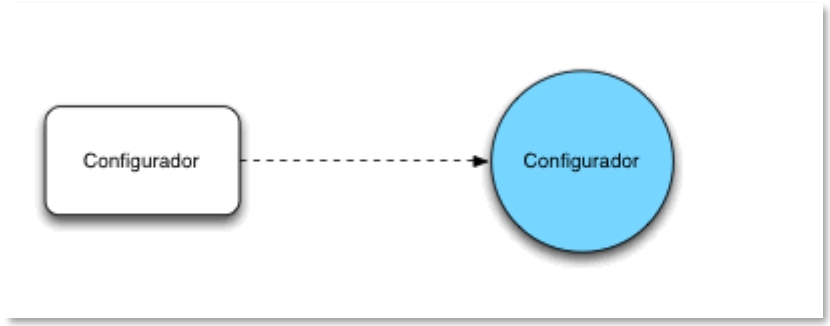
```java
package Comando;

import java.util.ArrayList;
import java.util.List;

public class SuperTarea implements TareaProducto {

    private final List<TareaProducto> lista;

    public SuperTarea() {
        this.lista = new ArrayList<>();
    }

    public void addTarea(TareaProducto tarea) {
        lista.add(tarea);
    }

    @Override
    @SuppressWarnings("empty-statement")
    public void ejecutar(Producto producto) {

        lista.forEach((t) -> t.ejecutar(producto));

    }

}
```

```java
package Comando;

public class Principal {

    public static void main(String[] args) {

        SuperTarea st = new SuperTarea();

        st.addTarea(new ValidarProducto());
        st.addTarea(new TareaEnvioCorreo());

        GestorTareas gt = new GestorTareas();
        Producto p = new Producto();
        p.setId(1);
        p.setNombre("Teclado");
        p.setPrecio(120);
        Producto p2 = new Producto();
        p2.setId(1);
        p2.setNombre("Ratón");
        p2.setPrecio(23.6);

        gt.ejecutar(st, p);
        gt.ejecutar(st,p2);

    }

}
```

PATRÓN SINGLETON: EJEMPLO PRÁCTICO

El patrón Singleton está incluido dentro de los patrones creacionales (este tipo de patrones que proporcionan herramientas crear objetos). Se encarga de que una clase determinada únicamente pueda tener un único objeto. Normalmente una clase puede instanciar todos los objetos que necesite. Este tipo de clases son habituales en temas como configurar parámetros generales de la aplicación ya que, una vez instanciado el objeto, los valores se mantienen y son compartidos por toda la aplicación.

Vamos a configurar una clase con el patrón Singleton, a esta clase la llamaremos Configurador.

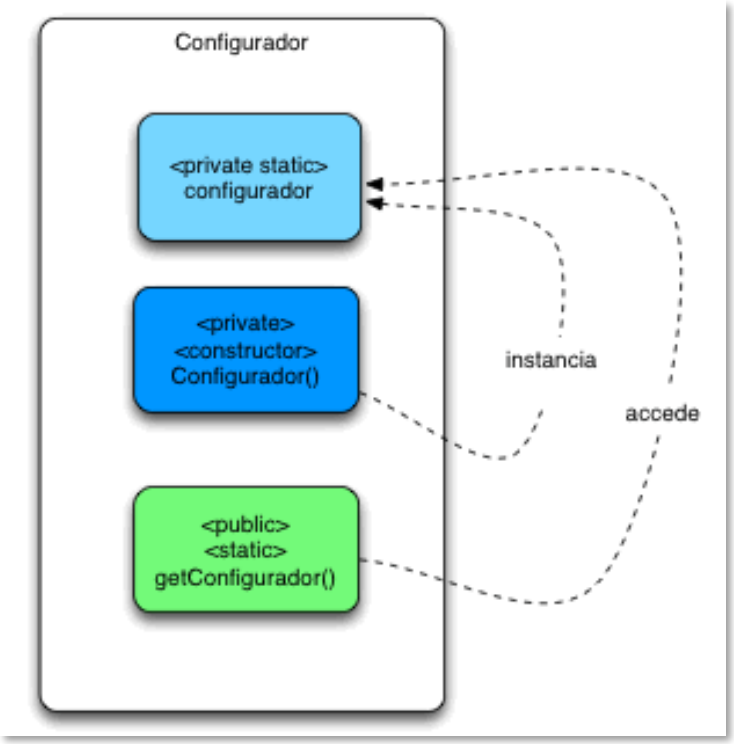


En este caso el configurador almacenará dos valores URL y base de datos que serán compartidos por el resto de las clases de la aplicación.

```java
package Singleton;

public class Configurador {

    private String url;
    private String baseDatos;
    private static Configurador miconfigurador;

    private Configurador(String url, String baseDatos) {

        this.url = url;
        this.baseDatos = baseDatos;

    }

    public static Configurador getConfigurador(String url, String

        if (miconfigurador == null) {

            miconfigurador = new Configurador(url, baseDatos);
        }
        return miconfigurador;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getBaseDatos() {
        return baseDatos;
    }

    public void setBaseDatos(String baseDatos) {
        this.baseDatos = baseDatos;
    }
}
```

Para conseguir que una clase sea de tipo Singleton necesitamos en primer lugar que su constructor sea privado. De esa forma ningún programa será capaz de construir

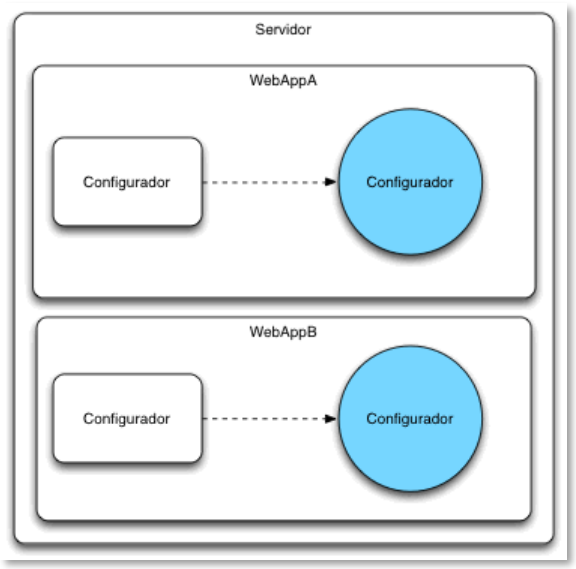
objetos de este tipo. En segundo lugar, necesitaremos disponer de una variable estática privada que almacene una referencia al objeto que vamos a crear a través del constructor. Por último, un método estático público que se encarga de instanciar el objeto la primera vez y almacenarlo en la variable estática.



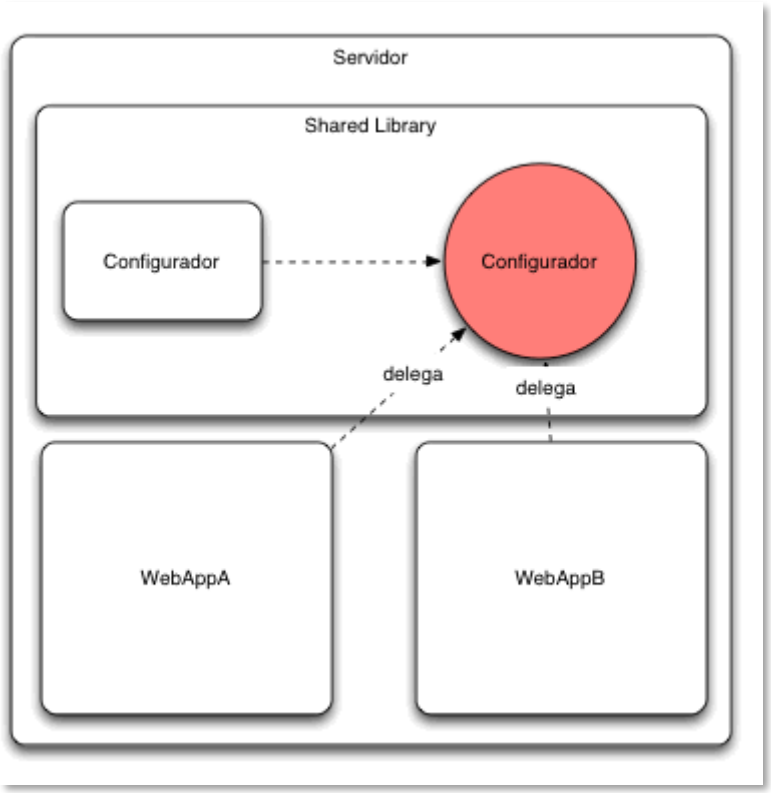
Una vez aclarado como funciona un Singleton es muy sencillo utilizarlo desde un programa ya que basta con invocar al método estático.

```java
package Singleton;

public class Principal2 {

public static void main(String[] args) {

 Configurador c= Configurador.getConfigurador("miurl", "mibaseDatos");

 System.out.println(c.getUrl());

 System.out.println(c.getBaseDatos());

}

}
```

El **patron Singleton** nos genera un objeto por cada clase cargada en el mismo ClassLoader, esto quiere decir que por ejemplo dos aplicaciones web que cada una tiene su propio WebClassLoader tendrán cada una su propia instancia.



Esto en principio no es problemático porque se encuentran aisladas. Ahora bien, hay situaciones en las que un administrador de sistemas puede decidir compartir librerías y clases entre distintas aplicaciones.



En este caso si podemos tener problemas ya que el objeto Singleton que en principio fue diseñado para configurar una aplicación concreta estará compartido por varias.

Patrones de arquitectura: MVVM, MVP y MVC

MVVM

Significa **Modelo Vista VistaModelo**, porque en este patrón de diseño **se separan los datos de la aplicación y la interfaz de usuario**, pero en vez de controlar manualmente los cambios en la vista o en los datos, estos se actualizan directamente cuando sucede un cambio en ellos, por ejemplo, **si la vista actualiza un dato que está presentando se actualiza el modelo automáticamente y viceversa.**

MVP

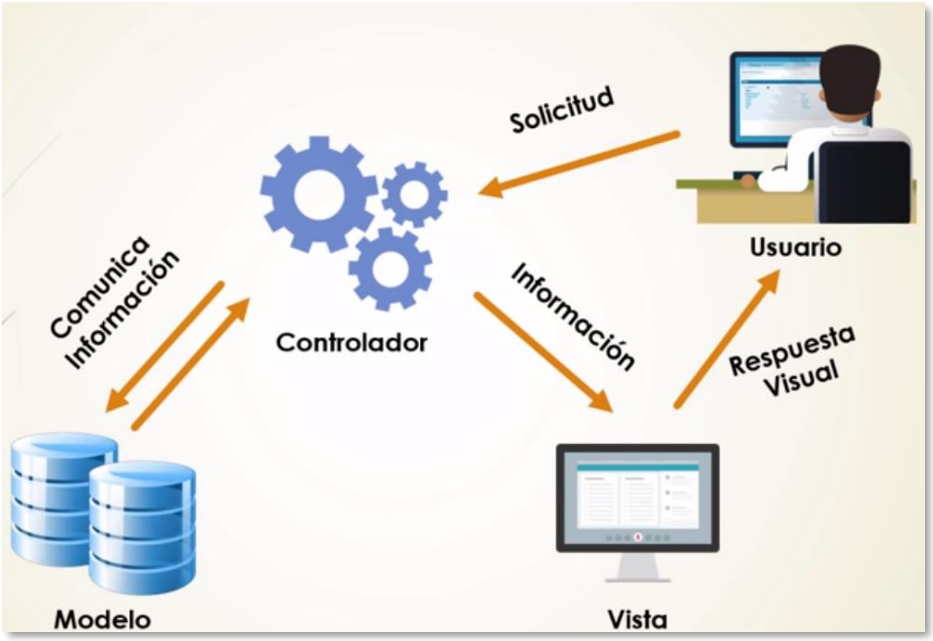
Significa **Modelo-Vista-Presentador**, y surge para ayudar a realizar pruebas automáticas de la interfaz gráfica, para ello la idea es **codificar la interfaz de usuario lo más simple posible, teniendo el menor código posible, sin necesidad de probarla.** En su lugar, toda l**a lógica de la interfaz de usuario se hace en una clase separada, llamada Presentador**, que no dependa en absoluto de los componentes de la interfaz gráfica y que, por tanto, es más fácil de realizar pruebas. La idea básica es que la clase Presentador haga **de intermediario entre la Vista** (la interfaz gráfica de usuario) **y el modelo de datos.** La vista tiene métodos en los que le pasan los datos preparados (una lista de cadenas, por ejemplo, métodos get directamente, en vez del modelo). Únicamente debe meter esos datos en los componentes gráficos (cajas de texto, checkbox, etc).

MVC

Significa **Modelo Vista Controlador**, trata de dividir la aplicación en tres tipos de elementos, el **modelo, las vistas (GUIs) y controladores**. La manera en que los elementos dentro de este patrón se comunican difieren y no sólo lo diferencia el tipo de aplicación que se está describiendo (Desktop, WEB), sino también por la parte de la aplicación que actualmente se esté trabajando (Frontend o Backend).

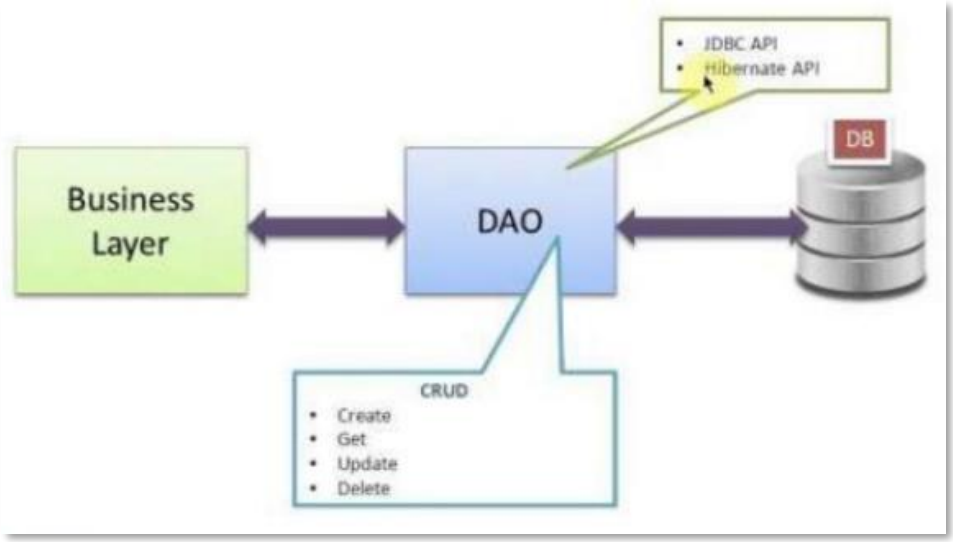
MVC permite dividir en partes, que de alguna manera son independientes, con lo que **si se hace algún cambio en el modelo no afecte a la vista o si hay algún**

cambio que sea mínimo. Este patrón permite separar una aplicación en tres capas, una forma de organizar y de hacer escalable un proyecto.

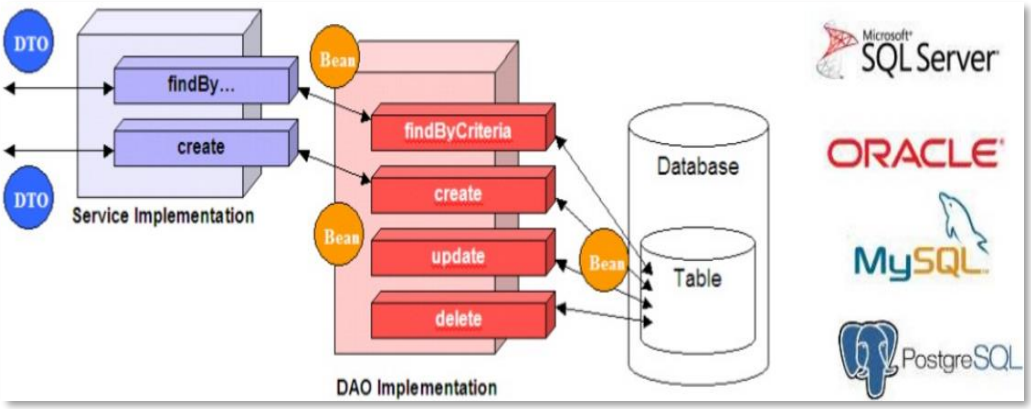


MVC define la separación de tres tipos de elementos:

- ✓ **Capa de datos (el modelo):** elementos que contienen los datos (archivo de texto, de una base de datos, etc) y definen la lógica para manipular dichos datos. A menudo **los objetos tienen una naturaleza reutilizable, distribuida, persistente y portátil** para una variedad de plataformas. En este modelo, el patrón más usado son los **DAO (Data Access Object) u Objeto de Acceso a Datos.**

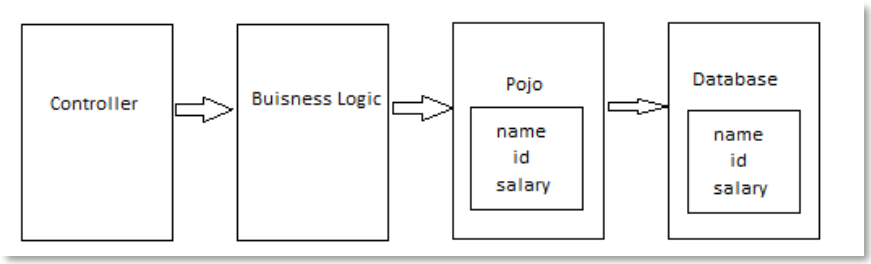


- DAO encapsula el acceso a la base de datos. Cuando el controlador necesita interactuar con la base de datos lo hace a través de la API que ofrece DAO. Generalmente esta API consiste en métodos CRUD (Create, Read, Update y Delete).



Algunos de los lenguajes más populares que soportan DAO son:

- Otro pequeño patrón que se utiliza dentro del modelo es el **VO (value Object).** Consiste en agrupar varios atributos dentro de un objeto para enviarlo y recibirlo con mayor seguridad y comodidad. También se trata el concepto de **DTO (Objeto de transferencia de datos), POJO (Plain Old Java Objec).**



✓ **Capa de la interfaz de usuario (la vista):** La parte donde **se interactúa con el usuario**, se especifica como se posicionarán los datos y como se desplegarán. Las interfaces de usuario que nos podemos encontrar son:

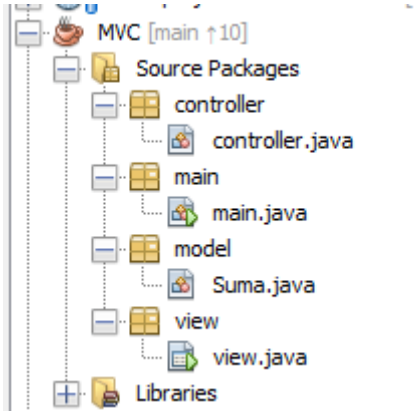
- **Interfaz de escritorio:** muestra la información de forma simultánea, realizando tareas de control y de dialogo de forma sencilla, la utilización de menús, botones y técnicas de presentación, reducen el manejo del teclado.
- **Interfaz web:** la interfaz utilizada desde un navegador web para comunicarse con un servidor http para visualizar el contenido o la página web. Une los atributos de la interfaz de escritorio, hipertexto y la multitarea.
- **Interfaz móvil:** se utiliza en dispositivos móviles, utiliza pantalla táctil, multimedia (reproductor vídeo, música, cámara fotográfica, etc)

✓ **Capa de controlador**: es el **intermediario entre la vista y el modelo.** Gestiona el flujo de información entre ellos y sus transformaciones, recibe los eventos de entrada (un click, un campo de texto, etc) y delega la búsqueda de datos al modelo y selecciona el tipo de respuesta más adecuada. Un controlador también realiza todas las tareas específicas de la aplicación, tales como la entrada del usuario o la carga de procesamiento de datos de configuración. Por lo general se puede ver **un controlador por aplicación o ventana (¿?),** en muchas aplicaciones, **el controlador está estrechamente acoplado a la vista.** Dado

que los controladores **son específicos de cada aplicación**, por lo general, es muy difícil encontrar reutilización en otras aplicaciones.

- ✓ El proceso de funcionamiento en una aplicación web por ejemplo es el siguiente:
  - El usuario interactúa con la interfaz (Vista).
  - El Controlador recibe la información que le envía la Vista.
  - El Controlador actúa sobre el modelo.
  - El Modelo devuelve la confirmación al Controlador y este actualiza la Vista.
  - La Vista espera una nueva acción del usuario.

La estructura de un proyecto en cualquier IDE de desarrollo siguiendo el diseño MVC podría ser el siguiente:



El diseño de una aplicación estrictamente de acuerdo con MVC no es siempre recomendable. Si está diseñando un programa intensivo de gráficos, probablemente tendremos muy pocas vistas y clases de modelo mucho más de lo que sugiere MVC y al programar una aplicación muy simple es común combinar el controlador con las clases de vista.

El campo de aplicación de este patrón es bastante amplio, aunque dependiendo de la aplicación, puede que algunas clases deben acoplarse a otras más que en otras aplicaciones.

El patrón MVC en su implementación embebe de diferentes patrones dependiendo de la naturaleza de la aplicación que se está diseñando.