PATRONES DE DISEÑO EN JAVA MVC, DAO, DTO

Los patrones de diseño en Java, concretamente los patrones **Modelo Vista Controlador (MVC), Data Acces Object (DAO) y Data Transfer Object (DTO)** y su implementación en Java.

QUÉ ES UN PATRÓN DE DISEÑO?

Un patrón de diseño es una solución probada que resuelve un tipo específico de problema en el desarrollo de software referente al diseño.

Existen una infinidad de patrones de diseño los mismos que se dividen en categorías por ejemplo: de creación, estructurales, de comportamiento, interacción etc.

Cada uno se especializa en resolver un problema específico, si quieres profundizar y revisar todas las categorías y ejemplos a detalle puedes visitar Design Patterns Book.

El tema es bastante extenso, que no alcanzaría una sola entrada, pero esta vez quiero hablar de los patrones MVC, DAO, DTO que en lo personal y en la práctica considero los más utilizados al menos para el desarrollo en Java.

POR QUÉ UTILIZAR UN PATRÓN DE DISEÑO?

Ahora, cuáles son las ventajas?, bueno son algunas, entre las principales es que permiten tener el código bien **organizado, legible y mantenible**, además te permite **reutilizar código** y aumenta la **escalabilidad** en tu proyecto.

En sí proporcionan una terminología estándar y un conjunto de buenas prácticas en cuanto a la solución en problemas de desarrollo de software.

Sin más palabras voy a empezar a describirlos con sus respectivos ejemplos.

EL PATRÓN MODEL VIEW CONTROLLER O MVC

En español Modelo Vista Controlador, este patrón permite separar una aplicación en 3 capas, una forma de organizar y de hacer escalable un proyecto, a continuación una breve descripción de cada capa.

Modelo: Esta capa representa todo lo que tiene que ver con el acceso a datos: guardar, actualizar, obtener datos, además todo el código de la lógica del negocio, básicamente son las clases Java y parte de la lógica de negocio y POJO's.

Vista: La vista tiene que ver con la presentación de datos del modelo y lo que ve el usuario, por lo general una vista es la representación visual de un modelo (POJO o clase java).

Por ejemplo el modelo usuario que es una clase en Java y que tiene como propiedades, nombre y apellido debe pertenecer a una vista en la que el usuario vea esas propiedades.

Controlador: El controlador es el encargado de conectar el modelo con las vistas, funciona como un puente entre la vista y el modelo, el controlador recibe eventos generados por el usuario desde las vistas y se encarga de direccionar al modelo la petición respectiva.

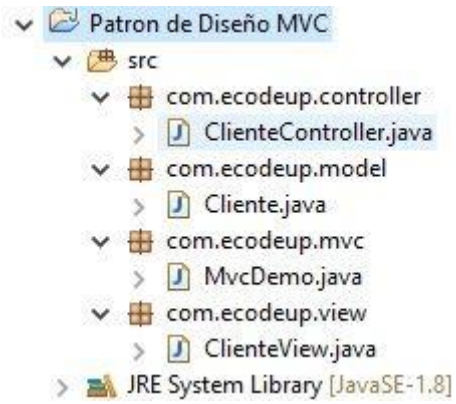
Por ejemplo el usuario quiere ver los clientes con apellido Álvarez, la petición va al controlador y él se encarga de utilizar el modelo adecuado y devolver ese modelo a la vista.

Si te das cuenta en ningún momento interactúan directamente la vista con el modelo, esto también mantiene la seguridad en una aplicación.

QUÉ GANO UTILIZANDO ESTE PATRÓN

Lo importante de este patrón es que permite dividir en partes, que de alguna manera son independientes, con lo que si por ejemplo hago algún cambio el modelo no afectaría a la vista o si hay algún cambio sería mínimo.

CÓMO IMPLEMENTO EL MODELO VISTA CONTROLADOR



Para usar este patrón de diseño en Java, primero creas el modelo, que es una clase en java y se llama ***Cliente.java***, esta clase sólo contiene los atributos, constructor, getters y setters.

```
package com.ecodeup.model;

public class Cliente {
	private int id;
	private String nombre;
	private String apellido;


	public Cliente() {
	}

	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 String getApellido() {
		return apellido;
	}
	public void setApellido(String apellido) {
		this.apellido = apellido;
	}
}
```

Luego creas la vista, la clase ***ClienteView.java***, que es una clase que va hacer de vista para el ejemplo y su función es presentar los datos del modelo.

```
package com.ecodeup.view;

public class ClienteView {
	public void imprimirDatosCliente(int id,String nombre,
String apellido) {
		System.out.println("**** DATOS CLIENTE ****");
		System.out.println("Id: "+id);
		System.out.println("Nombre: "+nombre);
		System.out.println("Apellido: "+apellido);
	}
```

```
}
```

Esta clase lo único que va hacer es imprimir los datos del modelo que es la clase Cliente.java.

Ahora creas el controlador, el controlador contiene 2 objetos el modelo, la vista así como los getters y setters para llenar las propiedades del modelo y un método(actualizarVista()) que llama a la vista que a su vez imprime las propiedades del modelo cliente.

```
1 package com.ecodeup.controller;
2
3 import com.ecodeup.model.Cliente;
4 import com.ecodeup.view.ClienteView;
5
6 public class ClienteController {
7       //objetos vista y modelo
8       private ClienteView vista;
9       private Cliente modelo;
1
0       //constructor para inicializar el modelo y la vista
1       public ClienteController(Cliente modelo, ClienteView
1 vista) {
1             this.modelo = modelo;
2             this.vista = vista;
1       }
3
1       //getters y setters para el modelo
4       public int getId() {
1             return modelo.getId();
5       }
1       public void setId(int id) {
6             this.modelo.setId(id);
1       }
7       public String getNombre() {
1             return modelo.getNombre();
8       }
1       public void setNombre(String nombre) {
9             this.modelo.setNombre(nombre);
2       }
0       public String getApellido() {
2             return modelo.getApellido();
1       }
2       public void setApellido(String apellido) {
2             this.modelo.setApellido(apellido);
2       }
3
2       //pasa el modelo a la vista para presentar los datos
4       public void actualizarVista() {
2
5       vista.imprimirDatosCliente(modelo.getId(),modelo.getNomb
2 re(), modelo.getApellido());
6       }
```

```
2 }
7
```

Finalmente queda hacer un test para comprobar el patrón de diseño **Modelo Vista Controlador** funciona:

```
  package com.ecodeup.mvc;

1 import com.ecodeup.controller.ClienteController;
2 import com.ecodeup.model.Cliente;
3 import com.ecodeup.view.ClienteView;
4
5 public class MvcDemo {
6
7       public static void main (String [] args){
8             // objeto vista, y modelo creado con el método
9 estático
10            Cliente modelo= llenarDatosCliente();
11            ClienteView vista= new ClienteView();
12
13            //se crea un objeto controlador y se le pasa el
14 modelo y la vista
15            ClienteController controlador= new
16 ClienteController(modelo, vista);
17
18            // se muestra los datos del cliente
19            controlador.actualizarVista();
20
21            // se actualiza un cliente y se muestra de nuevo
22 los datos
23            controlador.setNombre("Luis");
24            controlador.actualizarVista();
25      }
26      //método estático que retorna el cliente con sus datos
27      private static Cliente llenarDatosCliente() {
28            Cliente cliente = new Cliente();
29            cliente.setId(1);
30            cliente.setNombre("Elivar");
31            cliente.setApellido("Largo");
32            return cliente;
        }
  }
```

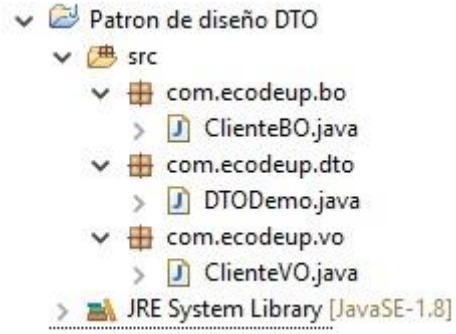
EL PATRÓN DATA TRANSFER OBJECT (DTO/VO)

Va de la mano con el patrón de diseño DAO.

Se utiliza para transferir varios atributos entre el cliente y el servidor o viceversa, básicamente consta de 2 clases:

- La primera es una clase java conocida como ***Value Object*** que únicamente contiene sus atributos, constructor, getters y setters, esta clase no tiene comportamiento.
- La segunda es una clase del lado del servidor conocida como clase de negocio (***en la implementación también se conoce como Business Object***) es la que se encarga de obtener datos desde la base de datos y llenar la clase ***Value Object*** y enviarla al cliente, o a su vez recibir la clase desde el cliente y enviar los datos al servidor, por lo general tiene todos los métodos CRUD (create, read, update y delete).

Se implementa de la siguiente forma:



Se crea la clase ***ClienteVO.java*** que será la clase también conocida como *Value Object:*

```
 1 package com.ecodeup.vo;
 2
 3 public class ClienteVO {
 4       private int id;
 5       private String nombre;
 6       private String apellido;
 7
 8
 9       public ClienteVO(int id, String nombre, String apellido)
10 {
11             this.id = id;
12             this.nombre = nombre;
13             this.apellido = apellido;
14       }
15       public int getId() {
16             return id;
17       }
18       public void setId(int id) {
19             this.id = id;
20       }
21
22       public String getNombre() {
23             return nombre;
```

```
24      }
25      public void setNombre(String nombre) {
26           this.nombre = nombre;
27      }
28
29      public String getApellido() {
30           return apellido;
31      }
32      public void setApellido(String apellido) {
33           this.apellido = apellido;
34      }
35
36      @Override
37      public String toString() {
38           return this.getNombre()+" "+this.getApellido();
39      }
  }
```

Se crea la clase ***ClienteBO.java*** conocida también como la clase de negocio, que es la que contiene todos los métodos CRUD:

```
1 package com.ecodeup.bo;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import com.ecodeup.vo.ClienteVO;
7
8 public class ClienteBO {
9
1       //lista de tipo cliente
0       List<ClienteVO> clientes;
1
1
1       //constructor, se guarda en la lista 2 clientes
2       public ClienteBO() {
1            clientes = new ArrayList<>();
3            ClienteVO cliente1= new
1 ClienteVO(,"Elivar","Largo");
4            ClienteVO cliente2= new
1 ClienteVO(1,"Priscila","Morocho");
5            clientes.add(cliente1);
1            clientes.add(cliente2);
6       }
1
7       //elimina el cliente que se le pasa como paraámetro
1       public void eliminarCliente(ClienteVO cliente) {
8            clientes.remove(cliente.getId());
1            System.out.println("Cliente "+cliente.getId()+"
9 eliminado satisfactoriamente");
2       }
0
```

```
2       //obtiene toda la lista de clientes
1       public List<ClienteVO> obtenerClientes(){
2            return clientes;
2       }
2
3       //obtiene un cliente de acuerdo al id pasado como
2 parámetro
4       public ClienteVO obtenerCliente(int id) {
2            return clientes.get(id);
5       }
2
6       // actualiza el cliente que se le pasa como parámetro
2       public void actualizarCliente(ClienteVO cliente) {
7
2       clientes.get(cliente.getId()).setNombre(cliente.getNombr
8 e());
2         clientes.get(cliente.getId()).setApellido(cliente.getAp
9 ellido());
3            System.out.println("Cliente id: "+
0 cliente.getId()+" actualizado satisfactoriamente");
3       }
1 }
```

Finalmente probamos el patrón **Data Transfer Object:**

```
1 package com.ecodeup.dto;
2
3 import com.ecodeup.bo.ClienteBO;
4 import com.ecodeup.vo.ClienteVO;
5
6 public class DTODemo {
7       public static void main(String[] args) {
8            //objeto business object
9            ClienteBO clienteBusinessObject = new ClienteBO();
1
0            //obtiene todos los clientes
1
1       clienteBusinessObject.obtenerClientes().forEach(System.o
1 ut::println);
2
1            // actualiza un cliente
3            System.out.println("****");
1            ClienteVO cliente =
4 clienteBusinessObject.obtenerCliente();
1            cliente.setNombre("Luis");
5            clienteBusinessObject.actualizarCliente(cliente);
1
6            // obtiene un cliente
1            System.out.println("****");
7            cliente=clienteBusinessObject.obtenerCliente();
1            System.out.println(cliente);
8
1            //elimina un cliente
```

```
9             System.out.println("****");
2             cliente=clienteBusinessObject.obtenerCliente();
0             clienteBusinessObject.eliminarCliente(cliente);
2
1       }
2 }
2
```

EL PATRÓN DATA ACCES OBJECT (DAO)

El problema que viene a resolver este patrón es netamente el acceso a los datos, que básicamente tiene que ver con la gestión de diversas fuentes de datos y además abstrae la forma de acceder a ellos.

Imagínate que tienes un sistema montado en producción con una base de datos MySQL y de pronto lo debes cambiar a PostgreSQL o a cualquier otro motor de base de datos.

Eso puede ser un verdadero problema.

Y precisamente esto lo que soluciona este patrón, tener una aplicación que no esté ligada al acceso a datos, que si por ejemplo la parte de la vista pide encontrar los clientes con compras mensuales mayores $200, el ***DAO*** se encargue de traer esos datos independientemente si está en un archivo o en una base de datos.

La capa DAO contiene todos los métodos CRUD (Create, Read, Update, Delete), por lo general se tiene un DAO para cada tabla en la base de datos, y bueno la implementación se la realiza de la siguiente manera.

Se crea una clase ***Cliente.java*** únicamente con sus constructores, getters y setters.

```
1 package com.ecodeup.model;
2
3 public class Cliente {
4       private int id;
```

```
5       private String nombre;
6       private String apellido;
7
8
9       public Cliente() {
10            super();
11      }
12      public Cliente(int id, String nombre, String apellido) {
13            super();
14            this.id = id;
15            this.nombre = nombre;
16            this.apellido = apellido;
17      }
18      public int getId() {
19            return id;
20      }
21      public void setId(int id) {
22            this.id = id;
23      }
24
25      public String getNombre() {
26            return nombre;
27      }
28      public void setNombre(String nombre) {
29            this.nombre = nombre;
30      }
31
32      public String getApellido() {
33            return apellido;
34      }
35      public void setApellido(String apellido) {
36            this.apellido = apellido;
37      }
38
39      @Override
40      public String toString() {
41            return this.getNombre()+" "+this.getApellido();
42      }
43 }
```

Se crea el acceso a los datos a través de una interface ***IClienteDao.java***, aquí se declara todos los métodos para acceder a los datos.

```
1 package com.ecodeup.idao;
2
3 import java.util.List;
4
5 import com.ecodeup.model.Cliente;
6
7 public interface IClienteDao {
8       //declaración de métodos para acceder a la base de datos
9       public List<Cliente> obtenerClientes();
```

```
10      public Cliente obtenerCliente(int id);
11      public void actualizarCliente(Cliente cliente);
12      public void eliminarCliente(Cliente cliente);
13 }
```

Se implementa en la clase ***ClienteDaoImpl.java*** haciendo un *implements* de la interfaz ***IClienteDao.java,*** lo que se hace aquí, no es más que implementar cada método de la interface.

```
package com.ecodeup.dao;

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

import com.ecodeup.idao.*;
import com.ecodeup.model.Cliente;

public class ClienteDaoImpl implements IClienteDao {

      //lista de tipo cliente
      List<Cliente> clientes;

      //inicializar los objetos cliente y añadirlos a la lista
      public ClienteDaoImpl() {
            clientes = new ArrayList<>();
            Cliente cliente1 = new Cliente(,"Javier", "Molina");
            Cliente cliente2 = new
Cliente(1,"Lillian","Álvarez");
            clientes.add(cliente1);
            clientes.add(cliente2);
      }

      //obtener todos los clientes
      @Override
      public List<Cliente> obtenerClientes() {
            return clientes;
      }

      //obtener un cliente por el id
      @Override
      public Cliente obtenerCliente(int id) {
            return clientes.get(id);
      }

      //actualizar un cliente
      @Override
      public void actualizarCliente(Cliente cliente) {

      clientes.get(cliente.getId()).setNombre(cliente.getNombre()
);
```

```
      clientes.get(cliente.getId()).setApellido(cliente.getApelli
do());
            System.out.println("Cliente con id:
"+cliente.getId()+" actualizado satisfactoriamente");
      }

      //eliminar un cliente por el id
      @Override
      public void eliminarCliente(Cliente cliente) {
            clientes.remove(cliente.getId());
            System.out.println("Cliente con id:
"+cliente.getId()+" elimnado satisfactoriamente");
      }
}
```

Por último se prueba el patrón DAO a través de la clase DaoDemo.java

```
1
2 package com.ecodeup.daodemo;
3
4 import com.ecodeup.dao.ClienteDaoImpl;
5 import com.ecodeup.idao.IClienteDao;
6 import com.ecodeup.model.Cliente;
7
8 public class DaoDemo {
9
1       public static void main(String[] args) {
0             // objeto para manipular el dao
1             IClienteDao clienteDao = new ClienteDaoImpl();
1
1             // imprimir los clientes
2
1       clienteDao.obtenerClientes().forEach(System.out::println
3 );
1
4             // obtner un cliente
1             Cliente cliente = clienteDao.obtenerCliente();
5             cliente.setApellido("Pardo");
1             //actualizar cliente
6             clienteDao.actualizarCliente(cliente);
1
7             // imprimir los clientes
1             System.out.println("*****");
8
1       clienteDao.obtenerClientes().forEach(System.out::println
9 );
2       }
0 }
```

Por lo general en un proyecto se suele unir estos tres patrones, ya que no sólo basta utilizar el MVC, puesto que la parte de acceso a datos queda un poco suelta, entonces si tú pregunta es: Si se puede utilizar los tres

patrones en un proyecto? la respuesta es SI, y de hecho se puede mezclar con otra infinidad de patrones que existen.

Todo dependerá del tipo de problema que quieras solucionar.

Y por último, tampoco se debe abusar con el uso de patrones, ya que utilizarlos de forma desmedida y de brindar ayuda pueden llegar a ser perjudicial para nuestro código.