

SISTEMAS UNI

Título: Patrón de arquitectura de software MVC

Autor: De la cruz rivera Jhordan

Profesor: Eric Gustavo Coronel Castillo

Lima, 27 enero del 2020

*Al profesor Gustavo Coronel,*

*que nos mostró una*

*nueva forma de programar en java*

*(Con enfoque de servicio)*

**Índice**

1. El patrón Modelo-Vista-Controlador (MVC) 4

1.1. Modelo 4

1.2. Vista 4

1.3. Control 4

2. Historia 5

3. Ejemplo de aplicación en Java 6

3.1. El modelo 6

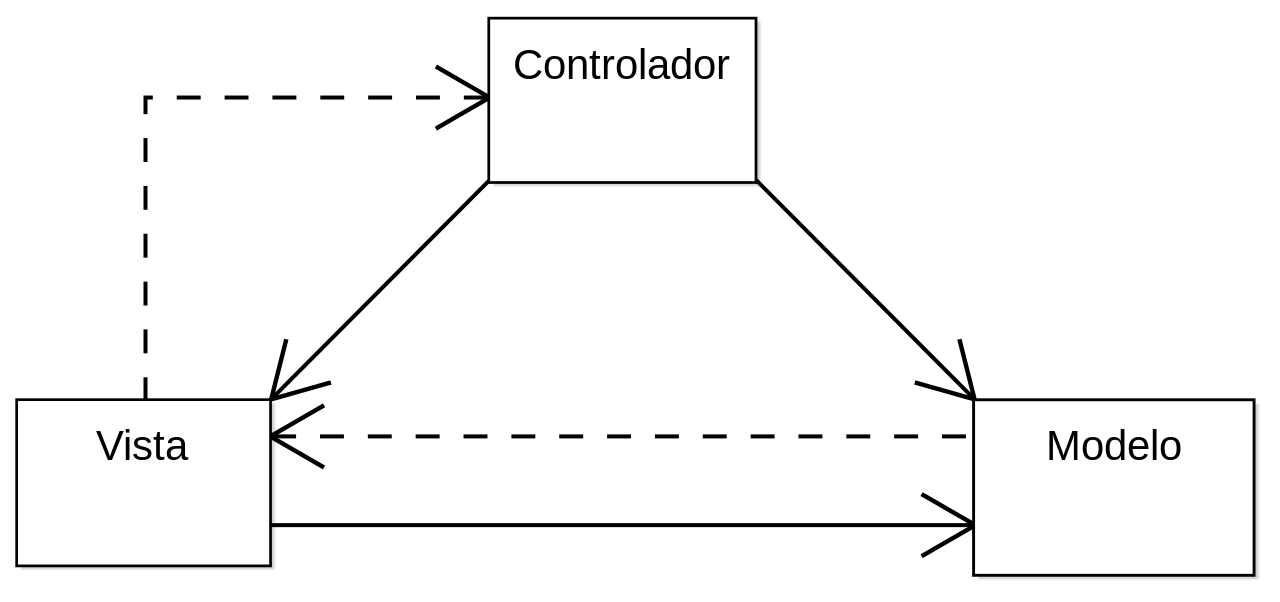
3.2. La Vista 6

3.3. El control 8

3.4. El test 10

4. Conclusiones y Recomendaciones 12

# **El patrón Modelo-Vista-Controlador (MVC)**



El patrón MVC (o como su nombre completo indica Modelo – Vista – Controlador) es un patrón de arquitectura de aplicaciones software y se compone de tres tipos de objetos:

* 1. **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.
  2. **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.

* 1. **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 encargar de direccionar al modelo la petición respectiva.

Por ejemplo el usuario quiere ver los clientes con apellido Álvarez, la petición va al controlador y el 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.

## **Historia**

El patrón MVC fue una de las primeras ideas en el campo de las interfaces gráficas de usuario y uno de los primeros trabajos en describir e implementar aplicaciones software en términos de sus diferentes funciones.5​

MVC fue introducido por Trygve Reenskaug (web personal) en Smalltalk-76 durante su visita a Xerox Parc6​7​ en los años 70, seguidamente, en los años 80, Jim Althoff y otros implementaron una versión de MVC para la biblioteca de clases de Smalltalk-80.8​ Solo más tarde, en 1988, MVC se expresó como un concepto general en un artículo9​ sobre Smalltalk-80.

En esta primera definición de MVC el controlador se definía como «el módulo que se ocupa de la entrada» (de forma similar a como la vista «se ocupa de la salida»). Esta definición no tiene cabida en las aplicaciones modernas en las que esta funcionalidad es asumida por una combinación de la 'vista' y algún framework moderno para desarrollo. El 'controlador', en las aplicaciones modernas de la década de 2000, es un módulo o una sección intermedia de código, que hace de intermediario de la comunicación entre el 'modelo' y la 'vista', y unifica la validación (utilizando llamadas directas o el «observer» para desacoplar el 'modelo' de la 'vista' en el 'modelo' activo10​).

Algunos aspectos del patrón MVC han evolucionado dando lugar a ciertas variantes del concepto original, ya que «las partes del MVC clásico realmente no tienen sentido para los clientes actuales»:11​

# **Ejemplo de aplicación en Java**

Para poder ejemplificar el uso del patrón MVC, crearemos un programa en Java que funcione como tipo de cambio para euros y pesetas, el cual nos permitirá hacer la conversión de la moneda con un valor de cambio previamente establecido (uno no inserta dicho valor en el programa)

* 1. **El modelo**

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;

}

}

* 1. **La Vista**

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);

}

}

* 1. **El control**

package com.ecodeup.controller;

import com.ecodeup.model.Cliente;

import com.ecodeup.view.ClienteView;

public class ClienteController {

//objetos vista y modelo

private ClienteView vista;

private Cliente modelo;

//constructor para inicializar el modelo y la vista

public ClienteController(Cliente modelo, ClienteView vista) {

this.modelo = modelo;

this.vista = vista;

}

//getters y setters para el modelo

public int getId() {

return modelo.getId();

}

public void setId(int id) {

this.modelo.setId(id);

}

public String getNombre() {

return modelo.getNombre();

}

public void setNombre(String nombre) {

this.modelo.setNombre(nombre);

}

public String getApellido() {

return modelo.getApellido();

}

public void setApellido(String apellido) {

this.modelo.setApellido(apellido);

}

//pasa el modelo a la vista para presentar los datos

public void actualizarVista() {

vista.imprimirDatosCliente(modelo.getId(),modelo.getNombre(), modelo.getApellido());

}

}

* 1. **Prueba:**

package com.ecodeup.mvc;

import com.ecodeup.controller.ClienteController;

import com.ecodeup.model.Cliente;

import com.ecodeup.view.ClienteView;

public class MvcDemo {

public static void main (String [] args){

// objeto vista, y modelo creado con el método estático

Cliente modelo= llenarDatosCliente();

ClienteView vista= new ClienteView();

//se crea un objeto controlador y se le pasa el modelo y la vista

ClienteController controlador= new ClienteController(modelo, vista);

// se muestra los datos del cliente

controlador.actualizarVista();

// se actualiza un cliente y se muestra de nuevo los datos

controlador.setNombre("Luis");

controlador.actualizarVista();

}

//método estático que retorna el cliente con sus datos

private static Cliente llenarDatosCliente() {

Cliente cliente = new Cliente();

cliente.setId(1);

cliente.setNombre("Elivar");

cliente.setApellido("Largo");

return cliente;

}

}

# **Conclusiones y Recomendaciones**

Después de haber recopilado información podemos concluir en que este patrón de arquitectura de software permite 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. Una mayor eficiencia a la hora de desarrollar una aplicación debido a la separación, relación de dependencia y estructura de las partes Modelo, Vista y Controlador.