UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ingeniería Industrial y de Sistemas

Sistemas UNI

Título: El Patrón MVC

Autor(es): Fabrizio Julio Castillo Castillo

Giro Alexandro Pinto López

Kevin Gerardo Rosado Sotomayor

Profesor(es): Eric Gustavo Coronel Castillo

Monografía presentada para el curso de Programación Java

Lima, enero del 2020

*Al profesor Gustavo Coronel,*

*que nos mostró una*

*nueva y sorprendente*

*forma de entender la programación.*

**Índice**

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

1.1. Historia 5

1.2. Funcionamiento 5

1.3. Flujo de control 5

1.4. Tipos de comportamiento MVC 6

2. MVC en aplicaciones web 8

2.1. La World Wide Web 8

2.2. Arquitectura MVC para web 8

3. Ejemplo de aplicación en Java 11

3.1. El modelo 11

3.2. La Vista 13

3.3. El control 16

3.4. El programa 17

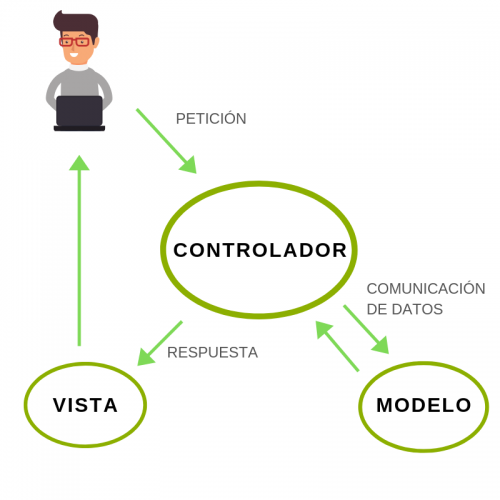
4. Conclusiones y Recomendaciones 18

# **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 que separa la lógica de negocio (model) de la interfaz de usuario (view).

MVC se compone de tres tipos de objetos:

* Modelo: El modelo encapsula el comportamiento y los datos correspondientes al dominio de la aplicación. Son los objetos de la aplicación, también conocida como lógica de negocio, o lógica de aplicación.
* Vista: Las vistas consultan el estado del modelo para mostrárselo al usuario.
* Controlador: Los controladores son los encargados de permitir que el usuario realice acciones.



Seguir este patrón tiene ventajas muy importantes tales como las siguientes:

* Facilita la evolución por separado de ambos aspectos: se puede trabajar por separado tanto el modelo como la vista y así ser más eficiente.
* Incrementa reutilización y flexibilidad: al separar el modelo de la vista, puede aplicarse el mismo modelo para diferentes tipos de vista o interfaz de usuario (por ejemplo, aplicación móvil, página web, etc.)

## **Historia**

Descrito por primera vez en 1979 para Smalltalk

Utilizado en múltiples frameworks:

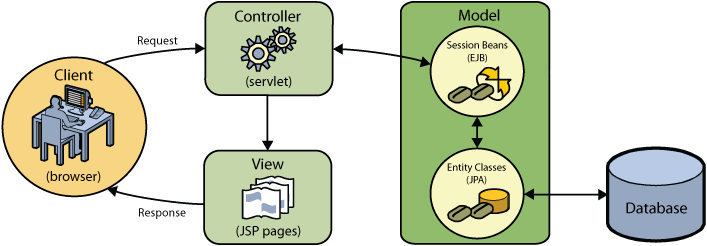
* Java Swing
* Java Enterprise Edition (J2EE)
* XForms (Formato XML estándar del W3C para la especificación de un modelo de proceso de datos XML e interfaces de usuario como formularios web)
* GTK+ (escrito en C, toolkit creado por Gnome para construir aplicaciones gráficas, inicialmente para el sistema X Window)
* ASP.NET MVC Framework (Microsoft)
* Google Web Toolkit (GWT, para crear aplicaciones Ajax con Java)
* Apache Struts (framework para aplicaciones web J2EE)
* Ruby on Rails (framework para aplicaciones web con Ruby)

## **Funcionamiento**

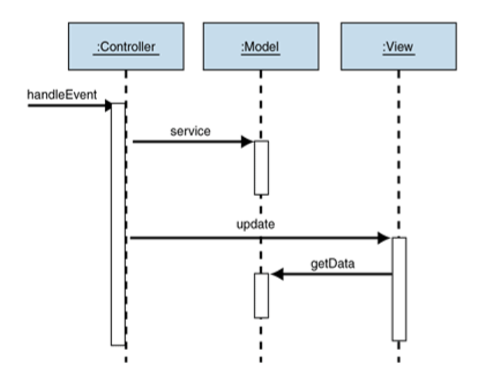
* Modelo-Vista-Controlador: Un modelo, varias vistas, varios controladores
* Las vistas y los controladores suelen estar muy relacionados
* Los controladores tratan los eventos que se producen en la interfaz gráfica (vista)
* Esta separación de aspectos de una aplicación da mucha flexibilidad al desarrollador

## **Flujo de control**

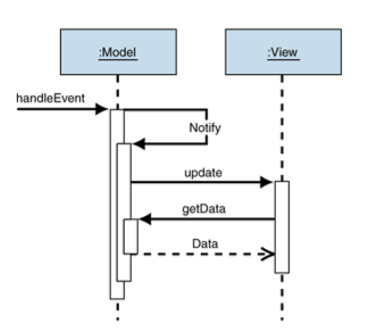
1. El usuario realiza una acción en la interfaz
2. El controlador trata el evento de entrada (Previamente se ha registrado)
3. El controlador notifica al modelo la acción del usuario, lo que puede implicar un cambio del estado del modelo (si no es una mera consulta)
4. Se genera una nueva vista. La vista toma los datos del modelo (El modelo no tiene conocimiento directo de la vista).
5. La interfaz de usuario espera otra interacción del usuario, que comenzará otro nuevo ciclo

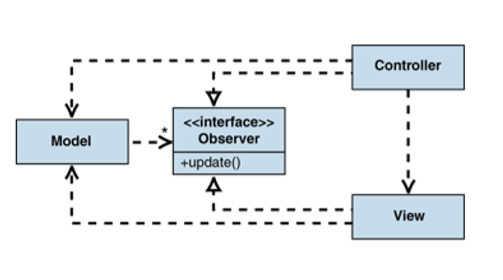


* 1. **Tipos de comportamiento MVC:**
* **Comportamiento pasivo:**
  + Utilizado cuando un controlador manipula el modelo exclusivamente.
  + El controlador modifica el modelo y le informa a la vista que este ha cambiado y debe ser refrescada.
  + En este escenario el modelo es completamente independiente de la vista y del controlador.

****

* **Comportamiento activo:**
  + Es usado cuando el modelo cambia de estado sin la intervención del controlador.
  + El modelo detecta los cambios a su estado interno cuando estos ocurren y deberá notificar a la vista para refrescarla.
  + En este escenario se crearía una dependencia entre el modelo y la vista.

****

****

# **MVC en aplicaciones web**

* 1. **La World Wide Web**

El diseño de interfaces Web es un tema complejo en el que no sólo intervienen procesos de diseño gráfico y programación, sino que también resultan imprescindibles aspectos de la arquitectura de la información, navegación, funcionalidad y, sobre todo, de la usabilidad (Belmonte, 2003). La ingeniería de la Web hace referencia a las metodologías, técnicas y herramientas que se utilizan para el desarrollo de aplicaciones Web complejas y de gran dimensión, en las que se apoya la evaluación, diseño, desarrollo, implementación y evolución de dichas aplicaciones Web. Ellas poseen determinadas características que lo hacen diferente del desarrollo de aplicaciones o software tradicional y sistemas de información. Es multidisciplinaria, aglutina contribuciones de: arquitectura de la información, ingeniería de hipermedia/hipertexto, diseño de interfaz de usuario, gráfico, usabilidad, análisis de sistemas, ingeniería de software, ingeniería de datos, indexado y recuperación de información, testeo, modelado. Así como simulación, despliegue de aplicaciones, operación de sistemas y gestión de proyectos (Méndez, 2008). El diseño Web no es un clon o subconjunto de la ingeniería de software, aunque ambas incluyen desarrollo de software y programación, la ingeniería de la Web utiliza principios de ingeniería de software, incluye nuevos enfoques, metodologías, herramientas, técnicas, guías y patrones para cubrir los requisitos únicos de las aplicaciones (Méndez, 2008).

* 1. **Arquitectura MVC para web**

El patrón Modelo, Vista y Controlador (MVC) es el más extendido para el desarrollo de aplicaciones donde se deben manejar interfaces de usuarios, éste se centra en la separación de los datos o modelo, y la vista, mientras que el controlador es el encargado de relacionar a estos dos (MacWilliams et al., 2003). Su principal característica es aislar la vista del modelo (Merlino, 2006; Ferro et al., 2003; Rivoir y Álvarez, 2005; Buhr et al., 1996; Corredera de Colsa, 2005). En la figura 1 se puede apreciar la separación de las tres capas y los componentes que la hacen funcional, por tener independencia entre capas, lo que hace que sea deseable para proyectos de grandes dimensiones (Ferro et al., 2003, Romero y Sánchez, 2003; Buhr et al., 1996; Gilart et al., 2005).

Las ventajas de usar el patrón MVC son:

a) Permitir la sustitución de las interfaces de usuario.

b) Generar componentes de las interfaces.

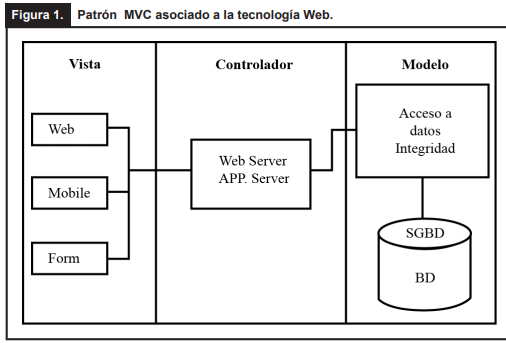
c) Diseñar vistas simultáneas del mismo modelo.

d) Aplicar fácilmente cambios de las interfaces.

También se han identificado ciertos problemas, como:

e) La complejidad aumenta rápidamente.

f) La vista y el modelo están muy acopladas. Considerando el acoplamiento como el grado de interdependencia entre las unidades de software (módulos, funciones, subrutinas) de un sistema informático. En este sentido en el patrón mvc, el acceso a datos depende directamente del mismo modelo que se mapea por medio de la consulta sql (vista), por tanto los dto y dao corresponden a una estructura muy acoplada a la vista, ya que los objetos de intercambio (dto) dependen directamente de los dao y la generación de los mismos dependen directamente del modelo.



* **Vista:** la página HTML
* **Controlador:** código que obtiene datos dinámicamente y genera el contenido HTML
* **Modelo:** la información almacenada en una base de datos o en XML junto con las reglas de negocio que transforman esa información (teniendo en cuenta las acciones de los usuarios)

# **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 ConversorEuros {

private double cambio;

public ConversorEuros ( double valorCambio ) {

// valor en la moneda de 1 euro

cambio = valorCambio;

}

public double eurosAmoneda (double cantidad) {

return cantidad \* cambio;

}

public double monedaAeuros (double cantidad) {

return cantidad / cambio;

}

}

public class ConversorEurosPesetas extends ConversorEuros

{ // Adaptador de clase

public ConversorEurosPesetas () {

super(166.386D);

}

public double eurosApesetas(double cantidad) {

return eurosAmoneda(cantidad);

}

public double pesetasAeuros(double cantidad) {

return monedaAeuros(cantidad);

}

}

public class ConversorEurosPesetas

{ // Adaptador de objetos

private ConversorEuros conversor;

public ConversorEurosPesetas () {

Conversor = new ConversorEuros(166.386D);

}

public double eurosApesetas(double cantidad) {

return conversor.eurosAmoneda(cantidad);

}

public double pesetasAeuros(double cantidad) {

return conversor.monedaAeuros(cantidad);

}

* 1. **La Vista**

Definimos una interfaz con las operaciones que el control puede necesitar para manipularla

public interface InterfazVista {

void setControlador(ControlConversor c);

void arranca(); // comienza la visualización

double getCantidad(); // cantidad a convertir

void escribeCambio(String s); //texto con la conversión

// Constantes que definen las posibles operaciones:

static final String AEUROS="Pesetas a Euros";

static final String APESETAS="Euros a Pesetas";

}

* Una ventana (JFrame) con
* Un campo de texto (JTextField) para indicarla entrada
* Una etiqueta (JLabel) para indicar el resultado de la
* conversión
* Dos botones (JButton) para las dos operaciones



public class VentanaConversor extends JFrame implements InterfazVista {

private JButton convertirApesetas;

private JButton convertirAeuros;

private JTextField cantidad;

private JLabel resultado;

public VentanaConversor () {

super("Conversor de Euros y Pesetas");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JPanel panelPrincipal = new JPanel();

panelPrincipal.setLayout(new BorderLayout(10,10));

cantidad = new JTextField(8);

JPanel panelaux = new JPanel(); panelaux.add(cantidad);

panelPrincipal.add(panelaux, BorderLayout.NORTH);

resultado = new JLabel("Indique una cantidad y pulse uno de los botones");

JPanel panelaux2 = new JPanel(); panelaux2.add(resultado);

panelPrincipal.add(panelaux2, BorderLayout.CENTER);

convertirApesetas = new JButton("A pesetas");

convertirApesetas.setActionCommand(APESETAS);

convertirAeuros = new JButton("A euros");

convertirAeuros.setActionCommand(AEUROS);

JPanel botonera = new JPanel();

botonera.add(convertirApesetas); botonera.add(convertirAeuros);

panelPrincipal.add(botonera, BorderLayout.SOUTH);

getContentPane().add(panelPrincipal);

}

}

// Métodos de la interfaz InterfazVista:

public void escribeCambio(String s) {

resultado.setText(s);

}

public double getCantidad() {

try {

return Double.parseDouble(cantidad.getText());

}

catch (NumberFormatException e) {

return 0.0D;

}

}

public void setControlador(ControlConversor c) {

convertirApesetas.addActionListener(c);

convertirAeuros.addActionListener(c);

}

public void arranca() {

pack();// coloca los componentes

setLocationRelativeTo(null);// centra la ventana en la pantalla

setVisible(true);// visualiza la ventana

}

* 1. **El control**

public class ControlConversor implements ActionListener {

private InterfazVista vista;

private ConversorEurosPesetas modelo;

public ControlConversor(InterfazVista vista, ConversorEurosPesetas modelo)

{

this.vista = vista;

this.modelo = modelo;

}

public void actionPerformed(ActionEvent evento) {

double cantidad = vista.getCantidad();

if ( evento.getActionCommand().equals(InterfazVista.AEUROS) ) {

vista.escribeCambio( cantidad + " pesetas son: “+ modelo.pesetasAeuros(cantidad) + " euros" );

}

else if ( evento.getActionCommand().equals(InterfazVista.APESETAS)) {

vista.escribeCambio( cantidad + " euros son: “+ modelo.eurosApesetas(cantidad) + " pesetas" );

}

else{

vista.escribeCambio( "ERROR" );

}

}

## **El programa**

public class ProgramaDeConversión {

public static void main(String[] args) {

// el modelo:

ConversorEurosPesetas modelo = new ConversorEurosPesetas();

// la vista:

InterfazVista vista = new VentanaConversor();

// y el control:

ControlConversor control = new ControlConversor (vista,

modelo);

// configura la vista

vista.setControlador(control);

// y arranca la interfaz (vista):

vista.arranca();

}

}

# **Conclusiones y Recomendaciones**

Tras haber recopilado información acerca del patrón MVC, incluida su aplicación web y un ejemplo programable en Java, podemos concluir en que este patrón de arquitectura de software ofrece 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, de esta forma el servicio ofrecido ante una problemática puede ser más cómodo y sencillo para el usuario, tomando en cuenta siempre la presentación de dicha aplicación que debe ser llamativa para los clientes y dinámica en todas las funcionalidades que pueda realizar.