2º D.A.M.

{MODELO VISTA CONTROLADOR}

Marta Tirador Gutiérrez

Contenido

Historia del MVC

El patrón de arquitectura MVC (Modelo-Vista-Controlador) tiene sus raíces en la década de 1970. Fue introducido por **Trygve Reenskaug**, un ingeniero de software noruego, mientras trabajaba en el desarrollo de Smalltalk en los laboratorios de investigación de Xerox PARC.

La idea detrás de MVC era proporcionar una estructura organizativa para el desarrollo de software que separara las preocupaciones relacionadas con la manipulación de datos, la presentación y la lógica del usuario.

¿Qué es el Modelo Vista Controlador?

MVC (Modelo-Vista-Controlador) es un patrón en el diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control. Este patrón divide una aplicación en tres componentes principales: el Modelo, la Vista y el Controlador.

Modelo (Model): Representa la estructura de datos y las reglas de negocio de la aplicación. En otras palabras, el Modelo se encarga de manejar los datos y las reglas que gobiernan esos datos. Puede incluir acceso a bases de datos, lógica de validación, cálculos y otras operaciones relacionadas con la manipulación de datos.

El Modelo también contribuye significativamente a una estructura más organizada y mantenible. Al separar las reglas de negocio de las preocupaciones de presentación y control, se logra una clara delimitación de responsabilidades. Esto conduce a módulos de código más cohesivos y desacoplados, lo que facilita la reutilización y el mantenimiento a largo plazo. Al centralizar la gestión de datos y la lógica, el Modelo evita la duplicación innecesaria de código y establece un flujo coherente y confiable de datos en toda la aplicación.

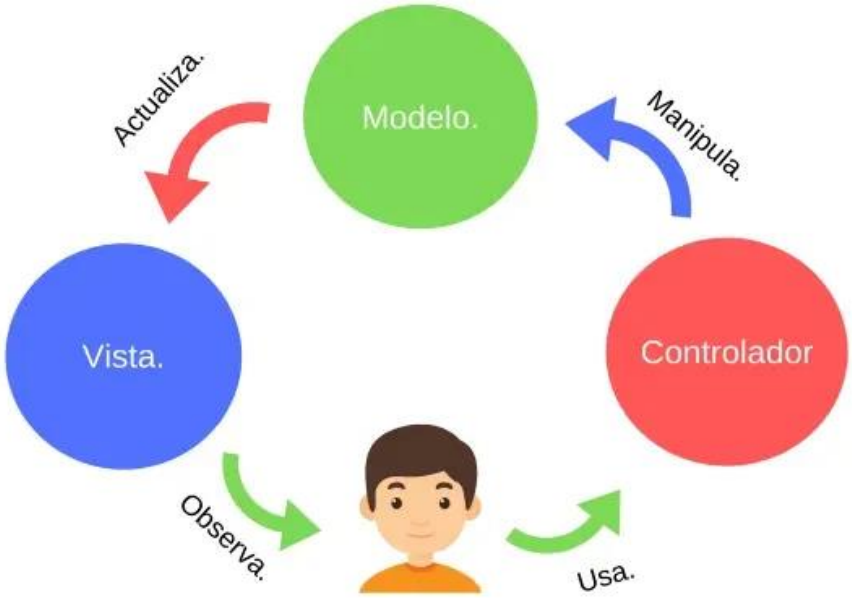
Vista (View): Es responsable de la presentación de los datos al usuario y de la interfaz de usuario. La Vista muestra la información contenida en el Modelo y también se encarga de recibir las interacciones del usuario, como clics de ratón o pulsaciones de teclas. La Vista no contiene las reglas de negocio y está diseñada para ser lo más pasiva posible, simplemente mostrando los datos proporcionados por el Modelo.

Controlador (Controller): Actúa como intermediario entre el Modelo y la Vista. Recibe las interacciones del usuario desde la Vista, procesa esa entrada y actualiza el Modelo en consecuencia. Además, el Controlador puede recibir actualizaciones del Modelo y reflejar esos cambios en la Vista. En resumen, el Controlador maneja el flujo de datos y eventos entre el Modelo y la Vista, implementa lógica(validar datos, tomar decisiones basadas en estados internos o realizar cálculos complejos antes de interactuar con el Modelo).

Flujo de datos y eventos

Los controladores, con sus reglas de negocio, hacen de puente entre el modelo y la vista. Además, los modelos pueden enviar datos a las vistas. El paso del flujo sería el siguiente:

1. El usuario realiza una solicitud a una aplicación. Esta solicitud le llega al controlador.
2. El controlador se comunica tanto con el modelo como con las vistas. A los modelos les solicita datos o les manda realizar actualizaciones de los datos y a las vistas les solicita la salida correspondiente.
3. Las vistas pueden solicitar más información a los modelos; cuando tienen toda la información, le envían al usuario la interfaz de salida.



¿Para qué se utiliza?

La funcionalidad del patrón Modelo-Vista-Controlador (MVC) radica en la separación de responsabilidades en una aplicación, lo que facilita el desarrollo, la mantenibilidad y la escalabilidad del código. Cada componente del patrón desempeña un papel específico y se comunica con los otros de manera definida, lo que permite un mejor control y organización del código.

¿Cómo se implementa?

El patrón Modelo-Vista-Controlador (MVC) se puede implementar en una variedad de lenguajes de programación y con numerosos frameworks. A continuación, se mencionan algunos de los lenguajes y frameworks más populares que utilizan el patrón MVC:

Java:

Frameworks MVC: Spring MVC, JavaServer Faces (JSF), Struts.

C#:

Frameworks MVC: ASP.NET MVC, ASP.NET Core MVC.

Python:

Frameworks MVC: Django, Flask (aunque Flask es más minimalista y no sigue estrictamente el patrón MVC, permite su implementación).

Ruby:

Frameworks MVC: Ruby on Rails.

JavaScript (Front-end):

Frameworks MVC: Angular, React (puede implementar patrones similares a MVC, como Flux o Redux).

JavaScript (Node.js - Back-end):

Frameworks MVC: Express.js (puede utilizarse siguiendo el estilo MVC), Sails.js.

PHP:

Frameworks MVC: Laravel, Symfony, CodeIgniter.

Swift (iOS):

Frameworks MVC: UIKit (utilizado con el patrón MVC en el desarrollo de aplicaciones iOS).

Kotlin (Android):

Frameworks MVC: Android no tiene un framework MVC específico, pero se puede implementar MVC en el desarrollo de aplicaciones Android.

.NET (General):

Frameworks MVC: ASP.NET MVC, ASP.NET Core MVC.

Es importante tener en cuenta que, aunque algunos frameworks se denominan explícitamente "MVC", la implementación exacta del patrón puede variar. Además, algunos lenguajes o frameworks pueden tener variaciones del patrón, como el Modelo-Vista-Presentador (MVP) o el Modelo-View-ViewModel (MVVM), que comparten conceptos similares, pero con enfoques ligeramente diferentes.

Beneficios que proporciona

En conjunto, estas funcionalidades proporcionan los siguientes beneficios:

-**Separación de Responsabilidades**: Cada componente tiene un propósito claro y se encarga de tareas específicas. Esto facilita la comprensión del código y permite realizar cambios en una parte del sistema sin afectar otras partes.

-**Reutilización del Código**: Al separar por partes, es más probable que puedas reutilizar componentes en diferentes partes de la aplicación o incluso en otras aplicaciones.

-**Mantenibilidad**: Facilita la identificación y corrección de errores, así como la realización de mejoras o actualizaciones en la aplicación.

-**Escalabilidad**: Permite escalar la aplicación de manera más eficiente, ya que los diferentes componentes pueden ser escalados independientemente según sea necesario.

-**Facilita las Pruebas**: La separación de responsabilidades facilita la realización de pruebas unitarias en los diferentes componentes, lo que contribuye a la calidad del software.

En resumen, la funcionalidad del patrón MVC se centra en proporcionar una estructura organizativa que mejora la modularidad, la claridad y la flexibilidad en el desarrollo de software.

Ejemplos

1º Java

Modelo: Representa los datos y la lógica de la aplicación

```
public class Modelo {
    private String mensaje;

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    public String getMensaje() {
        return mensaje;
    }
}
```

Vista: Presenta la información al usuario y captura la entrada del usuario

```
public class Vista {
    public void mostrarMensaje(String mensaje) {
        System.out.println("Mensaje: " + mensaje);
    }

    public String obtenerEntradaUsuario() {
```

```
        Scanner scanner = new Scanner(System.in);
        System.out.print("Ingrese un mensaje: ");
        return scanner.nextLine();
    }
}
```

Controlador: Gestiona la interacción entre el Modelo y la Vista

```
public class Controlador {
    private Modelo modelo;
    private Vista vista;

    public Controlador(Modelo modelo, Vista vista) {
        this.modelo = modelo;
        this.vista = vista;
    }

    public void actualizarMensaje() {
        String nuevoMensaje = vista.obtenerEntradaUsuario();
        modelo.setMensaje(nuevoMensaje);
    }

    public void mostrarMensaje() {
        String mensaje = modelo.getMensaje();
        vista.mostrarMensaje(mensaje);
    }
}
```

```
// Clase principal que une el modelo, la vista y el controlador
public class Principal {
    public static void main(String[] args) {
        Modelo modelo = new Modelo();
        Vista vista = new Vista();
        Controlador controlador = new Controlador(modelo, vista);

        // Actualizar el mensaje a través del controlador
        controlador.actualizarMensaje();

        // Mostrar el mensaje a través del controlador
        controlador.mostrarMensaje();
    }
}
```

En este ejemplo:

- Modelo representa los datos y la lógica de la aplicación.

- Vista se encarga de mostrar información al usuario y capturar la entrada del usuario.

- Controlador gestiona la interacción entre el Modelo y la Vista, actualizando el modelo según la entrada del usuario y mostrando la información al usuario.

La clase Principal es donde se instancian y se conectan el modelo, la vista y el controlador.

2º Ejemplo Java con framework Struts

Modelo: Mensaje.java

```
public class Mensaje {
   private String contenido;
   // Getters y setters
   // ...
}
```

Controlador: MensajeController.java

```
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Namespace;
import org.apache.struts2.convention.annotation.Result;
import com.opensymphony.xwork2.ActionSupport;

@Namespace("/mensaje")
public class MensajeAction extends ActionSupport {
   private Mensaje mensaje = new Mensaje();

   // Acción para mostrar el mensaje
   @Action(value = "mostrar", results = @Result(name = "success",
location = "/mostrar-mensaje.jsp"))
   public String mostrarMensaje() {
      return SUCCESS;
   }
```

```
    // Acción para mostrar el formulario
    @Action(value = "formulario", results = @Result(name = "success", location = "/formulario-mensaje.jsp"))
    public String mostrarFormulario() {
        return SUCCESS;
    }
    // Acción para guardar el mensaje
    @Action(value = "guardar", results = @Result(name = "success", location = "/mensaje/mostrar", type = "redirect"))
    public String guardarMensaje() {
        // Lógica para guardar el mensaje
        return SUCCESS;
    }
    // Getters y setters
    // ...
}
```

Vistas: mostrar mensaje.jsp y formulario-mensaje.jsp

mostrar-mensaje.jsp:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
```

```
<head>
    <meta charset="UTF-8">
    <title>Mostrar Mensaje</title>
</head>
<body>
    <h2>Mensaje:</h2>
    <p>${mensaje.contenido}</p>
</body>
</html>
```

formulario-mensaje.jsp:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Formulario de Mensaje</title>
</head>
<body>
    <h2>Ingrese un mensaje:</h2>
    <form action="<s:url action='guardar'/>" method="post">
        <input type="text" name="mensaje.contenido" required>
        <input type="submit" value="Guardar">
    </form>
</body>
</html>
```

En este ejemplo, Struts maneja las solicitudes y dirige a las acciones correspondientes en función de la configuración de las anotaciones @Action. El controlador MensajeAction interactúa con el modelo Mensaje y utiliza las vistas JSP para mostrar y recibir información del usuario.

Bibliografía

https://codingornot.com/mvc-modelo-vista-controlador-que-es-y-para-que-sirve

https://keepcoding.io/blog/que-es-la-arquitectura-mvc/

https://developer.mozilla.org/es/docs/Glossary/MVC

Apuntes - curso aplicaciones web con Java

ChatGPT

https://www.easyappcode.com/patron-de-diseno-mvc-que-es-y-como-puedo-utilizarlo

https://codigofacilito.com/

https://codejavu.blogspot.com/2013/06/ejemplo-modelo-vista-controlador.html