

# MODELO VISTA CONTROLADOR (MVC) Y PATRONES DE DISEÑO



Actividad de aprendizaje Contacto Docente N° 2


Marlon Santiago Loya Nasimba

Universidad de las Fuerzas Armadas – ESPE

1323: POO

Mgtr. Luis Enrique Jaramillo Montaña

# INTRODUCCIÓN



Los patrones de diseño ofrecen soluciones para resolver problemas frecuentes en el desarrollo de software. Su aplicación mejora la organización del código, facilitando su mantenimiento, reutilización y escalabilidad. Uno de los patrones más empleados en la creación de aplicaciones es el Modelo-Vista-Controlador (MVC, por sus siglas en inglés), esencial para desarrollar sistemas estructurados y bien organizados. Este enfoque establece una clara división de responsabilidades: el modelo gestiona la lógica de datos, la vista se encarga de la interfaz de usuario y el controlador coordina la interacción entre ambos. Adoptar el patrón MVC permite a los desarrolladores construir aplicaciones más fáciles de mantener y expandir, optimizando el proceso de desarrollo. Asimismo, la incorporación de patrones de diseño en la programación es una estrategia eficaz para abordar desafíos recurrentes con soluciones eficientes. Estos patrones proporcionan estructuras comprobadas para resolver problemas habituales en el diseño de sistemas, brindando un marco conceptual sólido que favorece la creación de software flexible y robusto.

# MODELO-VISTA-CONTROLADOR (MVC)

El patrón Modelo-Vista-Controlador (MVC) es una metodología esencial en el desarrollo de software que organiza una aplicación en tres partes interconectadas, cada una con funciones específicas. Esta estructura facilita la administración y el mantenimiento del código al establecer una clara división de responsabilidades. Gracias a MVC, los desarrolladores pueden mejorar la organización del software y trabajar en distintos componentes sin afectar el resto del sistema, lo que también favorece la escalabilidad y simplifica las pruebas.

**Modelo:** Gestiona la lógica de datos, responde a solicitudes del controlador, actualiza la información y notifica a la vista sobre cualquier cambio. Además, valida los datos antes de modificarlos, se encarga de su almacenamiento y emplea patrones como el observador para informar a otros componentes

**Vista:** Se encarga de mostrar los datos al usuario de forma clara y comprensible. Su función principal es reflejar el estado del modelo mediante una interfaz visual que permite la interacción del usuario. Además, se actualiza dinámicamente en función de los cambios en el modelo.

**Controlador:** Funciona como intermediario entre la vista y el modelo. Recibe las acciones del usuario, las procesa y comunica los cambios al modelo, asegurándose de que la vista refleje las modificaciones en los datos.





## EJEMPLO DE IMPLEMENTACIÓN MVC

Imaginemos una pequeña aplicación que muestra la información de un estudiante y permite actualizar su nombre en la base de datos (o en la estructura de datos interna, para simplificar)

### Modelo

```
1 v public class Student {
2     private String nombre;
3     private String matricula;
4
5     // Constructor
6 v public Student(String nombre, String matricula) {
7     this.nombre = nombre;
8     this.matricula = matricula;
9 }
10
11 // Getters y Setters
12 v public String getNombre() {
13     return nombre;
14 }
15
16 v public void setNombre(String nombre) {
17     this.nombre = nombre;
18 }
19
20 v public String getMatricula() {
21     return matricula;
22 }
23
24 v public void setMatricula(String matricula) {
25     this.matricula = matricula;
26 }
27 }
28
```

### Vista

```
1 v public class StudentView {
2 v     public void mostrarDetallesEstudiante(String nombre, String matricula) {
3         System.out.println("Información del Estudiante:");
4         System.out.println("Nombre: " + nombre);
5         System.out.println("Matrícula: " + matricula);
6     }
7 }
8
```

### Controlador

```
1 v public class StudentController {
2     private Student modelo;
3     private StudentView vista;
4
5 v public StudentController(Student modelo, StudentView vista) {
6     this.modelo = modelo;
7     this.vista = vista;
8 }
9
10 // Métodos para actualizar datos en el modelo
11 v public void setNombreEstudiante(String nombre) {
12     modelo.setNombre(nombre);
13 }
14
15 v public String getNombreEstudiante() {
16     return modelo.getNombre();
17 }
18
19 v public void setMatriculaEstudiante(String matricula) {
20     modelo.setMatricula(matricula);
21 }
22
23 v public String getMatriculaEstudiante() {
24     return modelo.getMatricula();
25 }
26
27 // Método para actualizar la vista con la información actual
28 v public void actualizarVista() {
29     vista.mostrarDetallesEstudiante(modelo.getNombre(), modelo.getMatricula());
30 }
31 }
32
```

De esta forma, con una clase Main, se instancia todo y se simulará, por ejemplo, la modificación del nombre del estudiante

```
1  public class Main {  
2      public static void main(String[] args) {  
3          // Crear el modelo con datos iniciales  
4          Student modelo = new Student("Carlos", "A00123");  
5  
6          // Crear la vista  
7          StudentView vista = new StudentView();  
8  
9          // Crear el controlador, inyectando modelo y vista  
10         StudentController controlador = new StudentController(modelo, vista);  
11  
12         // Mostrar datos iniciales  
13         controlador.actualizarVista();  
14  
15         // Supongamos que se quiere actualizar el nombre del estudiante  
16         controlador.setNombreEstudiante("María");  
17  
18         // Volver a mostrar los datos para ver el cambio  
19         System.out.println("\nDespués de la actualización:");  
20         controlador.actualizarVista();  
21     }  
22 }  
23
```

**Modelo:** Se pueden crear pruebas unitarias que verifiquen que la clase Student maneja bien sus getters y setters. Si intentamos asignar valores inválidos (por ejemplo, un nombre vacío), podríamos añadir validaciones al modelo y comprobar que funcionan.

**Vista:** La vista por sí sola, en este ejemplo, imprime datos. Podemos testear que se llama al método correcto con los parámetros correctos.

**Controlador:** Comprobamos que, tras solicitar un cambio de nombre, el modelo efectivamente contenga el nombre nuevo. Verificamos que cuando llamamos a actualizarVista(), la vista reciba la información adecuada.

# PATRONES DE DISEÑO

Los patrones de diseño son estrategias estructuradas que permiten resolver problemas recurrentes en el desarrollo de software. Funcionan como plantillas reutilizables que pueden adaptarse a distintos contextos, proporcionando soluciones comprobadas y eficientes. Su uso mejora la organización del código, facilita la comunicación entre desarrolladores y fomenta la escalabilidad y el mantenimiento de las aplicaciones. Además, ofrecen documentación clara y detallada, sirviendo como guía tanto para la implementación como para la colaboración en proyectos futuros. La relevancia de los patrones de diseño en el desarrollo de software radica en su capacidad para optimizar la calidad, la eficiencia y la mantenibilidad del código. No solo brindan soluciones validadas para desafíos comunes, sino que también aportan beneficios clave que contribuyen al éxito de un proyecto.

- Soluciones eficientes y probadas:
- Mayor claridad y comprensión del código:
- Optimización del proceso de desarrollo:
- Consistencia en la arquitectura del software: .
- Mejor comunicación en el equipo:



# TIPOS DE PATRONES

Los patrones de diseño se dividen en distintas categorías según su función y aplicación en el desarrollo de software. Esta clasificación permite una mejor comprensión y selección de los patrones más adecuados según las necesidades específicas de cada proyecto. A continuación, se presentan las principales categorías:

- **Patrones de creación :** Los patrones de creación se centran en el proceso de creación de objetos, proporcionando mecanismos para la inicialización y configuración de instancias. Estos patrones buscan optimizar y flexibilizar la creación de objetos, asegurando que las instancias se generen de manera eficiente y de acuerdo con las necesidades del sistema.
- **Patrones de estructura:** Los patrones de estructura los cuales ayudan a definir la estructura de las relaciones entre las clases y los objetos.
- **Patrones de comportamiento:** Los patrones de comportamiento se centran en la interacción entre objetos y la distribución de responsabilidades. Estos patrones buscan definir cómo los objetos colaboran entre sí, proporcionando soluciones flexibles y eficientes para la comunicación y coordinación en el sistema

## Patrones de creación

### Ejemplo:

Supongamos que en nuestra aplicación necesitamos una conexión única a la base de datos y queremos asegurarnos de que dicha conexión sea accedida por múltiples partes del sistema sin que se creen múltiples instancias de la misma. Para ello, podemos utilizar el patrón Singleton.

```
1  public class DatabaseConnection {
2
3      // 1. Variable estática que mantendrá la única instancia
4      private static DatabaseConnection instance;
5
6      // 2. Constructor privado para evitar instancias directas
7      private DatabaseConnection() {
8          // Aquí podrías colocar la lógica para establecer la conexión
9          // con la base de datos, cargar drivers, etc.
10         System.out.println("Conexión establecida con la base de datos.");
11     }
12
13     // 3. Método estático para obtener la instancia única
14     public static DatabaseConnection getInstance() {
15         if (instance == null) {
16             instance = new DatabaseConnection();
17         }
18         return instance;
19     }
20
21     // Métodos para interactuar con la base de datos
22     public void ejecutarConsulta(String sql) {
23         System.out.println("Ejecutando: " + sql);
24         // Lógica para ejecutar la consulta
25     }
26
27     public void cerrarConexion() {
28         System.out.println("Cerrando la conexión con la base de datos...");
29         // Lógica para cerrar la conexión
30     }
31 }
32
```

Uso del Singleton en otra parte de la aplicación  
Suponiendo que en la clase principal (o en distintos módulos) queremos usar la misma conexión. Solo vamos a llamar a `getInstance()` cada vez que necesitemos la conexión. Si ya estaba creada, obtendremos la misma referencia; si no, la creará en el momento.

```
1  public class Main {
2      public static void main(String[] args) {
3          // Solicitar instancia #1
4          DatabaseConnection conn1 = DatabaseConnection.getInstance();
5          conn1.ejecutarConsulta("SELECT * FROM usuarios");
6
7          // Solicitar instancia #2
8          DatabaseConnection conn2 = DatabaseConnection.getInstance();
9          conn2.ejecutarConsulta("SELECT * FROM productos");
10
11         // Verificamos si las referencias son las mismas
12         if (conn1 == conn2) {
13             System.out.println("Ambas referencias apuntan a la misma instancia de DatabaseConnection.");
14         }
15
16         // Cerramos la conexión al terminar
17         conn1.cerrarConexion();
18     }
19 }
20
```



## Patrones de Estructura

### Ejemplo

Imagina que tienes dispositivos europeos que necesitan enchufes de 2 pines (típico en Europa), pero estás en un país donde los enchufes son de 3 pines (típico en América). Para que tu dispositivo funcione, necesitas un adaptador.

```
1 // Main.java
2 public class Main {
3     public static void main(String[] args) {
4         // Tenemos un enchufe americano
5         EnchufeAmericano enchufeUSA = new EnchufeAmericano();
6
7         // Nuestro cliente espera un EnchufeEuropeo, así que
8         // usamos el Adaptador para "convertir" de enchufe americano a europeo
9         EnchufeEuropeo adaptado = new AdaptadorAmericanoAEuropeo(enchufeUSA);
10
11        // Ahora el cliente puede "ver" al enchufe adaptado como si fuera europeo
12        adaptado.conectarDosPines();
13    }
14 }
15
```

```
1 // EnchufeAmericano.java
2 public class EnchufeAmericano {
3     public void conectarTresPines() {
4         System.out.println("Enchufe Americano de 3 pines conectado.");
5     }
6 }
7
```

```
1 // AdaptadorAmericanoAEuropeo.java
2 public class AdaptadorAmericanoAEuropeo implements EnchufeEuropeo {
3
4     private EnchufeAmericano enchufeAmericano;
5
6     public AdaptadorAmericanoAEuropeo(EnchufeAmericano enchufeAmericano) {
7         this.enchufeAmericano = enchufeAmericano;
8     }
9
10    @Override
11    public void conectarDosPines() {
12        // Aquí hacemos la "traducción" de la llamada
13        System.out.println("Adaptador: convirtiendo enchufe de 2 pines a 3 pines...");
14        enchufeAmericano.conectarTresPines();
15    }
16 }
17
```

```
1 // Main.java
2 public class Main {
3     public static void main(String[] args) {
4         // Tenemos un enchufe americano
5         EnchufeAmericano enchufeUSA = new EnchufeAmericano();
6
7         // Nuestro cliente espera un EnchufeEuropeo, así que
8         // usamos el Adaptador para "convertir" de enchufe americano a europeo
9         EnchufeEuropeo adaptado = new AdaptadorAmericanoAEuropeo(enchufeUSA);
10
11        // Ahora el cliente puede "ver" al enchufe adaptado como si fuera europeo
12        adaptado.conectarDosPines();
13    }
14 }
15
```

## Patrones de comportamiento

### Ejemplo

Tenemos un ejemplo sencillo de un patrón de comportamiento. En el cual usaremos el Patrón Observer (Observador), que es muy común en escenarios donde uno o varios objetos (observadores) necesitan ser notificados cuando el estado de otro objeto (el sujeto u observable) cambia.

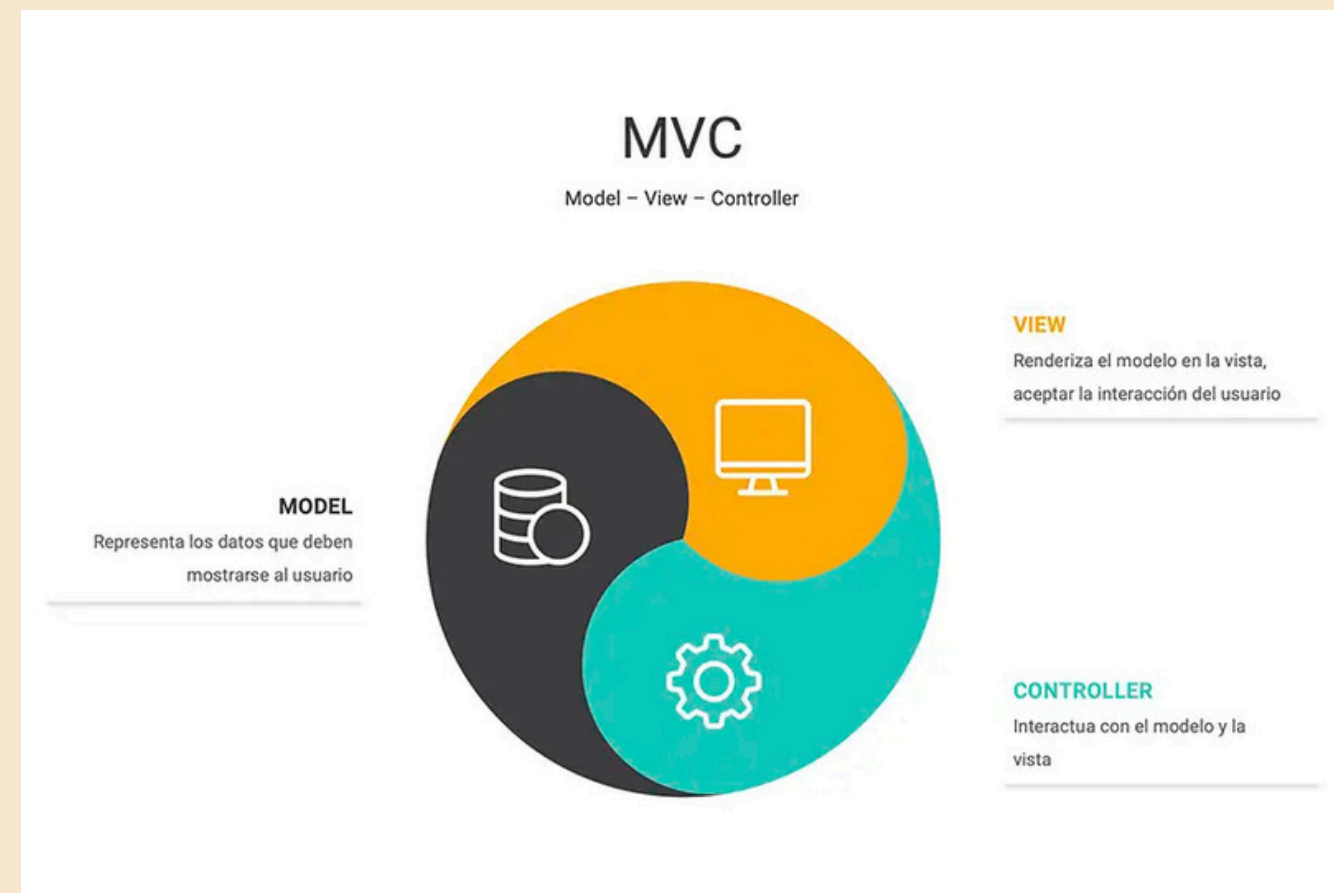
```
1  public class Main {
2      public static void main(String[] args) {
3          // 1. Creamos el sujeto (Canal de notificaciones)
4          CanalNotificaciones canal = new CanalNotificaciones();
5
6          // 2. Creamos observadores
7          Observer impresora1 = new ImpresoraNotificaciones("Impresora #1");
8          Observer impresora2 = new ImpresoraNotificaciones("Impresora #2");
9          HistorialNotificaciones historial = new HistorialNotificaciones();
10
11         // 3. Los registramos en el canal (attach)
12         canal.attach(impresora1);
13         canal.attach(impresora2);
14         canal.attach(historial);
15
16         // 4. Publicamos un mensaje
17         canal.publicarMensaje("Bienvenidos al canal de notificaciones.");
18
19         System.out.println("-----");
20
21         // 5. Desuscribimos a una impresora
22         canal.detach(impresora2);
23
24         // 6. Publicamos otro mensaje
25         canal.publicarMensaje("Segundo mensaje de actualización.");
26
27         // 7. Observamos los datos finales en el historial
28         System.out.println("Historial final: " + historial.getMensajesRecibidos());
29     }
30 }
31
```

### Salida en la consola:

```
Impresora #1 recibió el mensaje: Bienvenidos al canal de notificaciones.
Impresora #2 recibió el mensaje: Bienvenidos al canal de notificaciones.
Historial actualizado, total de mensajes: 1
-----
Impresora #1 recibió el mensaje: Segundo mensaje de actualización.
Historial actualizado, total de mensajes: 2
Historial final: [Bienvenidos al canal de notificaciones., Segundo mensaje de actualización.]
```

# CONCLUSIONES

El uso de patrones de diseño, como MVC y otros relevantes, permite mejorar la organización del código, promoviendo una separación clara de responsabilidades, facilitando el mantenimiento y favoreciendo la reutilización de componentes. Implementar estos patrones adecuadamente resulta en sistemas más flexibles, escalables y fáciles de modificar en el futuro.



# BIBLIOGRAFÍA

Fernández Romero, Y., & Díaz González, Y. (2012). Patrón Modelo-Vista-Controlador. Telem@tica (La Habana), 11(1), 47-57.

Sagredo, J. G. C., Espinosa, A. T., Reyes, M. M., & García, M. D. L. L. (2012). Automatización de la codificación del patrón modelo vista controlador (MVC) en proyectos orientados a la Web. CIENCIA ergo-sum, Revista Científica Multidisciplinaria de Prospectiva, 19(3), 239-250.

Cosmin, P. I. (2016). Patrones de Diseño. MoleQla: revista de Ciencias de la Universidad Pablo de Olavide, (23), 36.

Guerrero, C. A., Suárez, J. M., & Gutiérrez, L. E. (2013). Patrones de Diseño GOF (The Gang of Four) en el contexto de Procesos de Desarrollo de Aplicaciones Orientadas a la Web. Información tecnológica, 24(3), 103





# ¡GRACIAS!

