Qué es el patrón MVC

MVC era inicialmente un **patrón arquitectural**, un modelo o guía que **expresa cómo organizar y estructurar los componentes de un sistema software**, sus responsabilidades y las relaciones existentes entre cada uno de ellos.

Su nombre, MVC, parte de las iniciales de Modelo-Vista-Controlador (*Model-View-Controller*, en inglés), que son las capas o grupos de componentes en los que organizaremos nuestras aplicaciones bajo este paradigma.

Es a menudo considerado también un **patrón de diseño de la capa de presentación**, pues define la forma en que se organizan los componentes de presentación en sistemas distribuidos.

Por tanto, teniendo en cuenta su antigüedad (1980), es obvio que **ni siquiera fue ideado expresamente para sistemas web** aunque ahora se use mucho en ellas. Existen implementaciones para todo tipo de sistemas (escritorio, clientes y servidores web, servicios web, *Single Page Applications* o SPA, etc.) y lenguajes (Smalltalk, Java, Ruby, C++, C#, Python, PHP, JavaScript, NodeJS, etc.).

La arquitectura MVC propone, independientemente de las tecnologías o entornos en los que se base el sistema a desarrollar, la **separación de los componentes de una aplicación en tres grupos** (o capas) principales: **el modelo, la vista, y el controlador**, y describe **cómo se relacionarán entre ellos** para mantener una estructura organizada, limpia y con un acoplamiento mínimo entre las distintas capas.

El Modelo

En la capa Modelo encontraremos siempre una representación de los **datos del dominio**, es decir, aquellas entidades que nos servirán para almacenar información del sistema que estamos desarrollando. Por ejemplo, si estamos desarrollando una aplicación de facturación, en el modelo existirán las clases `Factura`, `Cliente` o `Proveedor`, entre otras.

Si nuestra aplicación forma parte de un sistema distribuido, es decir, consume servicios prestados por otros sistemas, en el Modelo encontraremos **las clases de transferencia de datos** (DTO, *Data Transfer Objects*) que nos permitirán intercambiar información con ellos. (DTO = Clases creadas expresamente para obtener datos de la BBDD)

Asimismo, encontraremos la **lógica de negocio** de la aplicación, es decir, la implementación de las reglas, acciones y restricciones que nos permiten gestionar las entidades del dominio. Será por tanto el responsable de que el sistema se encuentre siempre en un estado consistente e íntegro.

Por último, el Modelo será también el encargado de **gestionar el almacenamiento y recuperación de datos y entidades del dominio**, es decir, incluirá mecanismos de persistencia o será capaz de interactuar con ellos. Dado que habitualmente la persistencia se delega a un motor de bases de datos, es muy frecuente encontrar en el Modelo la implementación de componentes tipo DAL (*Data Access Layer*, o Capa de Acceso a Datos) y ORMs (Mapeador de BBDD a código).

El Modelo contiene principalmente las entidades que representan el dominio, la lógica de negocio, y los mecanismos de persistencia de nuestro sistema.

La Vista

Los componentes de la Vista son los responsables de **generar la interfaz** de nuestra aplicación, es decir, de componer las pantallas, páginas, o cualquier tipo de resultado utilizable por el usuario o cliente del sistema. De hecho, suele decirse que **la Vista es una representación del estado del Modelo** en un momento concreto y en el contexto de una acción determinada.

Por ejemplo, si un usuario está consultando una factura a través de una aplicación web, la Vista se encargará de representar visualmente el estado actual de la misma en forma de página visualizable en su navegador. Si en un contexto B2B el cliente de nuestro sistema es a su vez otro sistema, la vista podría ser un archivo XML con la información solicitada. En ambos casos se trataría de la misma factura, es decir, la misma entidad del Modelo, pero su representación es distinta en función de los requisitos.

Cuando las vistas componen la interfaz de usuario de una aplicación, deberán contener los **elementos de interacción** que permitan al usuario enviar información e invocar acciones en el sistema, como botones, cuadros de edición o cualquier otro tipo de elemento, convenientemente adaptados a la tecnología del cliente.

En la Vista encontraremos los componentes responsables de generar la interfaz con el exterior, por regla general, aunque no exclusivamente, el UI de nuestra aplicación.

El Controlador

La misión principal de los componentes incluidos en el Controlador es actuar como **intermediarios entre el usuario y el sistema**. Serán capaces de capturar las acciones de éste sobre la Vista, como puede ser la pulsación de un botón o la selección de una opción de menú, interpretarlas y actuar en función de ellas. Por ejemplo, retornando al usuario una nueva vista que represente el estado actual del sistema, o invocando a acciones definidas en el Modelo para consultar o actualizar información.

Realizarán también tareas de **transformación de datos** para hacer que los componentes de la Vista y el Modelo se entiendan. Así, traducirán la información enviada desde la interfaz, por ejemplo los valores de campos de un formulario recibidos mediante el protocolo HTTP, a objetos que puedan ser comprendidos por el Modelo, como pueden las clases o las entidades del dominio.

Y de la misma forma, el Controlador tomará la información procedente del Modelo y la adaptará a formatos o estructuras de datos que la Vista sea capaz de manejar.

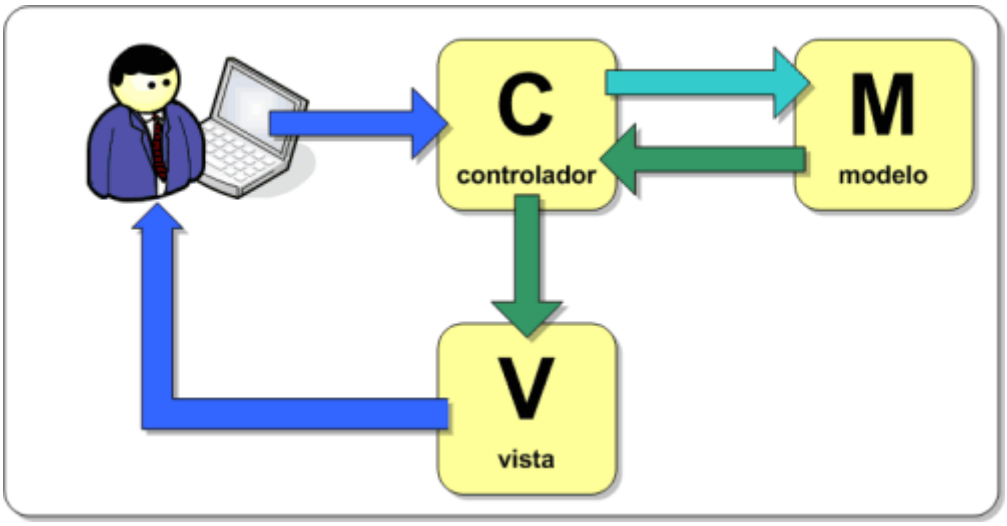
Por todo ello, podríamos considerar **el Controlador como un coordinador general del sistema**, que regula la navegación y el flujo de información con el usuario, ejerciendo también como intermediario entre la capa de Vista y el Modelo.

En el Controlador se encuentran los componentes capaces de procesar las interacciones del usuario, consultar o actualizar el Modelo, y seleccionar las Vistas apropiadas en cada momento.

Relación entre Modelo, Vista y Controlador

Nota: existen distintas variantes del patrón MVC. Aquí estamos considerando la utilizada más frecuentemente por los desarrolladores.

El siguiente diagrama refleja las relaciones existentes entre los componentes del Modelo, Vista y Controlador, y de éstos a su vez con el usuario, o cliente, del sistema:



Como se muestra en el diagrama, las acciones e información procedentes del usuario serán recogidas exclusivamente por los Controladores. Ningún componente de otra capa debe acceder a los datos generados desde el cliente, de la misma forma que sólo los componentes de la Vista estarán autorizados a generar interfaces de usuario con las que enviar información de retorno.

Destaca también el **papel central del Controlador**. Tiene acceso bidireccional al Modelo, es decir, será capaz tanto de actualizar su estado, invocando por ejemplo métodos o acciones incluidos en su lógica de negocio, como de consultar la información que sea necesaria para completar sus tareas. Sin embargo, en ningún caso el Modelo será consciente o mostrará acoplamiento alguno respecto a las clases Controlador que lo están utilizando, ni conocerá las distintas representaciones (Vistas) que pueden realizarse de él cara al usuario.

Por otra parte, el Controlador es el encargado de seleccionar la Vista más apropiada en función de la acción llevada a cabo por el usuario, suministrándole toda la información que necesite para componer la interfaz. Para pasar esta información, el Controlador puede usar clases del Modelo o, lo que es más habitual, clases específicamente diseñadas para ello, denominadas *View-Models*, que contendrán toda la información que la Vista necesite para maquetarse y mantendrá a ésta aislada de los cambios en el Modelo.

La responsabilidad de la Vista, por tanto, se reduce a **generar la interfaz partiendo de los datos** que le suministre el controlador.

Ventajas y desventajas de patrón MVC

El uso del patrón MVC ofrece múltiples ventajas sobre otras maneras de desarrollar aplicaciones con interfaz de usuario, y en especial para la Web.

- La clara separación de responsabilidades impuesta por el uso del patrón MVC hace que los componentes de nuestras aplicaciones tengan sus misiones bien definidas. Por lo tanto, nuestros sistemas serán **más limpios**, **simples**, más fácilmente **mantenibles** y, a la postre, más **robustos**.
- **Mayor velocidad de desarrollo en equipo**, que es consecuencia de lo anterior, ya que al estar separado en tres partes tan diferenciadas, diferentes programadores pueden ocuparse de cada parte en paralelo. Esto la hace **ideal para el desarrollo de aplicaciones grandes**.
- **Múltiples vistas** a partir del mismo modelo, pudiendo reaprovechar mucho mejor los desarrollos y asegurando consistencia entre ellas.
- **Facilidad para realización de pruebas unitarias**.

Pero, por supuesto, no todo es siempre maravilloso, así que su uso presenta también algunas desventajas, entre las que cabe destacar:

- **Hay que ceñirse a las convenciones y al patrón**. El uso de las convenciones impuestas por el *framework* y la estructura propuesta por el patrón arquitectural MVC nos obliga a ceñirnos a las mismas, lo que puede resultar a veces algo tedioso si lo comparamos con la forma habitual de trabajar con otros *frameworks* que dan más libertad al desarrollador. La división impuesta por el patrón MVC obliga a mantener un mayor número de archivos, incluso para tareas simples.
- **Curva de aprendizaje**. Dependiendo del punto de partida, el salto a MVC puede resultar un cambio radical y su adopción requerirá cierto esfuerzo. Además, utilizarlo implica conocer bien las tecnologías subyacentes con las que se implemente: la plataforma de programación utilizada, además de la tecnología utilizada para la interfaz de usuario (HTML, CSS, JavaScript en el caso de la Web).