Patrones de capa de presentación

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho

Universitat d'Alacant
Universidad de Alicante
Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Contenidos

Introducción

Introducción

En una arquitectura en capas, la **Capa de Presentación** tiene las siguientes responsabilidades:

- Gestionar la interacción con el usuario
- Comunicarse con las capas inferiores que proveen las funcionalidades deseadas
- Mostrar/actualizar la información como resultado de las llamadas a la lógica de negocio

Para estructurar correctamente el código de la capa de presentación conviene distribuir estas responsabilidades entre distintos objetos.

Introducción

Los patrones estructurales más usados para la capa de presentación son: [Fowler, 2003, Osmani, 2015]

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)

Para todos ellos se considera que el **Modelo** representa los objetos del modelo de dominio en la **capa de lógica de negocio**.

Los demás objetos (**Vista** y **Controlador/Presenter/ViewModel**) pertenecen a la **capa de presentación**.

Introducción

Estos patrones son independientes del tipo de aplicación (escritorio, web, móvil, ...).

Dependiendo del framework y de las características de la aplicación será más sencillo emplear unos u otros.

- Normalmente los frameworks para aplicaciones web de servidor están preparados para usar el patrón MVC
- La mayoría de frameworks para aplicaciones web cliente (*frontend*) permite usar cualquiera de los tres

Patrón Model-View-Controller

Model-View-Controller (MVC)

Aunque aparentemente se trata de un patrón sencillo, es uno de los patrones con más variantes y sobre el que menos consenso hay.

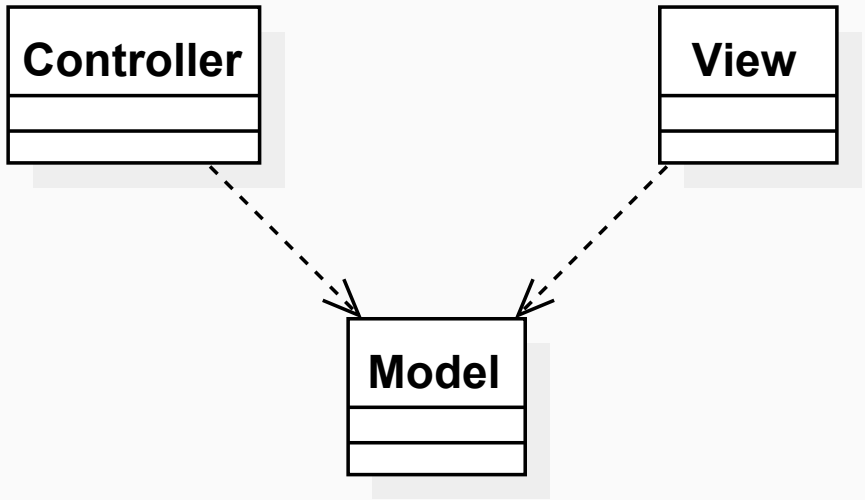
Hoy en día es el **patrón más utilizado en aplicaciones web**, aunque de forma muy distinta a como fue diseñado en sus orígenes.

MVC original

El patrón MVC fue concebido inicialmente para pequeños componentes gráficos en Smalltalk-80.

Cada componente gráfico (cuadro de texto, checkbox, etc.) tiene una vista y un controlador:

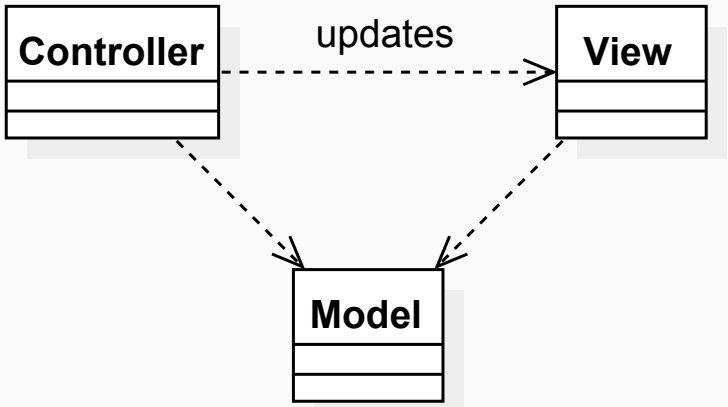
- La vista se encarga de dibujar el componente (había que escribir el código para esto) a partir de los datos del modelo
- El controlador recibe la interacción del usuario y manipula el modelo



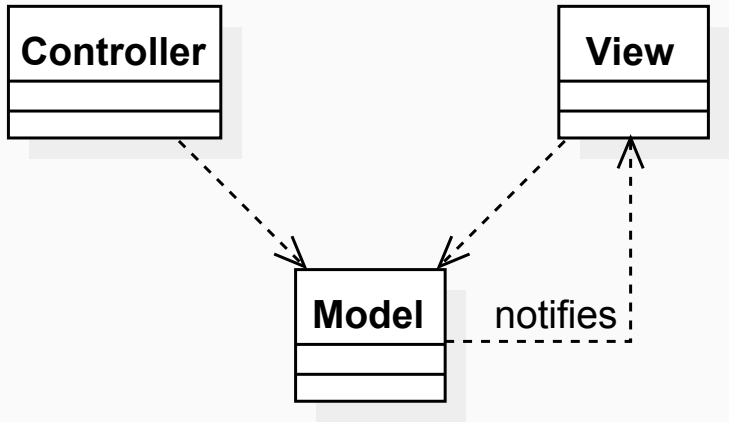
MVC original

Hay dos posibilidades para implementar la forma en que se actualiza la vista:

- El controlador actualiza la vista después de manipular el modelo
- El modelo notifica a la vista para que se actualize (**modelo activo**)



Modelo pasivo

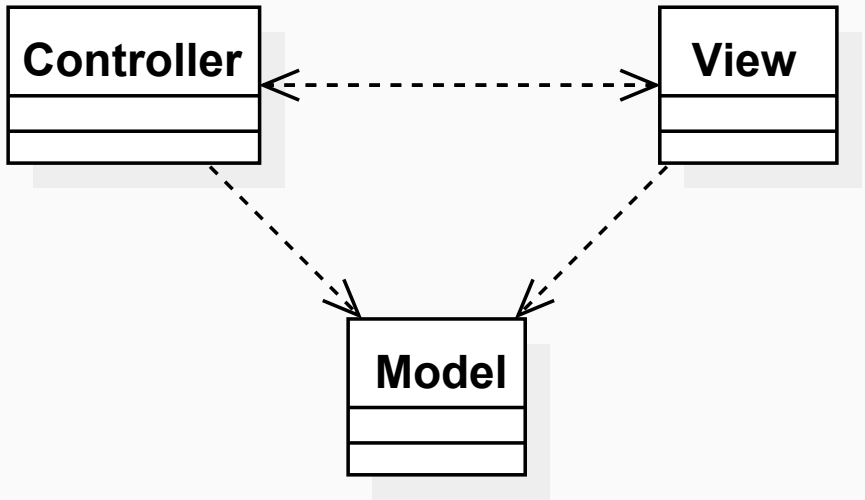


Modelo activo

MVC en aplicaciones web

En aplicaciones web simples (sin clientes enriquecidos con JavaScript) la comunicación entre objetos es distinta:

- La vista gestiona la interacción con el usuario y notifica al controlador
- El controlador manipula el modelo y decide qué vista mostrar a continuación

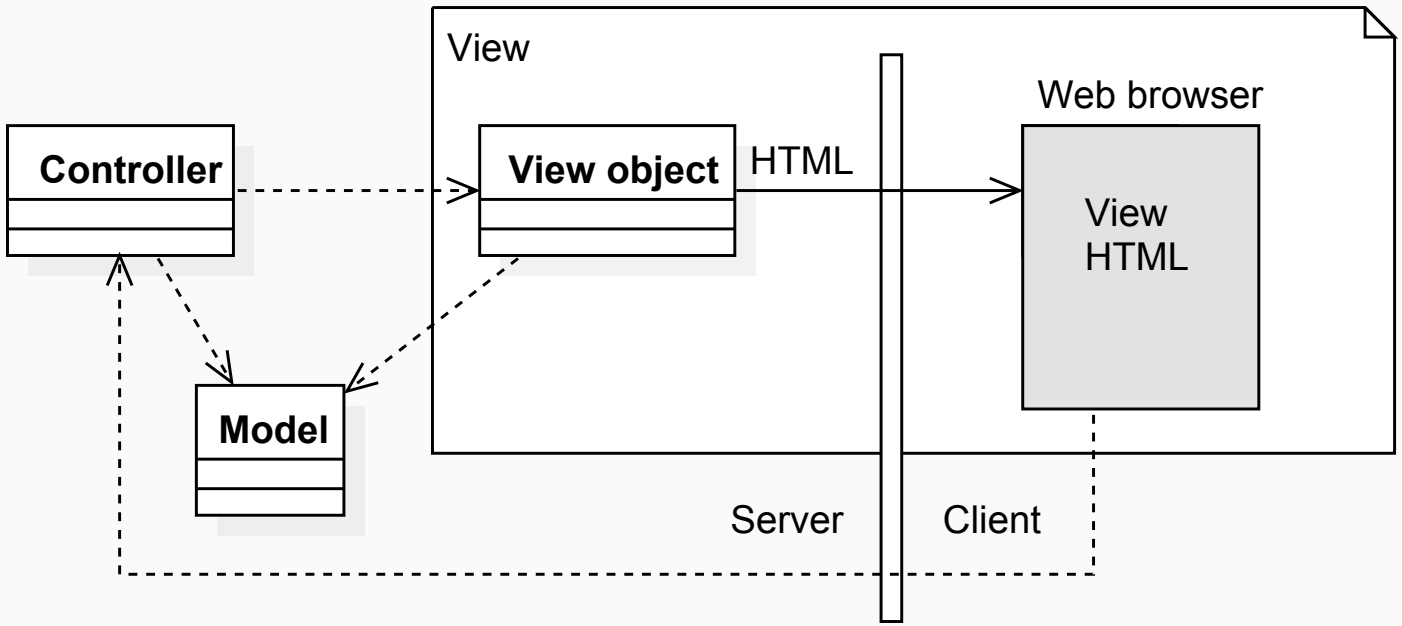


MVC en aplicaciones web

Esta forma de comunicarse es necesaria porque la aplicación está dividida físicamente entre el cliente y el servidor.

La vista tiene dos partes:

- Un objeto (dinámico) que se instancia en la capa de presentación del servidor para generar el HTML
- El HTML (estático) mostrado en el navegador del cliente



Tipos de controladores

Hay varias formas de programar los controladores:

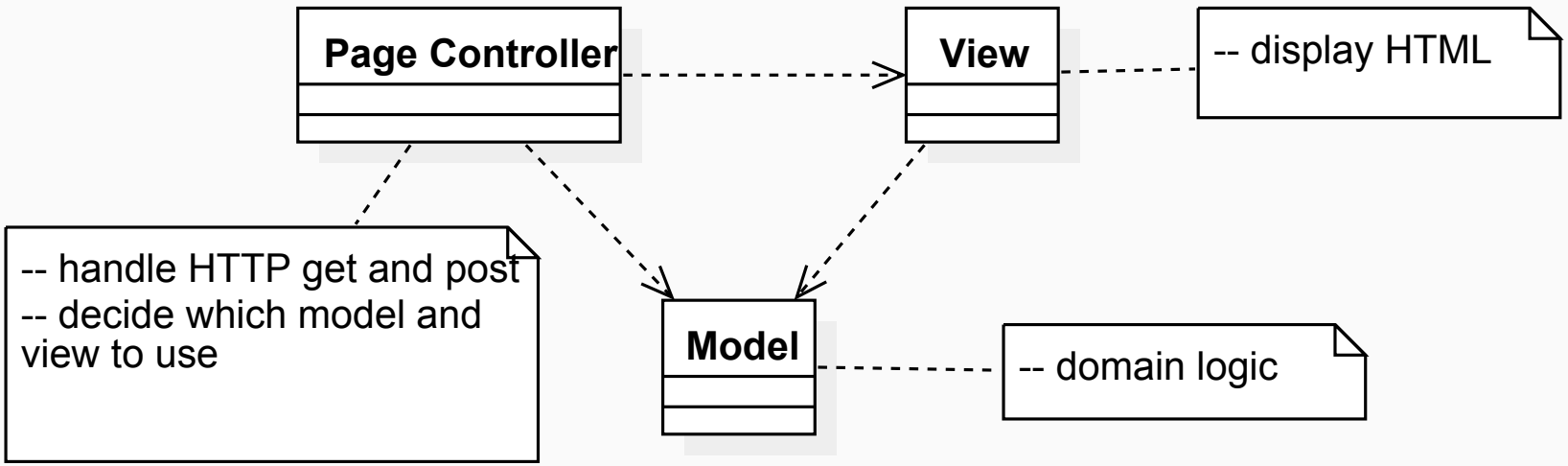
- Page controller
- Front controller

Un **Front Controller** puede colaborar además con un **Application Controller**.

Page Controller

Patrón Page Controller [Fowler, 2003]

Un objeto que gestiona una petición para una página o acción específica en un sitio web.



Page controller

Es el patrón utilizado por ASP.NET (*code-behind*).

Implica la creación de un controlador para cada página lógica de la aplicación.

Responsabilidades:

- Decodificar la URL y extraer los datos de la petición
- Crear e invocar los objetos del modelo
- Seleccionar la vista a mostrar

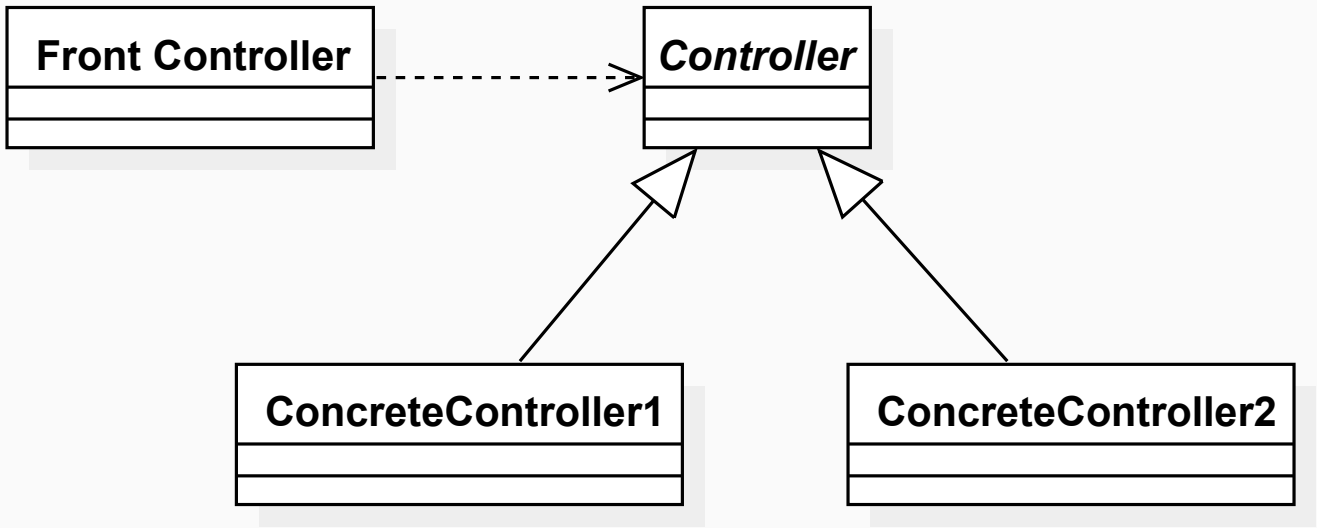
Es manejable cuando la navegación es simple. Para aplicaciones más complejas es muy difícil de gestionar.

Alternativa $\rightarrow$ Patrón *Front Controller*

Front Controller

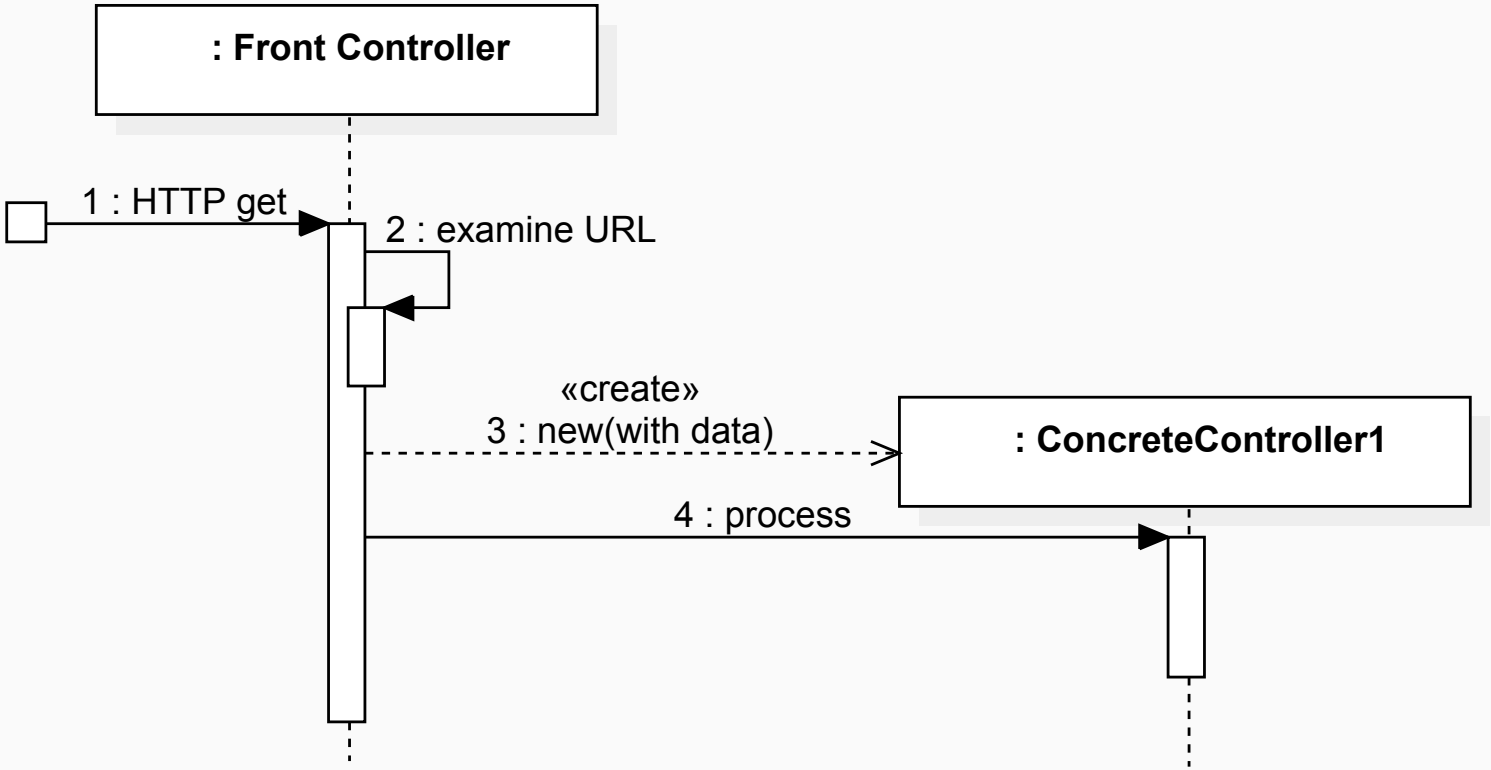
Patrón Front Controller [Fowler, 2003]

Un controlador que gestiona todas las peticiones de un sitio web.



Front Controller

El **Front Controller** realiza el comportamiento común a todas las acciones, y luego delega en controladores específicos para cada acción.



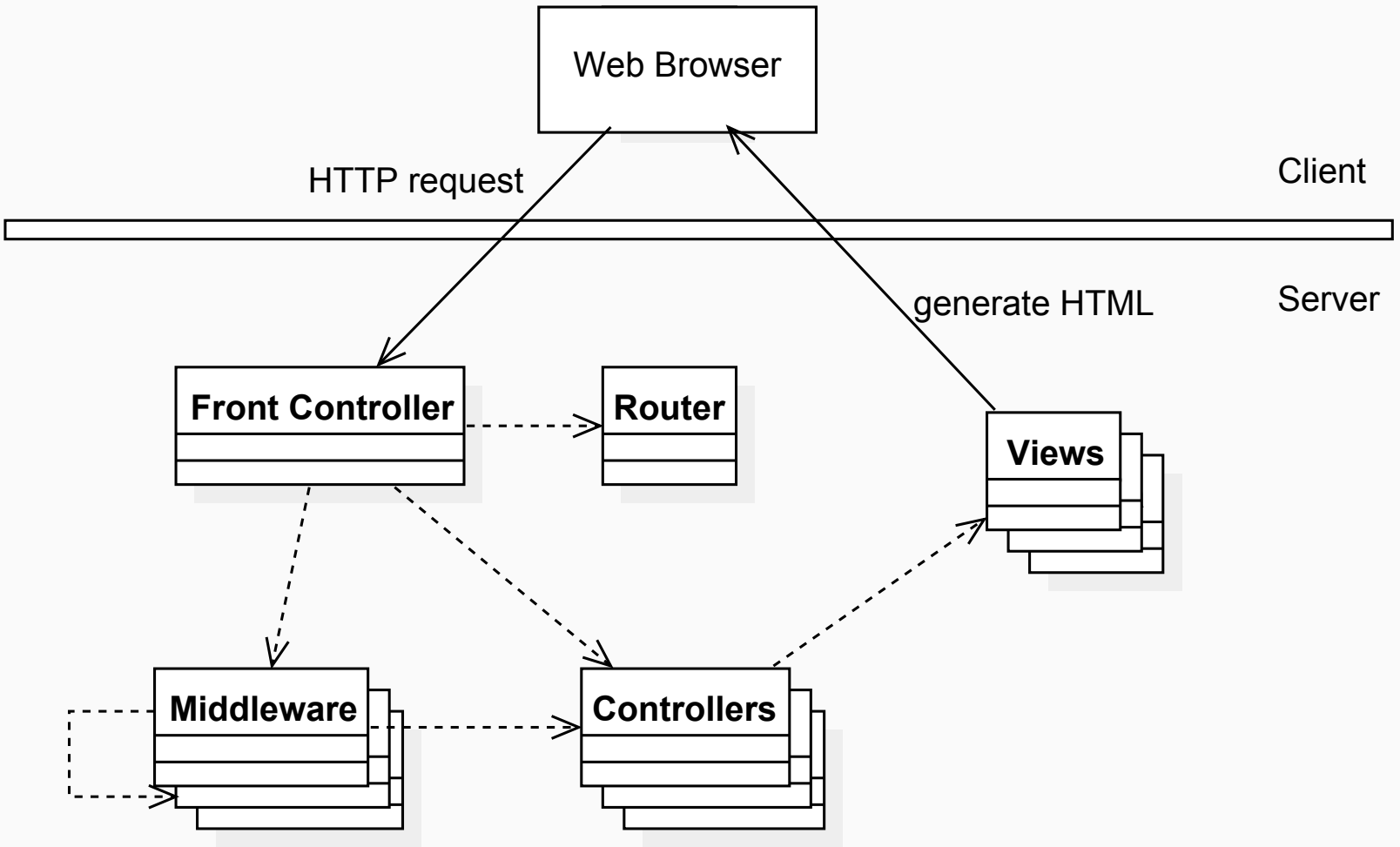
Front Controller

La principal ventaja frente al patrón Page Controller es que ahora un solo objeto controlador gestiona todas las peticiones para evitar duplicación de comportamiento.

En frameworks web modernos, el Front Controller delega algunas de sus responsabilidades en otros objetos:

- Objetos **Middleware** para chequeos de seguridad, internacionalización, etc.
- Un enrutador (**Router**) para seleccionar el controlador específico que corresponde a cada acción
- Si la lógica para gestionar la navegación es compleja se puede usar un **Application Controller**

Front Controller



Estructura de un framework web actual

Middleware

Cada objeto Middleware se especializa en realizar un tipo de comprobación:

- Verificar si el usuario está autentificado
- Comprobar si el usuario tiene permisos para ejecutar la acción
- Limitar el acceso a los recursos por número de accesos o ancho de banda (*throttling*)
- Comprobar y/o modificar los valores de entrada
- ...

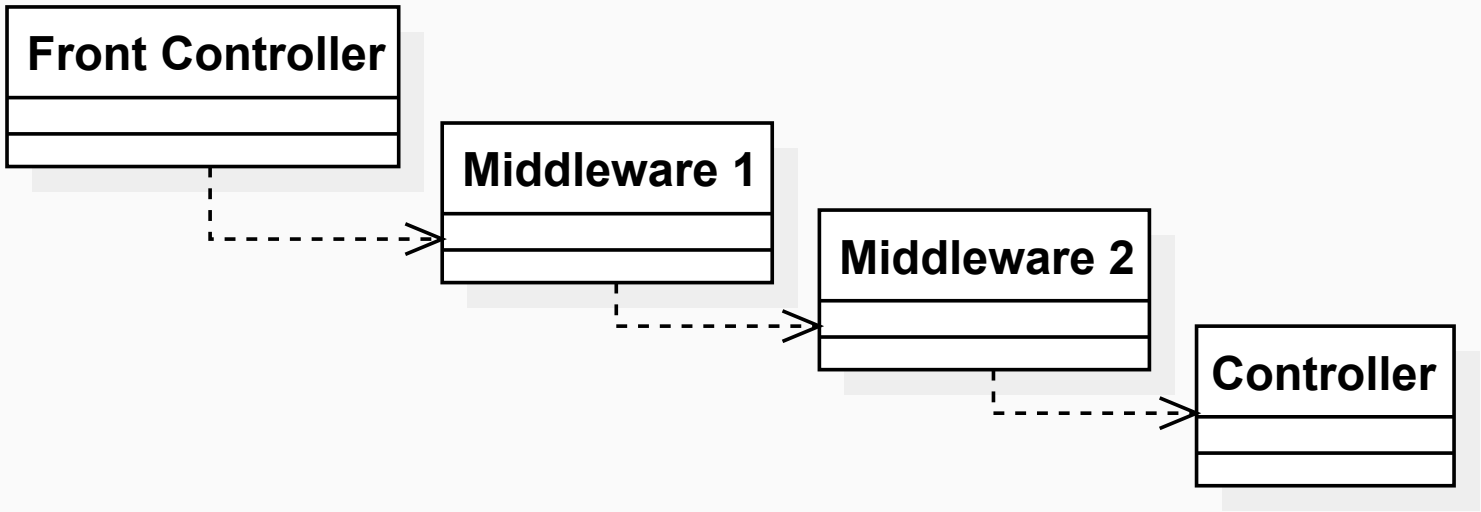
Los frameworks incorporan algunos Middleware por defecto (p.ej. comprobar usuario autentificado).

También se pueden crear objetos Middleware personalizados.

Middleware

Se pueden encadenar distintos objetos Middleware para realizar varias comprobaciones antes de ejecutar el controlador:

- Si las comprobaciones son correctas pasa la ejecución al siguiente Middleware, o finalmente al Controlador
- Si alguna de las condiciones definidas en los Middleware no se cumple se puede detener la ejecución. Por eso se les llama también **Filtros**



Router

El objeto **Router** es el responsable de seleccionar el controlador apropiado para cada petición.

En algunos frameworks como Laravel las rutas se especifican en un archivo de **rutas** que el Router carga al iniciar la aplicación:

```
Route::get('/', 'HomeController@index');
Route::get('post/create', 'PostController@create');
Route::post('post', 'PostController@store');
```

Otros frameworks como ASP.NET MVC están configurados para hacer una correspondencia entre la ruta y el método del controlador a ejecutar. Por ejemplo, la ruta `/Product/Edit/3` se correspondería con:

- Controlador = Product
- Método = Edit
- id = 3

Application Controller

En ocasiones un controlador no puede decidir directamente cuál es la siguiente vista a mostrar, ya que puede depender del estado de los modelos o de la ejecución de una regla de negocio.

Ejemplo

Para una compañía aseguradora, después de modificar los datos de un parte tras la actuación de un especialista podrían pasar dos cosas:

- Si el incidente se ha resuelto mostraría la vista para cerrar el parte
- Si es necesaria la actuación de otro especialista se mostraría la vista correspondiente

Application Controller

En estos casos es conveniente mantener un **Application Controller** en una capa separada, p.ej. en la capa de lógica de dominio.

Si el flujo de trabajo es complejo podría ser útil usar una máquina de estados, representada por algún tipo de metadatos.

Patrón Model-View-Presenter

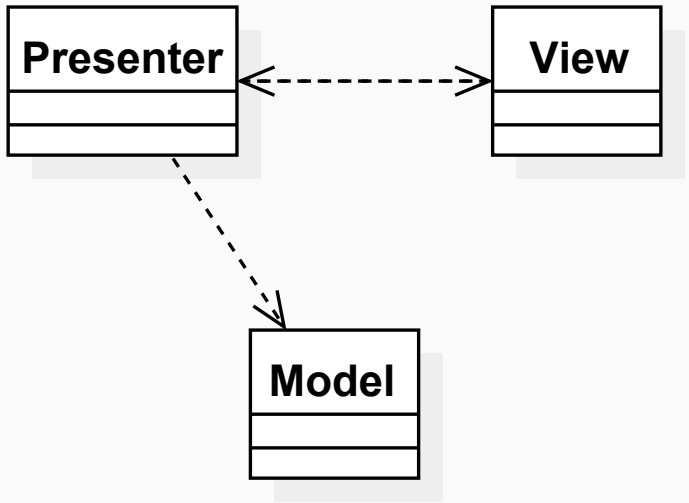
Patrón MVP

El patrón MVC tiene varios inconvenientes:

- La vista está acoplada al modelo, si no tomamos precauciones podemos acabar con vistas que deben hacer un procesamiento complejo para mostrar los datos
- Cada vista tiene su propio controlador, hay poca reutilización de código
- Es difícil probar los controladores porque están acoplados con vistas concretas

Patrón Model-View-Presenter

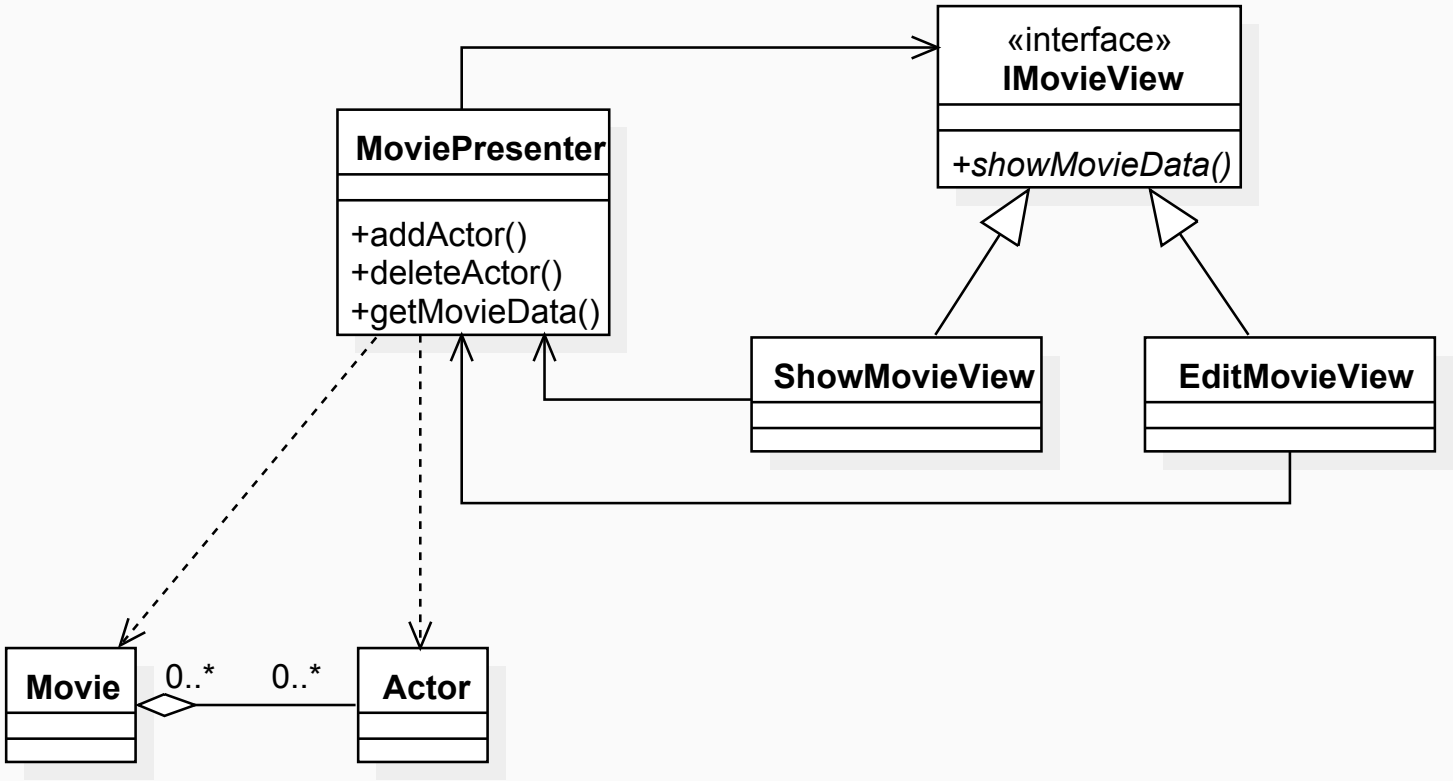
Para solucionar estos problemas, el patrón Model-View-Presenter (MVP) propone el uso de un objeto **Presenter** en lugar del controlador de MVC.



La vista ya no depende del modelo, ahora el objeto Presenter lee el modelo y prepara los datos para pasárselos a la vista.

Patrón Model-View-Presenter

Para facilitar la reutilización de código, los objetos Presenter están desacoplados de las vistas concretas que los usan.



De esta manera también es más fácil probar los Presenter **inyectándoles** vistas mock.

Patrón Model-View-Presenter

Cuando se crea una instancia de la vista, ésta crea a su vez un Presenter para que cargue la información que necesita.

Si la aplicación es distribuida el Presenter realiza la petición en un hilo paralelo, de manera que la vista puede continuar su ejecución sin bloquear el interfaz. Cuando el presenter recibe la información, pasa los datos a la vista para que los muestre.

Cada vez que una vista necesita nueva información se la pide al Presenter siguiendo el mismo procedimiento.

Patrón Model-View-ViewModel

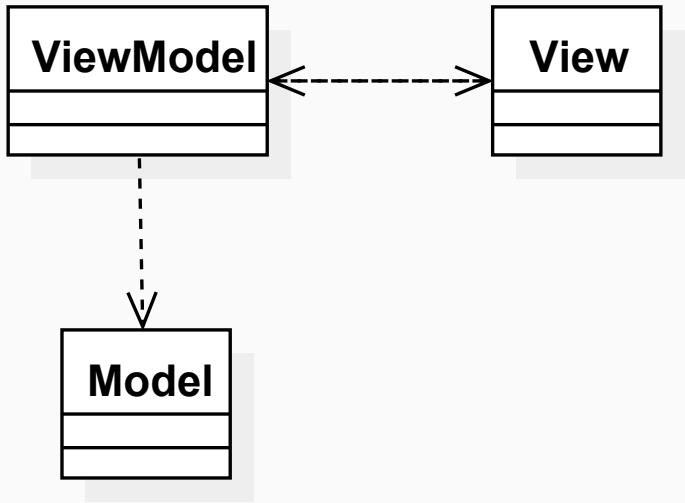
Patrón MVVM

Cuando una aplicación tiene mucha interacción en el interfaz, la complejidad del código puede aumentar demasiado:

- Una sola vista puede necesitar datos de muchos modelos
- Hay que gestionar todos los eventos de los controles gráficos del interfaz y responder adecuadamente a cada acción
- Un cambio en el modelo puede necesitar que se actualicen varias partes de una pantalla

Patrón MVVM

Para este tipo de aplicaciones surgió el patrón Model-View-ViewModel (MVVM).



El objeto **ViewModel** es una representación del modelo en la capa de presentación, que además contiene la lógica necesaria para responder a los eventos del interfaz.

Patrón MVVM

El objeto ViewModel es el responsable de comunicarse con la capa de lógica de negocio cuando se solicita desde la vista.

El verdadero potencial de este patrón está en el uso de **Data Binding**, creando un enlace entre los controles del interfaz y los objetos ViewModel:

- Cuando el usuario actualiza el campo de un formulario, se actualiza automáticamente la propiedad asociada del ViewModel
- Cuando el ViewModel se actualiza como resultado de una llamada a la lógica de negocio, se actualizan automáticamente los controles asociados en el interfaz

Esto permite construir interfaces con mucho menos código.

Aplicaciones web RIA

Aplicaciones web RIA

Rich Internet Application (RIA)

Una Aplicación Rica (o enriquecida) de Internet es una aplicación web que tiene un interfaz con características similares a las aplicaciones de escritorio.

Se pueden usar distintas tecnologías para implementarlas: JavaScript, Java Applets, Flash, Silverlight, ...

Las más extendidas actualmente son las aplicaciones de tipo **Single Page Application (SPA)**, y usan JavaScript y AJAX en el cliente.

Aplicaciones web RIA

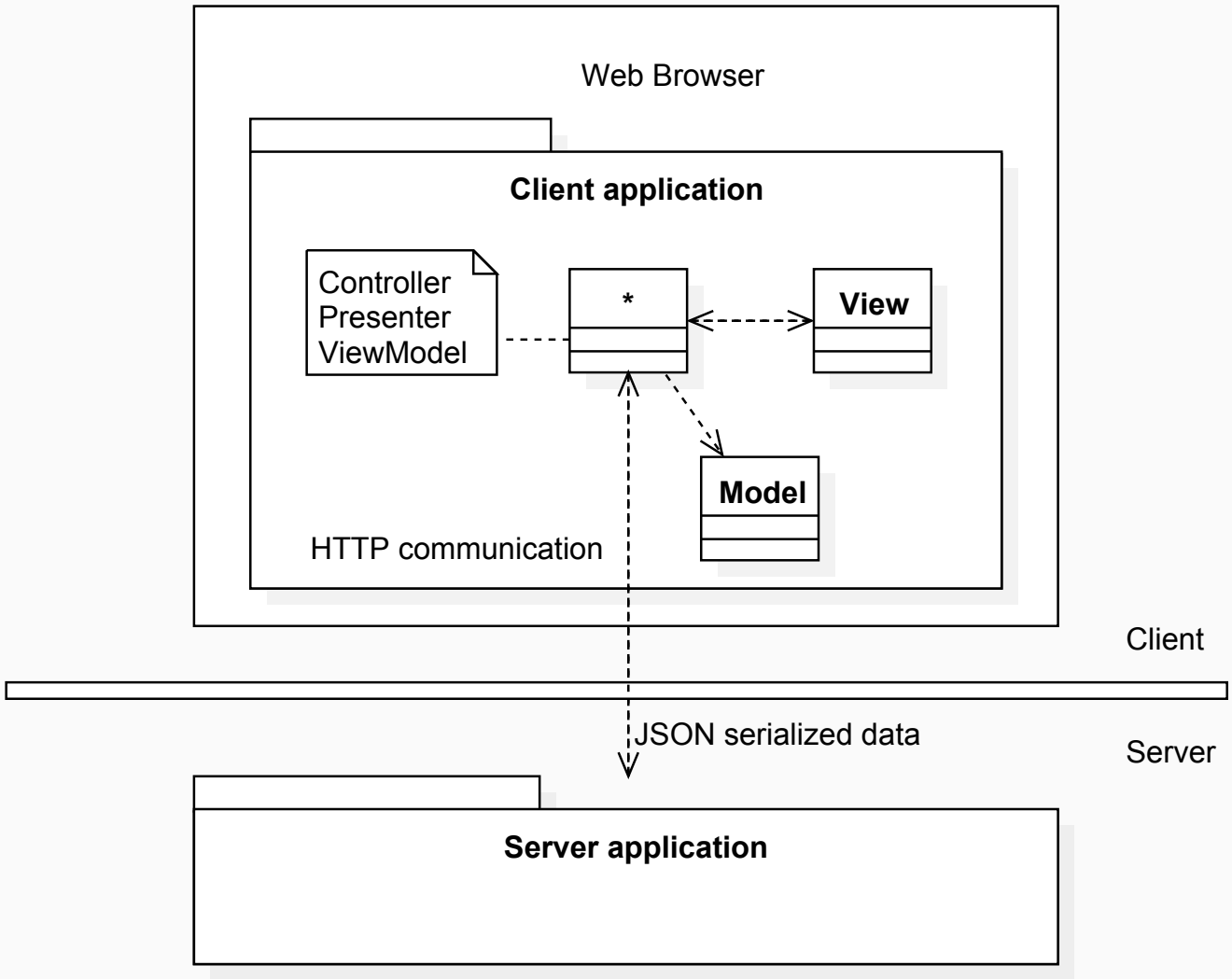
Funcionamiento de una aplicación SPA:

- El cliente carga un HTML mínimo y los scripts con la lógica de la aplicación cliente
- El interfaz se construye dinámicamente, generando el HTML necesario a partir de los datos proporcionados por el servidor
- El cliente realiza peticiones AJAX al servidor, la página no se recarga
- Los datos se transmiten serializados en formato JSON

Independientemente de la arquitectura usada en el servidor, estas **aplicaciones cliente necesitan estructurar su código**.

Para esto existen frameworks que favorecen el uso de los distintos **patrones MV***.

Aplicaciones web RIA



Estructura típica de una aplicación SPA

¿Preguntas?

Referencias i

Fowler, M. (2003).
***Patterns of Enterprise Application Architecture*.**
Addison-Wesley Professional.

Osmani, A. (2015).
***Learning JavaScript Design Patterns*.**
O'Reilly Media.
https://addyosmani.com/resources/essentialjsdesignpatterns/book/.