Diseño de Sistemas Software

PEEA

Capa de Presentación

Carlos Pérez

Santiago Meliá

Introducción

- Cuando se separa una aplicación en distintas capas, la **Capa de Presentación** tiene las siguientes responsabilidades:
  - Mostrar la información
  - Gestionar la interacción con el usuario
  - Comunicarse con las capas inferiores que proveen las funcionalidades deseadas

Introducción

- Los objetos en esta capa (normalmente interfaces de usuario o *vistas*) deben mantenerse lo más independientes posible de la lógica de negocio
  - Es más fácil probar cada capa por separado
  - La interfaz de usuario se puede sustituir por otra tecnología cuando sea necesario
  - Se pueden añadir nuevas interfaces (web, móvil, …) sin que afecte a la lógica de dominio

Introducción

- Los patrones de presentación PEAA estaban enfocados principalmente al desarrollo de aplicaciones web
- Hay nuevas tecnologías (como WPF y XAML para .NET apps) que permiten aplicar estos patrones indistintamente a aplicaciones web, móviles o de escritorio
- Uno de los patrones más usados para estructurar la capa de presentación es **Model View Controller** (MVC)
- Hay otros patrones de presentación alternativos como **Model View Presenter (MVP)** y **Model View ViewModel (MVVM)**

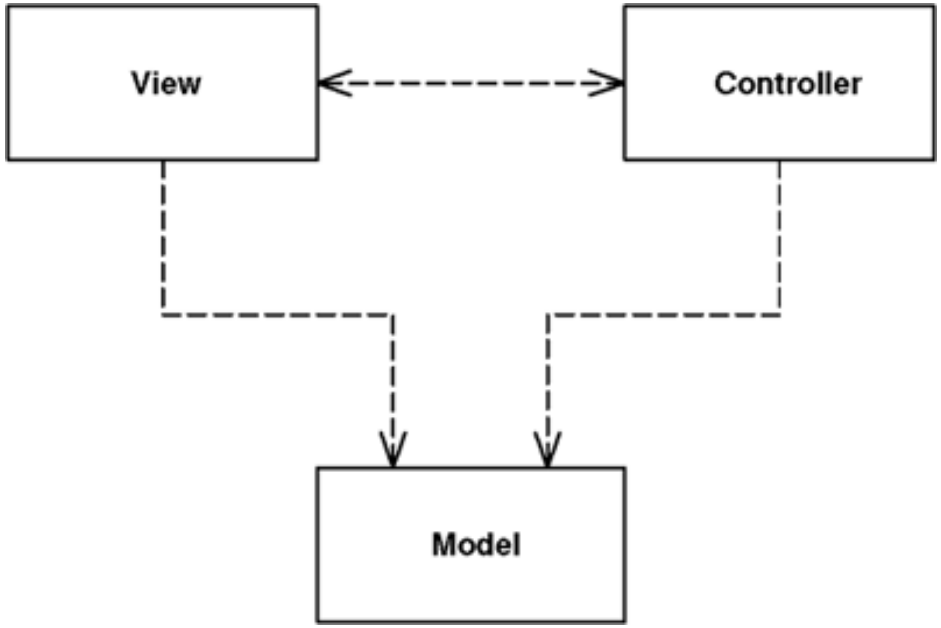
Model View Controller

Capa de Presentación

Model View Controller (MVC)

- Definición:

  “Separa la interacción de la interfaz de usuario en tres roles diferenciados”



Model View Controller (MVC)

- Separa la presentación del modelo
  - Disminuye el acoplamiento
  - Permite mostrar la información del modelo de distintas formas
  - Facilita la automaticación de pruebas de la lógica de negocio
- Separa la vista del controlador
  - El controlador puede decidir qué vista mostrar en cada caso
- Tanto la presentación como el controlador dependen del modelo, pero el modelo no depende de ninguno de los dos

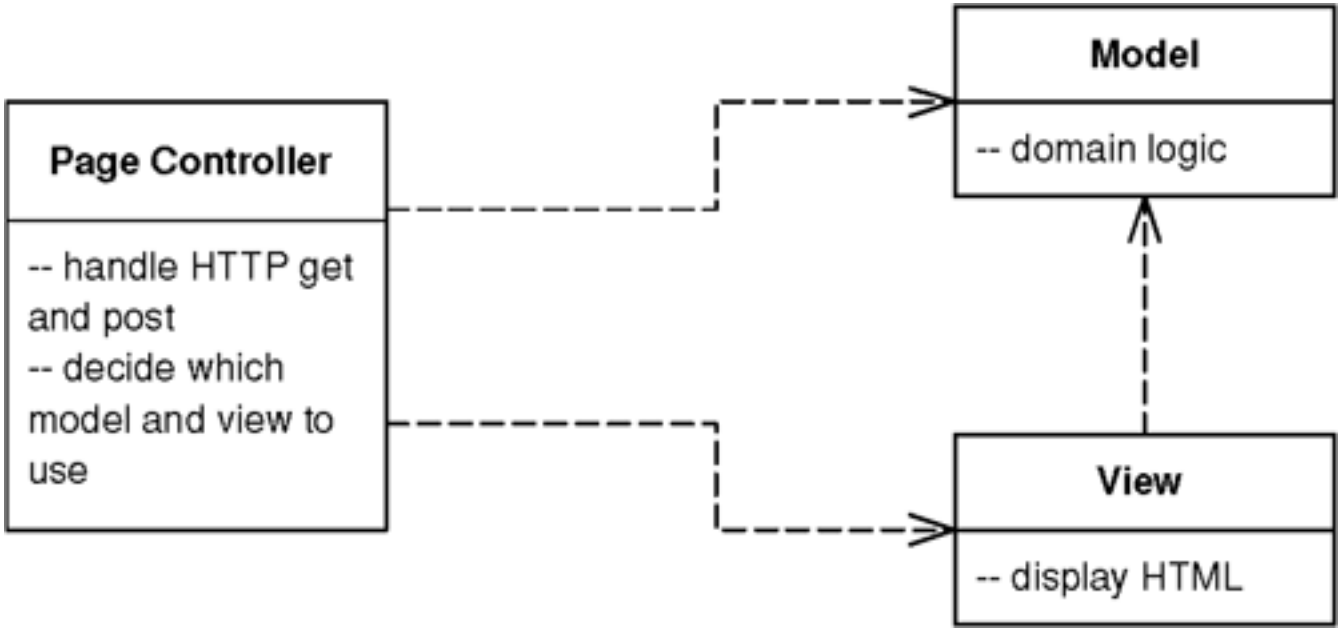
Controller

- El controlador en MVC está pensado para actuar como un **input controller**
  - Procesa las peticiones y las interacciones (ratón y teclado)
  - Pasa la información de la petición al modelo, que la procesa y devuelve los resultados
  - Manipula el resultado para mostrarlo en la vista
- Tipos de *input controller*
  - Page controller
  - Front controller
- *Input controller* puede colaborar con un *Application Controller*

Page Controller

- Definición:

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



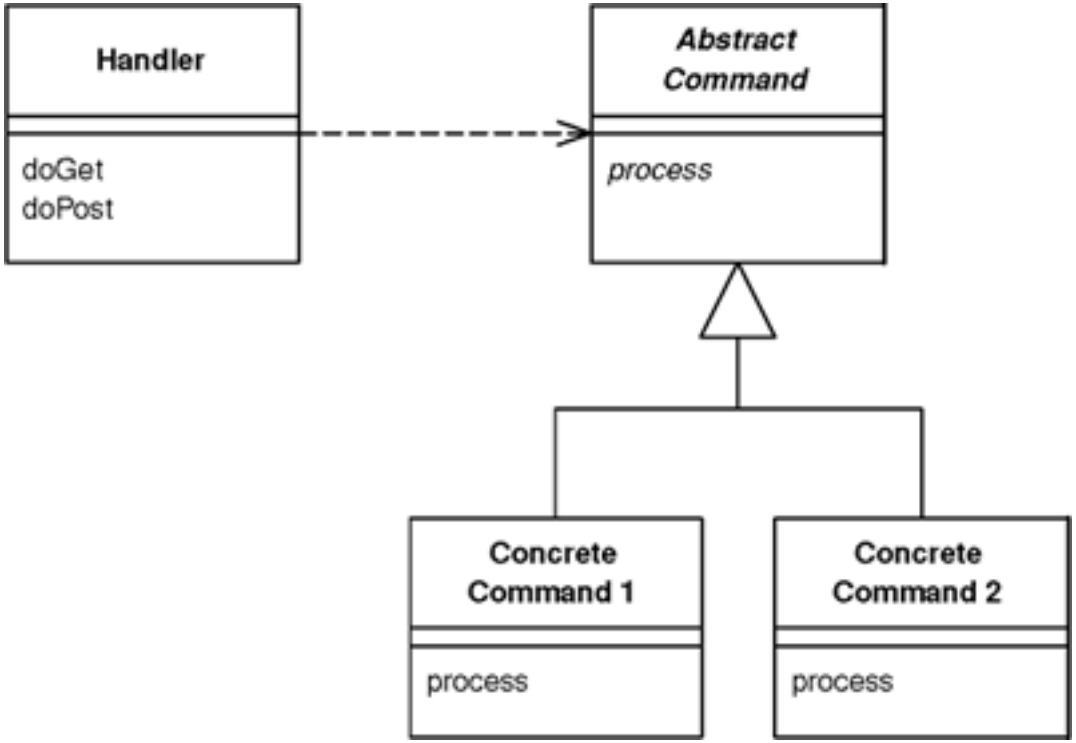
Page Controller

- En las aplicaciones web, el patrón Page Controller 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
- Útil cuando la navegación es simple

Front Controller

- Definición:

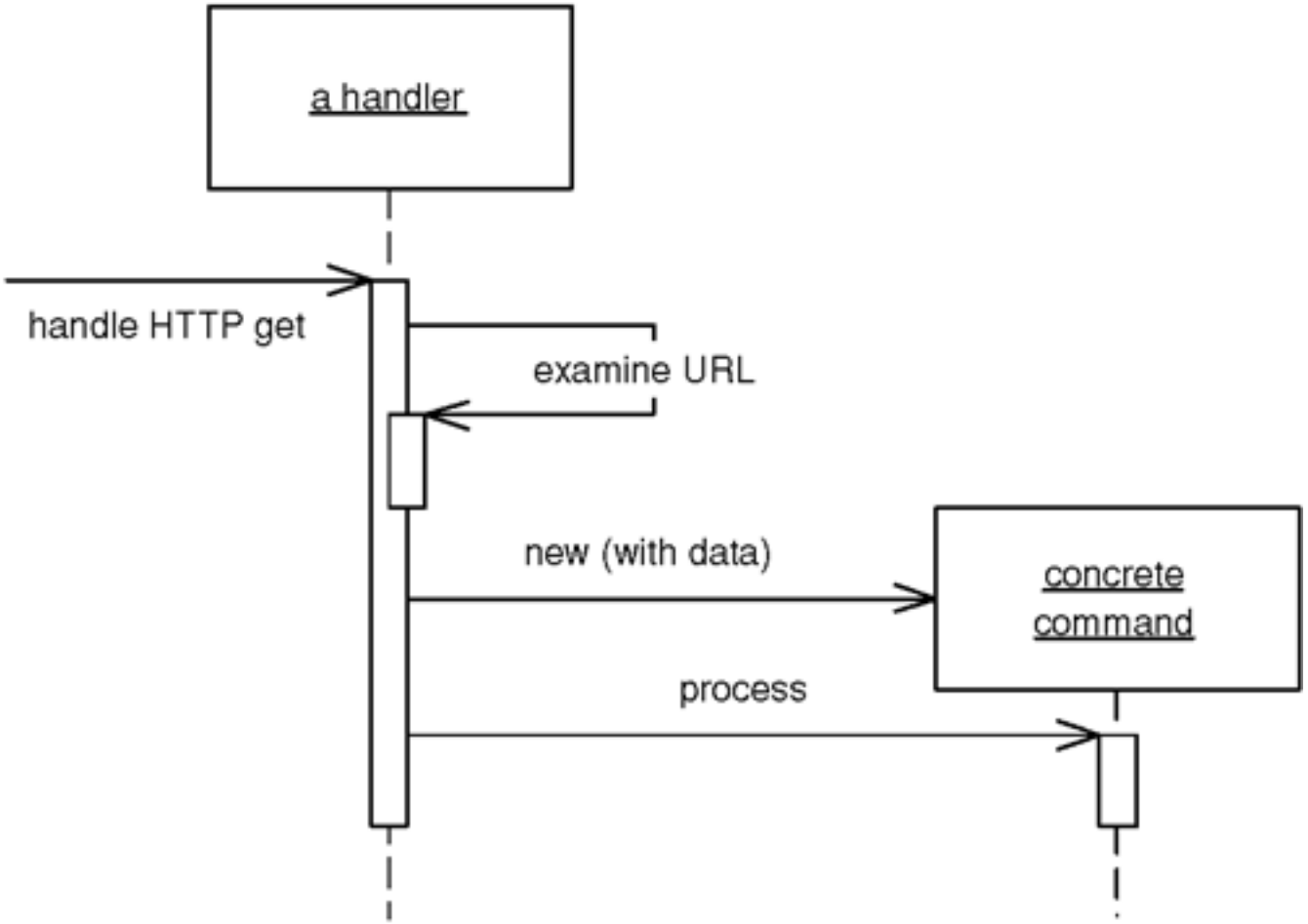
  "Un controlador que gestiona todas las peticiones de un sitio web"



Front Controller

- Un solo objeto controlador gestiona todas las peticiones para evitar duplicación de comportamiento
  - Chequeos de seguridad
  - Internacionalización
  - Personalización basada en el usuario
- El front controller realiza el comportamiento común a todas las acciones, y luego delega en objetos *Command* para el comportamiento particular de cada acción

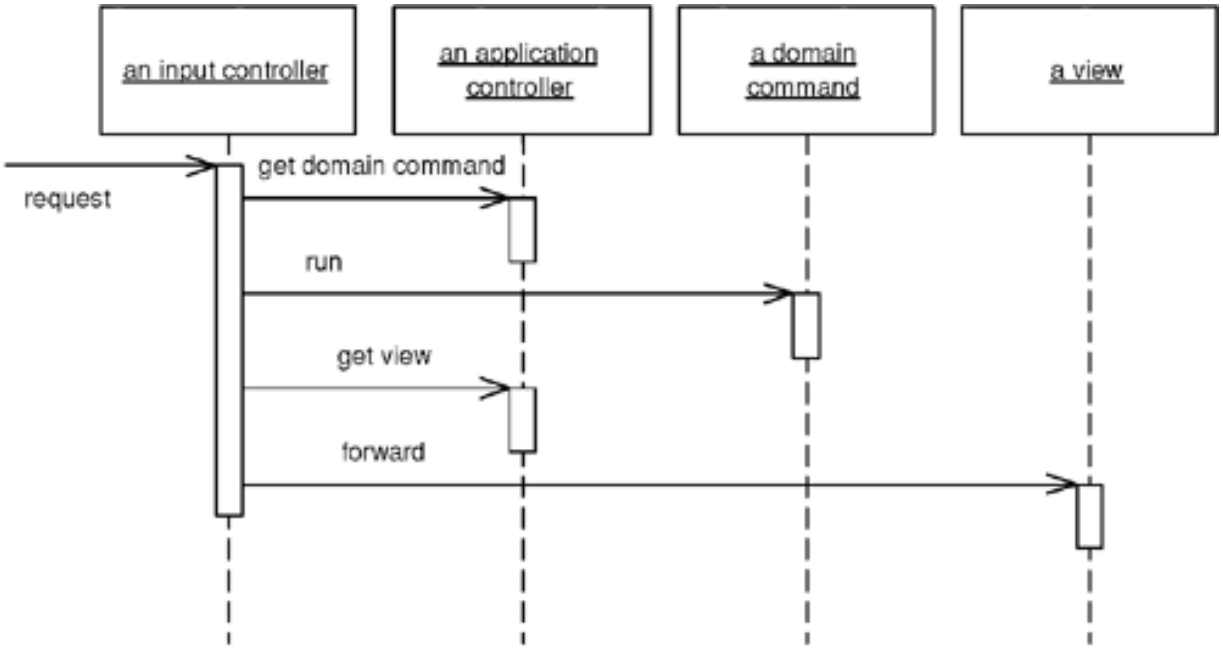
Front Controller



Application Controller

- Definición:

  "Un punto centralizado para gestionar la navegación y el flujo de una aplicación"



Application Controller

- Input Controller puede delegar las siguientes responsabilidades en un Application Controller
  - Decidir qué lógica de dominio ejecutar
  - Decidir la vista con la que mostrar el resultado

- Necesita manterner referencias a las colecciones de vistas y acciones de dominio (patrón Command)

Application Controller

- Es mejor mantenerlo separado del interaz de usuario, en una capa separada
- Podría ser parte de la capa de dominio si el flujo de trabajo es parte de la lógica de dominio
- Si el flujo de trabajo es complejo, podría ser útil usar una máquina de estados, representada por algún tipo de metadatos

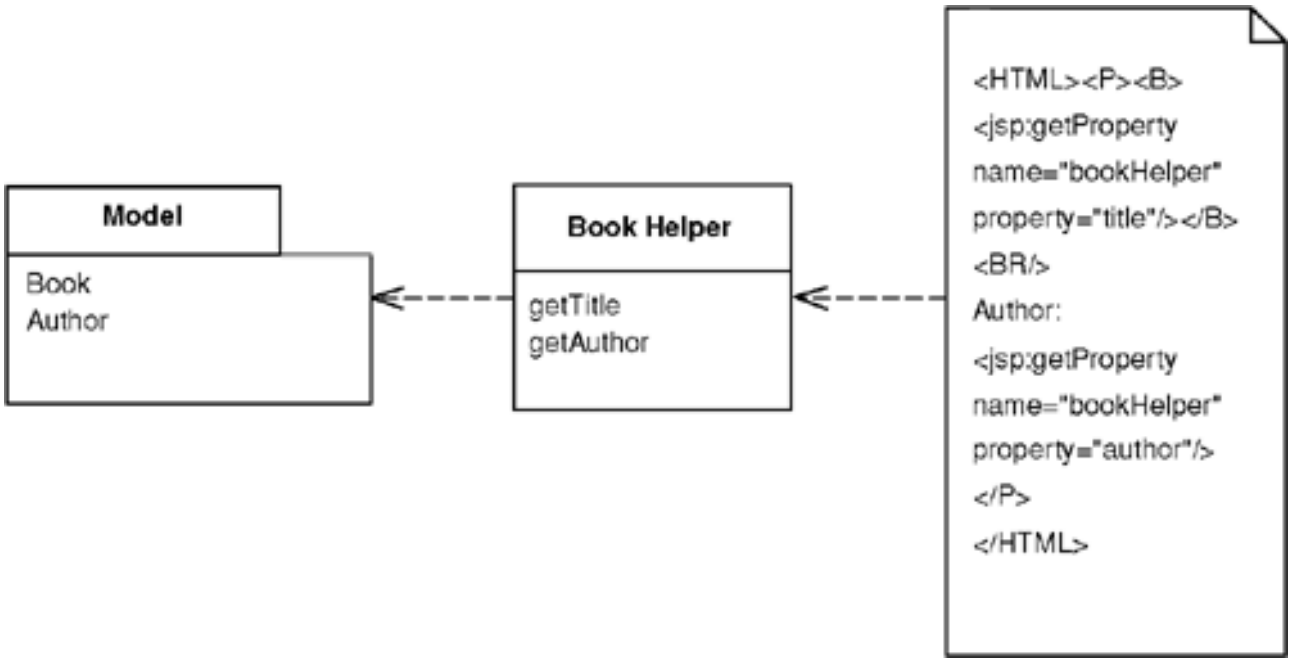
View

- Dos patrones básicos
  - **Transform View**
  - **Template View**
- Si algunas pantallas son muy similares pueden compartir la misma vista

Template View

- Definición:

  “Transforma la información a HTML incrustando marcas en una página HTML”



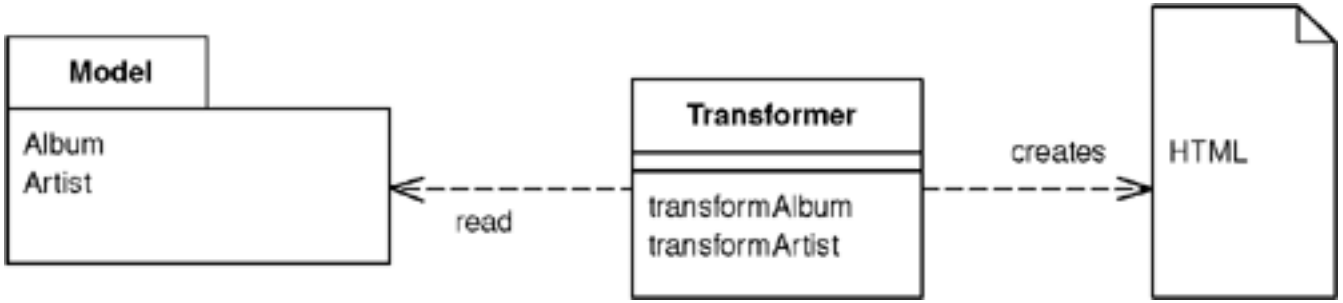
Template View

- El principal objetivo del Template View es evitar escribir todo el código HTML mediante programación en el objeto vista
- Las vistas se escriben como HTML (u otro lenguaje de marcado), con marcas que se reemplazan en tiempo de ejecución por el resultado de algún cálculo
- Ejemplos típicos son **server pages**: ASP, JSP, PHP, …

Transform View

- Definición:

  “Una vista que procesa los datos elemento por elemento y los transforma en HTML”



Transform View

- Los datos recuperados desde la lógica de dominio se transforman y formatean a la salida final usando un conjunto de reglas de transformación
- La elección típica es usar XSLT como lenguaje de transformación
- Los datos del modelo de dominio se pueden recuperar directamente como XML, o como objetos que se serializan automáticamente a XML

Model View Presenter

Capa de Presentación

Model View Presenter (MVP)

- En el MVC el controlador se encarga de gestionar los eventos el usuario, modificar los componentes de interfaz y llamar al modelo en respuesta de dichos eventos
- En el MVC controlador se encuentra muy ligado con las vistas que gestiona, dificultando su reutilización para diferentes vistas
- El controlador del MVC es difícil de mantener al no poderse probar separadamente del interfaz de usuario

Model View Presenter (MVP)

- **Problema:** ¿Cómo proveemos de una separación en el Interfaz de Usuario que facilite la reutilización del controlador, permita realizarle pruebas y mejore la separación IU y Lógica?

Model View Presenter (MVP)

- **Fuerzas:**
  - Deseamos que las vistas puedan ser testeadas mediantes técnicas de automatización de pruebas
  - Deseamos compartir código del controlador entre pantallas (incluso de diferente tecnología) que requieren el mismo comportamiento
  - Aumentar la separación entre el interfaz y la lógica de negocio (View y Model) permite hacer el código más fácil de mantener

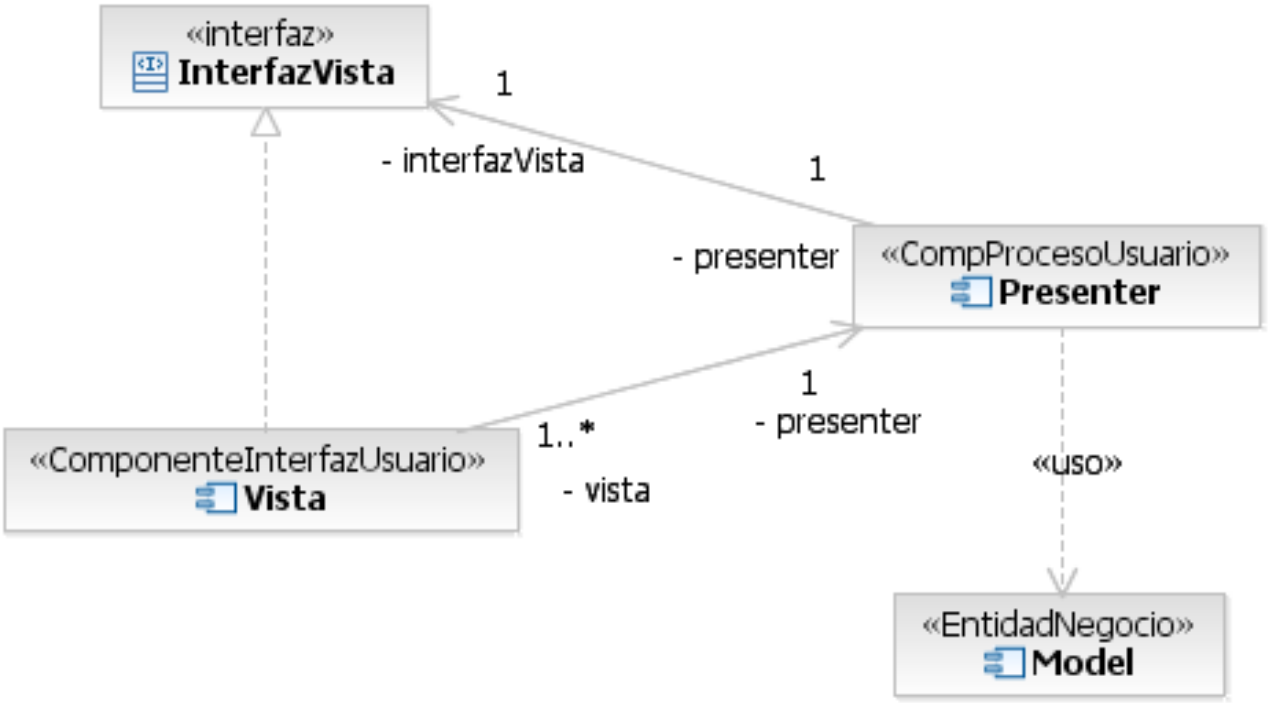
Model View Presenter (MVP)

- **Solución:**
  - Separar las responsabilidades de la visualización del IU y la gestión de eventos en diferentes clases, llamadas respectivamente **View** y **Presenter**
  - La clase **View** gestiona los controles en la página, captura los eventos y redirige las peticiones a la clase Presenter (View sería el XAML y el code-behind)
  - El **presenter** contiene la lógica que responde a las peticiones de la vista (carga y modificación de datos), actualiza el modelo, y a su vez, manipula el estado de la vista

Model View Presenter (MVP)

- **Solución (Cont.):**
  - Para facilitar las pruebas del Presenter, hacemos que éste mantenga una referencia a un interfaz de la Vista que permita cambiar su estado facilitando el remplazo de la vista por otra tecnología, o por mock, permitiendo así realizar tests funcionales (**Patrón Dependency Injection**)
  - El Presenter puede comunicarse con la vista y con el modelo, pero la vista es completamente independiente del modelo (a diferencia del patrón MVC)

Model View Presenter (MVP)



Model View ViewModel

Capa de Presentación

Model View ViewModel (MVVM)

- **Contexto**:
  - Debemos elegir la arquitectura de Interfaz de Usuario de una aplicación con interfaz rico que contiene mucha interacción entre el interfaz de usuario y los datos de la aplicación
  - La vista recupera los datos y maneja los eventos de usuario, que pueden modificar otros controles en respuesta a ellos o enviar los cambios para actualizar los datos

Model View ViewModel (MVVM)

- **Contexto**:
  - Incluir el código que realiza todas las funciones en la vista hace que se complique en exceso
  - Sin embargo, debemos mantener el alto nivel de interacción que requiere dicha interfaz
  - Se hace muy difícil compartir código entre las vistas que requieren el mismo comportamiento, dificultando su mantenimiento y sus pruebas

Model View ViewModel (MVVM)

- **Problema**:
  - ¿Se puede hacer una arquitectura de presentación para las interfaces ricas que mejore el mantenimiento y las pruebas?

Model View ViewModel (MVVM)

- **Fuerzas**:
  - Se quiere maximizar el código que puede ser probado automáticamente
  - Se quiere compartir código entre las vistas que tienen el mismo comportamiento
  - Se desea separar la lógica de negocio de la lógica de presentación

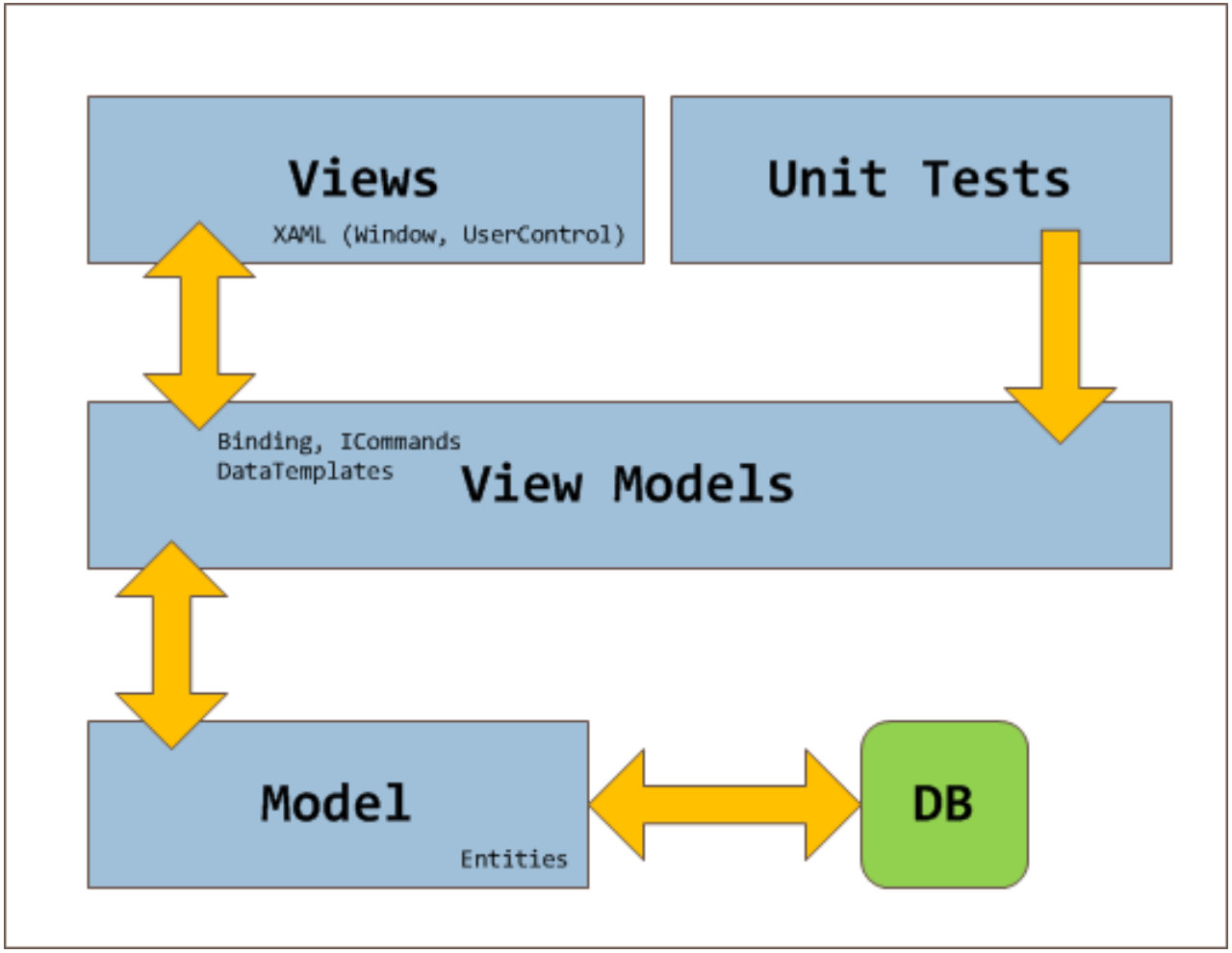
Model View ViewModel (MVVM)

- **Solución**: Patrón *Presentation Model* [Fowler, 2004] o *Model View Viewmodel* (MVVM) (Gossman, 2005)
- Propone una separación de responsabilidades en la interfaz de usuario:
  - La representación visual en el componente **View**
  - El estado del interfaz y el comportamiento en el componente **Viewmodel**
  - La lógica de negocio en un componente **Model** (que notifique de los cambios a la UI)

Model View ViewModel (MVVM)

- La View contiene los controles en la interfaz de usuario y el Viewmodel actua como una fachada del modelo con el estado y el comportamiento de la IU
- El Viewmodel encapsula el acceso al modelo proveyendo una interfaz publica que es fácil de consumir desde la vista (p.ej. usando data binding)

Model View ViewModel (MVVM)



https://sureshkumarveluswamy.wordpress.com/tag/mvvm/

Bibligrafía

- Fowler, Martin

  Patterns of Enterprise Application Architecture

  Addison Wesley, 2003.