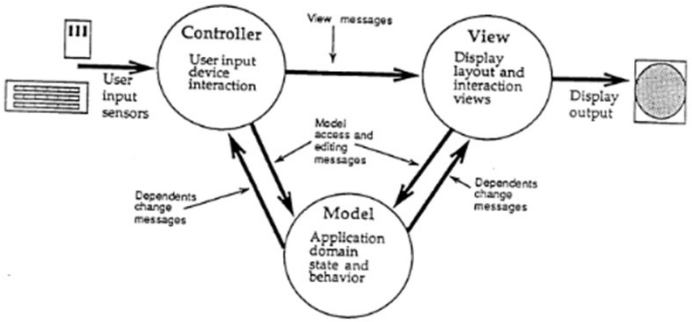
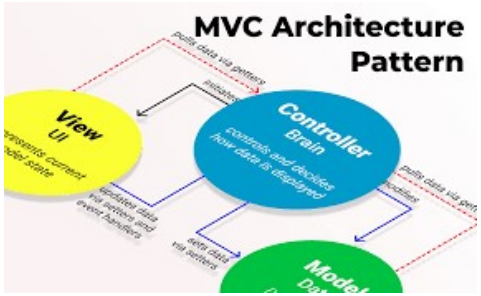
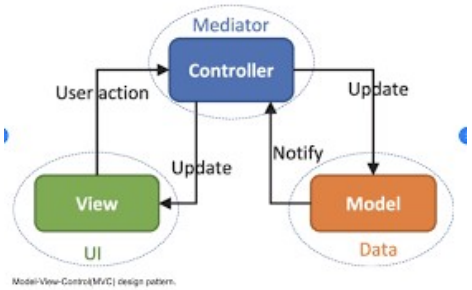
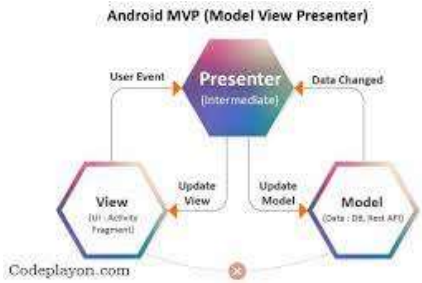


Patrones de diseño de interfaz

MVC, MVC Web, MVVM, MVP







Objetivo de una arquitectura/patrón de UI

- Guiar el desarrollo de la Interfaz (dado que lo hacemos una y otra vez)
- Separar lógica del dominio de la lógica de presentación
- Producir código de UI mas fácil de entender y mantener
- Simplificar el testing
- Fomentar la producción/evolución de librerías/frameworks de UI
- …

Flow synchronization

- Las actualizaciones de vistas/ventanas se disparan en respuesta ciertos eventos en el flujo de la aplicación
  - Se cierra una ventana de Dialogo, y se actualiza la ventana principal
  - Se avanza /retrocede en un wizard y se actualiza la nueva ventana
  - Se responde a un request HTTP y como consecuencia se actualiza la pagina
- PROS:
  - Fácil de implementar y entender para interfaces simples (wizards, root-child)
  - Las órdenes de actualización están junto a los manejadores de eventos.
- CONS:
  - Se vuelve difícil tan pronto la UI se vuelve mas compleja (produce acoplamiento entre ventanas)
  - No adecuado cuando hay que reaccionar a eventos del modelo (o de otras vistas/sistemas)

Observer synchronization

- La interfaz
  - Envía mensaje a modelo en respuesta a eventos de interfaz
  - Se subscribe a notificaciones de cambios en el modelo de dominio
  - Se actualiza (selectivamente) en respuesta a notificaciones del modelo
- El modelo de dominio
  - Implementa un mecanismo de subscripción y notificación de cambios
  - Notifica cada vez que algo cambia
- PROS
  - Escala a interfaces complejas y múltiples vistas sobre un mismo modelo
  - Permite reaccionar a eventos originados en el modelo o en otros sistemas
- CONS
  - La relación entre eventos y actualizaciones de UI se vuelve implícita
  - Requiere atención al ciclo de vida de las subscripciones

Model – View – Controller

El controlador reacciona a eventos de entrada

Controller
<<Observer.Observer>>

La vista se encarga de “pintar” datos del modelo

View
<<Observer.Observer>>

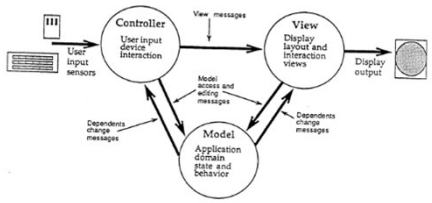
El controlador observa al modelo y le envía mensaje en respuesta a acciones del usuario

La vista observa al modelo y se actualiza, enviando mensajes al modelo, cuando este cambia

Model
<<Observer.Subject>>

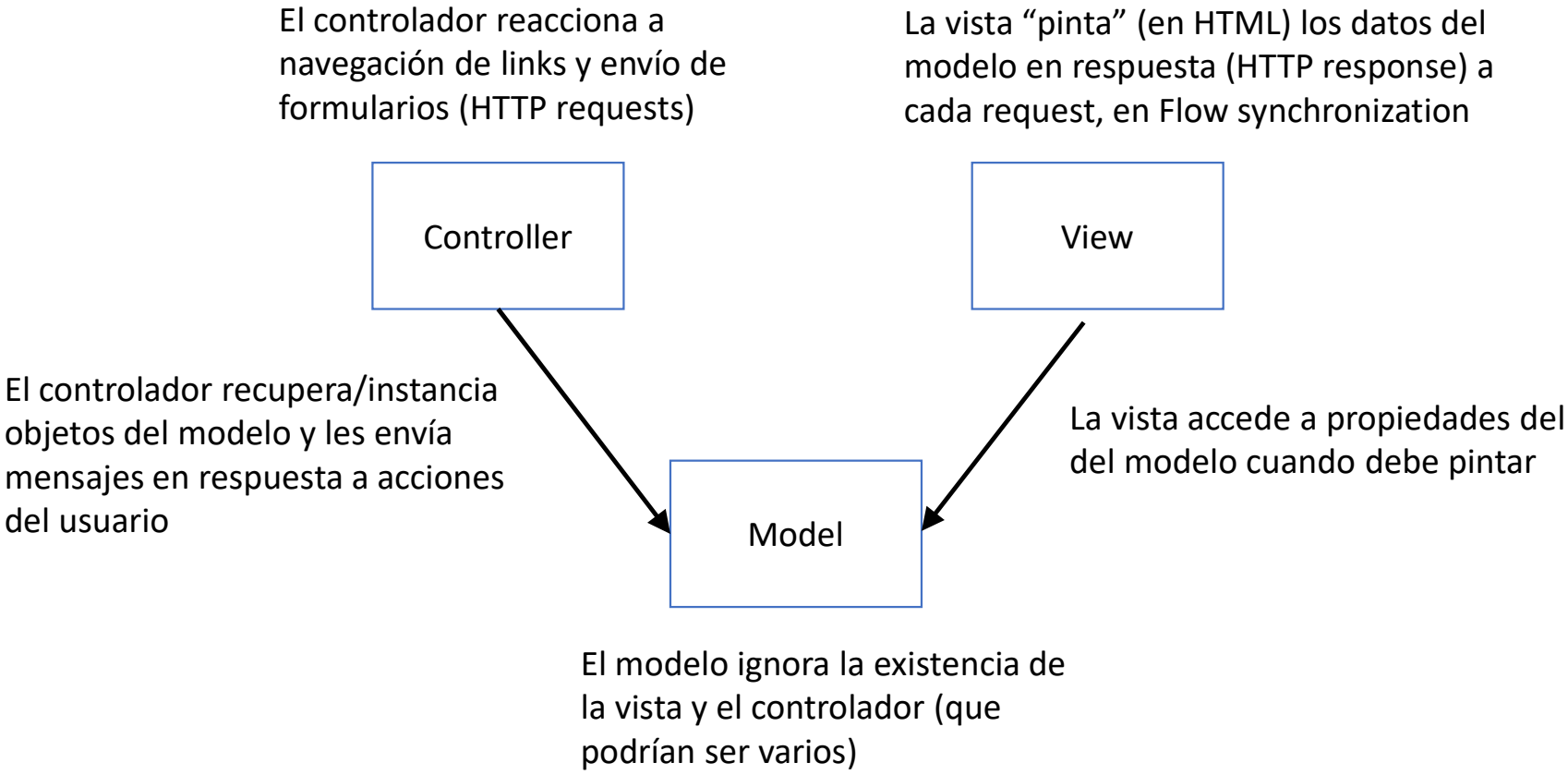
El modelo ignora la existencia de la vista y el controlador (que podrían ser varios)

Model – View – Controller



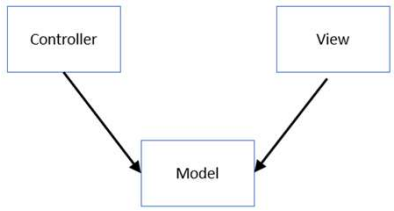
- Buena separación de concerns (cada objeto hace una sola cosa)
  - Aunque en aplicaciones de escritorio, es frecuente que se termine programando vista y controlador en un solo objeto
- Se subclasifica directamente View/Controller solo cuando requiero vistas totalmente nuevas
- Se subclasifica o configura (Pluggable views) una vista existente (lista, botón, etc.) para especializarla
- Se componen vistas (Patrón Composite). La misma estructura se repite desde el widget a la ventana
- Se le critica que mucha dependencia entre la vista y el modelo dificulta el testing
- No hay un lugar claro para el estado y comportamiento de “la aplicación”

(Server-side) Model – View – Controller (Web)

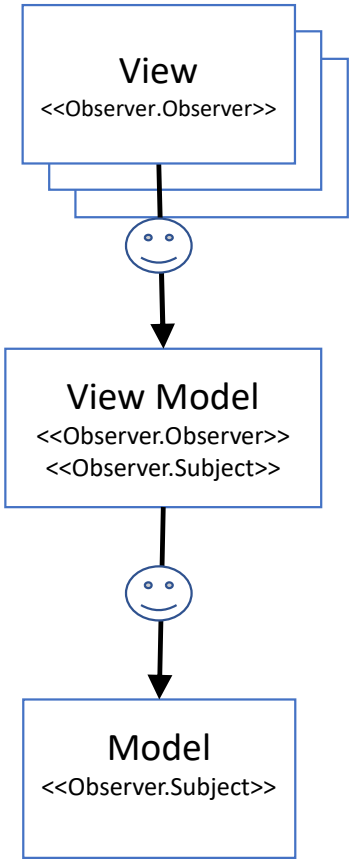


(Server-side) Model – View – Controller (Web)

- Model hacer referencia tanto a modelo de dominio como a modelo de la aplicación (por ejemplo estado de paginación, objetos específicos de interfaz)
- El controlador interactúa con el modelo del dominio, e instancia objetos de modelo según lo requerido para pintar las vistas
- El controlador determina con que vista se responde
- Buena separación de concerns que lo volvieron muy popular
- La granularidad de las View es (generalmente) una página
- No aplica exclusivamente a OO



Model – View – View Model



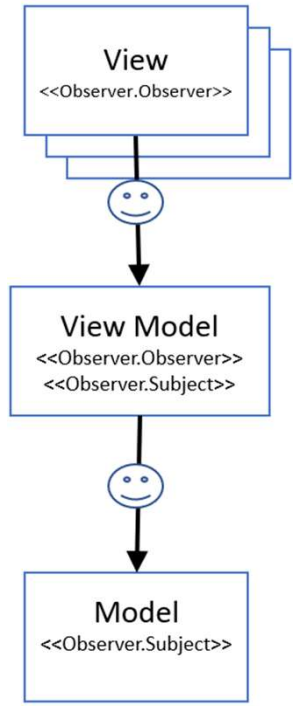
View combina aspectos que antes se separaban en view y controller

Mantiene estado de la/s vista/s
Conecta las views a partir estados compartidos
Implementa las reacciones específicas a los eventos de interfaz
Hace de intermediario entre views y modelo de dominio

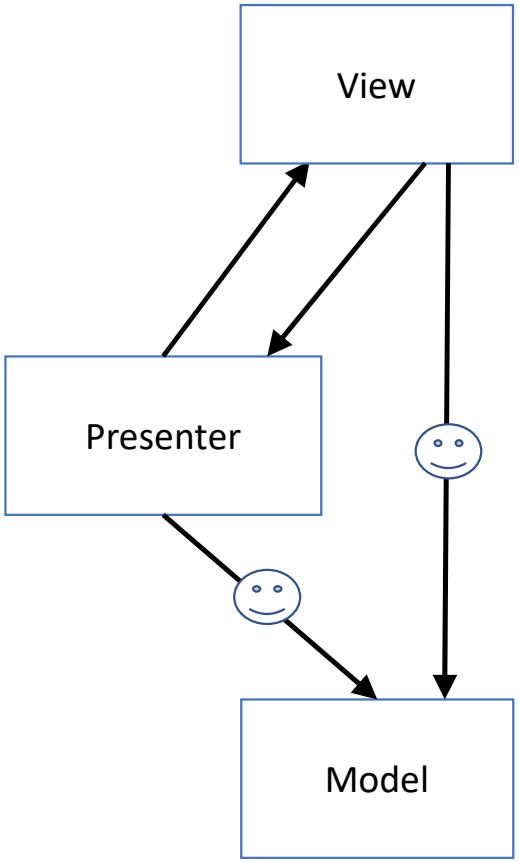
El modelo ignora la existencia de View y del View Model

Model – View – View Model

- Agrega un nivel mas de separación de concerns (el View model)
- Sigue dependiendo fuertemente del observer
  - En .Net (y otros frameworks modernos) se oculta por medio de mecanismos propios (data binding declarativo)



Model – View – Presenter



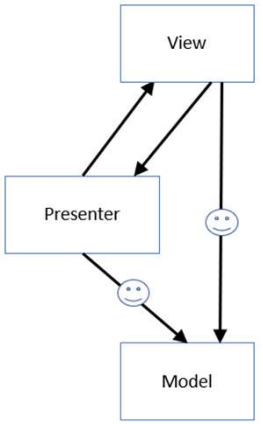
Combina lo que antes era view y controller
Delega los eventos en el presenter
En algunas implementaciones, View observa a Model

Implementa manejadores para los eventos de las views
Puede sincronizar vista y modelo
Puede modificar a la vista mediante envíos de mensajes
Concentra la lógica de interfaz

El modelo (de dominio) ignora la existencia de View y Presenter

Model – View – Presenter

- Es mas flexible respecto a sincronización vista modelo
  - Puede imlementarse por medio de Flow synchronization
  - La vista puede sincronizarse via observer o por acción del presenter
- No es tan estricto en términos de separación de concerns y reducir el acomplamiento
- Android (la parte de construcción de UI) y Swing (Java) siguen la filosofía MVP



MVx – Algunas conclusiones

- El modelo nunca conoce la existencia de las vistas, controladores, presentadores, etc.
- El modelo (o plataforma) debe ofrecer algún mecanismo de sincronización tipo Observer (salvo Flow Synchronization)
- Vista y controlador suelen estar separados en la web, juntos en aplicaciones de escritorio
- La granularidad de lo que llamamos “vista” y “controlador” varía, desde un widget hasta una página web
- La “lección” es “separar concerns” para facilitar comprensión, desarrollo y testing.