Comparación de Arquitecturas de Software

Antonio Sánchez Ramírez y Juan Jesús Suárez Miranda

Parte 1: Estudio Comparativo:

1. Describir la estructura básica de cada arquitectura.

- MVC: propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador; es decir: por un lado, define componentes para la representación de la información y, por otro lado, para la interacción del usuario.

- MVP: patrón arquitectónico de interfaz de usuario diseñada para facilitar pruebas de unidad automatizada y mejorar la separación de inquietudes en lógica de presentación, el presentador se encargará de formatear los datos para que la vista los muestre.

- MVVM: Extremadamente parecido a MVC la diferencia radica en que el Controlador actuará como un intermediario directo esperando las interacciones del usuario para cambiar al modelo y la vista, mientras que MVVM utiliza a ViewModel que permitirá ser notificado por Model, ViewModel con esta notificación actualizará automáticamente a la Vista.

2. Definir los roles y responsabilidades de cada componente (Modelo, Vista, Controlador/Presentador/ViewModel).

- Modelo: representación de la información con la que el sistema opera, por lo que gestiona todos los accesos a dicha información, consultas y actualizaciones, implementando los privilegios de acceso descritos en las especificaciones de la aplicación (lógica de negocio).

- Vista: Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario), por tanto, requiere de dicho 'modelo' la información que debe representar como salida.

- Controlador: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos)

- Presentador: Actúa como mediador entre el Modelo y la Vista, orquestando el flujo de datos y manejando las interacciones del usuario. Es responsable de recuperar datos del Modelo, procesarlos y actualizar la Vista en consecuencia.

- ViewModel: Actor intermediario entre el modelo y la vista, contiene toda la lógica de presentación y se comporta como una abstracción de la interfaz. La comunicación entre la vista y el modelo de vista se realiza por medio los enlaces de datos.

3. Comparar el nivel de acoplamiento y cohesión entre los componentes

1. MVC (Modelo-Vista-Controlador)

Acoplamiento: En MVC, el acoplamiento puede ser considerado moderado. La Vista y el Controlador están más acoplados, ya que la Vista necesita conocer al Controlador para enviarle eventos. Sin embargo, el Modelo está relativamente desacoplado de la Vista y el Controlador, lo que permite que se pueda cambiar sin afectar a los otros componentes.

Cohesión: La cohesión en MVC es bastante buena. Cada componente tiene una responsabilidad clara: el Modelo maneja la lógica de negocio y los datos, la Vista se encarga de la presentación y el Controlador actúa como intermediario. Esto permite que cada parte se enfoque en su tarea específica.

2. MVP (Modelo-Vista-Presentador)

Acoplamiento: En MVP, el acoplamiento es más bajo en comparación con MVC. La Vista se comunica con el Presentador, pero no tiene conocimiento directo del Modelo. Esto permite que la Vista y el Modelo sean más independientes entre sí, lo que facilita la prueba y el mantenimiento.

Cohesión: La cohesión en MVP es alta. El Presentador tiene la responsabilidad de manejar la lógica de presentación y de interactuar con el Modelo, mientras que la Vista se centra únicamente en la interfaz de usuario. Esto hace que cada componente tenga un propósito claro y definido.

3. MVVM (Modelo-Vista-ViewModel)

Acoplamiento: MVVM tiende a tener el menor acoplamiento de las tres arquitecturas. La Vista se enlaza directamente con el ViewModel a través de data binding, lo que significa que no necesita conocer los detalles del Modelo. Esto permite una gran flexibilidad y facilita la prueba de los componentes.

Cohesión: La cohesión en MVVM también es alta. El ViewModel actúa como un intermediario que maneja la lógica de presentación y expone los datos de una manera que la Vista puede consumir fácilmente. Esto permite que cada componente mantenga su enfoque y responsabilidad.

4. Evaluar la facilidad de realizar pruebas unitarias en cada arquitectura.

MVC:

En MVC, el Modelo es generalmente fácil de probar, ya que contiene la lógica de negocio. Sin embargo, las pruebas del Controlador pueden ser un poco más complicadas, ya que a menudo dependen de la Vista. Esto puede hacer que las pruebas unitarias sean menos efectivas, especialmente si la Vista está muy acoplada al Controlador. Además, las interacciones entre la Vista y el Controlador pueden requerir pruebas más integradas en lugar de pruebas unitarias puras.

MVP:

MVP es muy amigable para las pruebas unitarias. El Presentador puede ser probado de manera aislada, ya que no tiene dependencia directa de la Vista, solo interactúa a través de una interfaz. Esto permite simular la Vista en las pruebas, lo que facilita la verificación de la lógica de presentación sin necesidad de una interfaz gráfica. El Modelo también se puede probar de forma independiente, lo que mejora aún más la capacidad de prueba.

MVVM:

MVVM es quizás la más fácil de probar de las tres arquitecturas. El ViewModel puede ser probado de manera aislada, ya que no tiene dependencia directa de la Vista. Gracias al data binding, la Vista se actualiza automáticamente con los cambios en el ViewModel, lo que significa que puedes centrarte en probar la lógica del ViewModel sin preocuparte por la implementación de la Vista. Además, el Modelo también se puede probar de forma independiente, lo que proporciona una gran flexibilidad.

5. Analizar la curva de aprendizaje y complejidad.

1. MVC

Curva de Aprendizaje:

- Baja a Moderada: Es uno de los patrones más utilizados, lo que significa que hay mucha documentación y ejemplos disponibles. Los desarrolladores con experiencia en la programación de aplicaciones web suelen estar familiarizados con este patrón.

Complejidad:

- Moderada: La separación de responsabilidades puede complicar el flujo de datos, especialmente en aplicaciones más grandes. Sin embargo, su simplicidad en la estructura básica lo hace accesible.

2. MVP

Curva de Aprendizaje:

- Moderada: Al igual que MVC, MVP es un patrón bien documentado, pero la idea de un "presentador" que actúa como intermediario puede requerir un cambio en la forma de pensar sobre la arquitectura de la aplicación.

Complejidad:

- Moderada a Alta: La interacción entre el modelo, la vista y el presentador puede agregar una capa adicional de complejidad. Es esencial que el presentador maneje correctamente la lógica de presentación y la interacción del usuario, lo que puede ser un reto en aplicaciones más complejas.

3.MVVM

Curva de Aprendizaje:

- Moderada a Alta: MVVM puede ser más complicado de entender al principio, especialmente para quienes no están familiarizados con conceptos como el data binding y la programación reactiva. Es común en el desarrollo de aplicaciones con frameworks como WPF o Xamarin.

Complejidad:

- Alta: MVVM permite una separación muy clara de preocupaciones, pero también introduce conceptos avanzados como el data binding y la gestión del estado. Esto puede llevar a una mayor complejidad en la implementación, especialmente en aplicaciones más grandes.

6. Comparar la escalabilidad de las arquitecturas.

1. MVC

- Escalabilidad: Moderada.
- Ventajas: Fácil de agregar nuevas vistas y modelos.
- Desventajas: Controladores pueden volverse complejos, dificultando la gestión en aplicaciones grandes.

2. MVP

- Escalabilidad: Moderada a Alta.
- Ventajas: Buena separación de la lógica de presentación permite reutilizar presentadores.
- Desventajas: Mayor cantidad de código puede complicar la gestión en aplicaciones grandes.

3. MVVM

- Escalabilidad: Alta.
- Ventajas: Excelente para aplicaciones complejas; el data binding facilita actualizaciones de la UI.
- Desventajas: Más difícil de entender al principio, especialmente para desarrolladores menos experimentados.

7. Presentar ejemplos de casos de uso y aplicaciones prácticas donde cada arquitectura sería más adecuada.

1. MVC

Casos de Uso:

- Aplicaciones web simples: Ideal para aplicaciones con una interfaz de usuario básica, como blogs o sitios de noticias.
- Proyectos pequeños o medianos: Perfecto para proyectos donde la rapidez de desarrollo es prioritaria.

Aplicaciones Prácticas:

- Frameworks: Ruby on Rails, Django (en parte).
- Ejemplo: Un sistema de gestión de contenido (CMS) que requiere mostrar y editar entradas de manera sencilla.

2. MVP

Casos de Uso:

- Aplicaciones con lógica de presentación compleja: Útil para aplicaciones donde la interacción del usuario requiere un manejo más detallado de la lógica.
- Aplicaciones móviles: Ideal para aplicaciones en Android, donde la separación clara de la lógica y la vista es crucial.

Aplicaciones Prácticas:

- Desarrollo de apps móviles: Aplicaciones como gestores de tareas o calculadoras.

- Ejemplo: Una app de gestión de tareas que necesita validar y presentar datos de manera dinámica, como listas de tareas pendientes.

3. MVVM

Casos de Uso:

- Aplicaciones ricas en UI: Adecuado para aplicaciones que requieren una interfaz de usuario interactiva y con cambios frecuentes.

- Desarrollo con frameworks que soportan data binding: Ideal para entornos como WPF, Xamarin o Angular.

Aplicaciones Prácticas:

- Desarrollo de aplicaciones de escritorio: Aplicaciones como software de diseño gráfico o herramientas de análisis de datos.

- Ejemplo: Una aplicación de administración de ventas que necesita mostrar datos en tiempo real y permitir la interacción del usuario de manera fluida.

Parte 2: Análisis de un Escenario Práctico:

En este escenario para una aplicación de red social, se va a utilizar la arquitectura de MVVM, ya que, en una aplicación de red social presenta claras ventajas en comparación con otros patrones como MVC y MVP, especialmente cuando se busca mejorar la experiencia de usuario y la mantenibilidad del código.

Las ventajas que hemos visto respecto a las otras arquitecturas son las siguientes:

1. Separación más clara de responsabilidades: la lógica de negocio y de presentación se maneja en el ViewModel, permitiendo que la Vista se enfoque únicamente en mostrar los datos y reaccionar a eventos de UI

2. Actualizaciones automáticas de la interfaz de usuario: en una aplicación de red social, es el uso de data binding (vinculación de datos), permite que cualquier cambio en el modelo o el ViewModel se refleje automáticamente en la Vista, sin necesidad de intervenciones manuales.

3. Facilidad de escalabilidad y mantenibilidad: Dado que el ViewModel no tiene referencia directa a la Vista, el código es mucho más modular, facilitando la adición de nuevas funcionalidades.

4. Mejoras en la testabilidad: Al no depender de la interfaz gráfica, el ViewModel es mucho más fácil de probar que el Presentador en MVP o el Controlador en MVC, que a menudo están directamente acoplados a la Vista.

5. Optimización de la experiencia de usuario: Con MVVM, estas actualizaciones son más automáticas gracias al uso de técnicas como observadores o reactive programming, donde la interfaz de usuario puede reaccionar de forma inmediata a cualquier cambio en los datos.

6. Soporte para múltiples plataformas: El ViewModel puede ser compartido entre diferentes implementaciones de la Vista, lo que reduce la duplicación de esfuerzos y aumenta la consistencia entre plataformas.

1. ¿Cómo el MVVM maneja las interacciones del usuario con la interfaz?

En una aplicación de red social, **MVVM** maneja las interacciones del usuario de manera fluida a través del **ViewModel** y el uso de **data binding**. Cuando un usuario interactúa con la interfaz (por ejemplo, al dar "me gusta" a una publicación o enviar un comentario), la Vista

captura esa acción, pero en lugar de gestionar directamente la lógica de negocio, delega el manejo de la interacción al **ViewModel**. El ViewModel actualiza los datos pertinentes (el "me gusta" o comentario), que a su vez notifican al Modelo y envían la actualización al servidor.

El **ViewModel** refleja automáticamente cualquier cambio en la interfaz, como actualizar el contador de "me gusta" o mostrar el nuevo comentario en la lista, sin que la Vista tenga que manejar la lógica de presentación directamente. Esto crea una separación clara entre la lógica de la interfaz y la lógica de negocio, lo que permite que las interacciones del usuario se gestionen de manera más eficiente y escalable.

2. ¿Cómo se facilita el mantenimiento del código al usar MVVM?

MVVM mejora significativamente la **mantenibilidad del código** gracias a la separación de responsabilidades. Al desacoplar la Vista del ViewModel, los desarrolladores pueden trabajar en diferentes aspectos de la aplicación sin afectar otras partes. El código de la **Vista** (que gestiona el diseño y la interacción visual) y el **ViewModel** (que gestiona la lógica de presentación) se mantienen aislados, lo que permite que los cambios en la interfaz no afecten directamente a la lógica subyacente y viceversa. Esto es esencial en una red social, donde la interfaz puede evolucionar con nuevas funciones o cambios en el diseño sin necesidad de reestructurar la lógica de negocio.

Además, el uso de técnicas como **data binding** elimina el código repetitivo necesario para actualizar manualmente la Vista cada vez que cambia el Modelo, reduciendo la probabilidad de errores y facilitando la extensión del código con nuevas características.

3. ¿Consideraciones de rendimiento de la arquitectura?

Aunque **MVVM** facilita el mantenimiento y la escalabilidad, también se deben tener en cuenta algunas **consideraciones de rendimiento** en una aplicación de red social. El uso extensivo de **data binding** puede introducir una sobrecarga en el rendimiento si no se maneja adecuadamente, especialmente cuando se trabaja con grandes volúmenes de datos en tiempo real, como un feed de noticias que se actualiza constantemente. Los

enlaces de datos dinámicos pueden generar múltiples actualizaciones de la interfaz simultáneamente, lo que podría afectar la fluidez de la aplicación.

Para mitigar estos posibles problemas de rendimiento, es crucial optimizar las actualizaciones de la interfaz y evitar vínculos de datos innecesarios. El uso eficiente de **observables** y técnicas como la **paginación** de datos puede ayudar a mantener una experiencia rápida y reactiva. También es importante considerar la carga de procesamiento que el ViewModel puede llevar si se maneja toda la lógica de actualización de la interfaz y la sincronización con el servidor.

Parte 3: Reflexión Personal:

Nuestra opinión para proyectos móviles a gran escala es que MVVM es la arquitectura más adecuada. Su flexibilidad y capacidad para gestionar las actualizaciones automáticas de la interfaz la hacen ideal para aplicaciones que necesitan ser rápidas y responsivas. Además, MVVM facilita la escalabilidad, lo que es crucial cuando las aplicaciones crecen en complejidad. Otro punto fuerte es su soporte para la prueba y mantenimiento del código, permitiendo un desarrollo más limpio y organizado. Aunque MVC o MVP pueden ser útiles en proyectos más pequeños, creo que MVVM realmente sobresale cuando se trata de aplicaciones móviles complejas que requieren modularidad, un buen rendimiento y una experiencia de usuario impecable.