ULPGC · UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Práctica 8

Recomendaciones de arquitectura

Programación de Aplicaciones Móviles Nativas
12 de Noviembre de 2023

Autores:

Ana del Carmen Santana Ojeda (ana.santana152@alu.ulpgc.es)

Alejandro David Arzola Saavedra (alejandro.arzola101@alu.ulpgc.es)

Índice

1. Introducción

La arquitectura de una aplicación móvil desempeña un papel fundamental en su éxito, impactando directamente en su **escalabilidad, mantenibilidad y eficiencia**.

Para **la elección de las distintas arquitecturas recomendadas**, se ha optado por seleccionar aquellas que tienen la **prioridad "Muy recomendada"**.

Este documento **explora las recomendaciones de arquitectura** proporcionadas por la documentación oficial de Android para desarrolladores[1].

Seleccionaremos y analizaremos las **cinco recomendaciones** que consideramos más cruciales, **evaluando su aplicabilidad** en el proyecto de la asignatura.

2. Selección de arquitecturas

2.1. Arquitectura 1: Capa de datos

La capa de datos **expone los datos de la aplicación y contiene la lógica empresarial**. Esta separación ayuda a mantener un **código más limpio y facilita la modificación y expansión de la aplicación**.

Facilita el mantenimiento y la escalabilidad al tener una **capa dedicada para la gestión de datos**, mejorando la claridad y la organización del código.

Beneficios

- **Organización del Código**: La separación de la lógica de datos mejora la **estructura general del código**.
- **Mantenibilidad**: Facilita la **identificación y corrección de problemas** relacionados con los datos.
- **Escalabilidad**: Permite agregar fácilmente **nuevas fuentes de datos o realizar cambios en la lógica empresarial sin afectar otras partes de la aplicación**.
- **Claridad**: Proporciona una **visión clara** de dónde se encuentra y cómo funciona **la lógica de datos** en la aplicación.

Impacto en el Rendimiento y la Experiencia del Usuario

- **Rendimiento**: Al modularizar la lógica de datos, se **facilita la optimización de operaciones específicas**, lo que puede contribuir a un mejor rendimiento.
- **Experiencia del Usuario**: Una gestión eficiente de los datos puede mejorar la **velocidad de carga y respuesta de la aplicación**, impactando positivamente la experiencia del usuario.

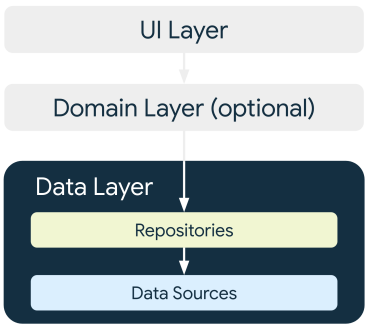

Figura 1: Capa de datos

[1] https://developer.android.com/topic/architecture/recommendations?hl=es-419

2.2. Arquitectura 2: Control de dependencias

La inserción de dependencias es fundamental para la creación de **código limpio y mantenible**. Al pasar las dependencias a través del constructor, se **mejora la claridad del código**, se facilita la prueba unitaria y se **reduce el acoplamiento entre componentes**. Esto es esencial para la **creación de un sistema flexible y fácil de mantener**.

Esta recomendación es fundamental en el desarrollo moderno de software y es **crucial en arquitecturas modulares y basadas en componentes**. Mejora la **flexibilidad y la evolución del sistema a lo largo del tiempo**.

Beneficios

- **Legibilidad del Código**: Hace que el código sea más **fácil de entender** al indicar claramente las dependencias requeridas.
- **Mantenibilidad**: **Facilita la modificación y actualización de dependencias** sin afectar otras partes del código.
- **Pruebas Unitarias**: Permite la **fácil sustitución de dependencias** durante las pruebas unitarias, mejorando la robustez del código.

Impacto en el Rendimiento o la Experiencia del Usuario

- **Rendimiento**: El impacto en el rendimiento es mínimo, ya que **la inserción de dependencias no agrega una carga significativa a la ejecución del programa**.
- **Experiencia del Usuario**: **No hay impacto directo en la experiencia del usuario**, pero **contribuye a la creación de un código más estable y fácil de mantener**, lo cual puede tener **beneficios indirectos en la experiencia del usuario**.

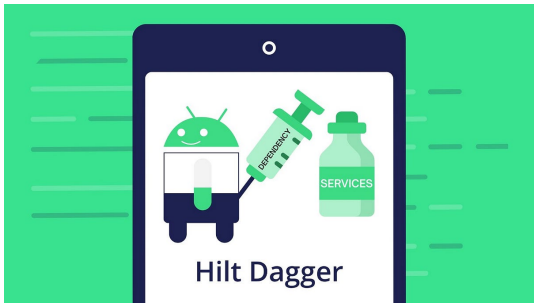

Figura 2: Control de depedencias

2.3. Arquitectura 3: No anular los métodos de ciclo de vida de Activity ni Fragment.

Esta elección se basa en la **mejora estructural del código** al **evitar anulaciones directas de métodos de ciclo de vida**. En lugar de ello, al adoptar el patrón **LifecycleObserver**, se facilita la gestión del ciclo de vida y se mejora la extensibilidad del código.

Evitar anulaciones directas contribuye a un **código más claro y menos propenso a errores**, además de facilitar la comprensión del flujo de vida de la actividad o fragment.

Factores Considerados

- **Claridad del Código**: La adopción de **LifecycleObserver mejora la claridad del código** al separar la lógica del ciclo de vida de la actividad o fragmento.
- **Facilita la Mantenibilidad**: Al evitar anulaciones directas, se hace **más sencillo realizar cambios y mantenimiento en el código** relacionado con el ciclo de vida.
- **Reducción de Errores**: Evitar la anulación directa de métodos ayuda a **prevenir errores comunes asociados con la gestión del ciclo de vida**.

- **Extensibilidad del Código**: El uso de LifecycleObserver **facilita la extensión del código sin modificar directamente los métodos de ciclo de vida**, mejorando la modularidad.

Figura 3: Ciclo de vida

2.4. Arquitectura 4: Usar corrutinas y flujos

El uso de corrutinas y flujos para establecer la **comunicación entre capas, especialmente en operaciones asíncronas**.

La decisión de seguir esta recomendación se fundamenta en varios **factores clave**:

- **Eficiencia en Operaciones Asíncronas**: Permiten realizar tareas de manera concurrente sin bloquear el hilo principal, **mejorando la capacidad de respuesta de la aplicación**.
- **Concisión del Código**: El uso de corrutinas y flujos **simplifica el código** al proporcionar una **sintaxis concisa y estructurada** para manejar operaciones asíncronas.
- **Compatibilidad con Flujos de Datos**: La utilización de flujos facilita **la transmisión de datos entre las capas de la aplicación**.
- **Rendimiento**: Corrutinas están diseñadas para ser livianas y eficientes, **minimizando el impacto en el rendimiento de la aplicación** durante operaciones asíncronas.

La aplicabilidad general, el impacto positivo en **el rendimiento y la mejora de la experiencia del usuario** son factores determinantes para seguir esta recomendación.

Figura 4: Corrutinas y flujos

2.5. Arquitectura 5: Repositorios

Los repositorios se alinean con las mejores prácticas en el desarrollo de software, especialmente cuando se trata de arquitecturas de aplicaciones bien estructuradas. La idea central es **separar las preocupaciones y seguir el principio de responsabilidad única y exponer los datos de la aplicacion mediante un repositorio**. No interactuar de forma directa con la base de datos.

Aplicabilidad General
La arquitectura que separa claramente **la capa de datos y utiliza un repositorio** es generalmente aplicable a una amplia gama de aplicaciones.

Impacto en el Rendimiento:

El impacto en el rendimiento generalmente está más relacionado con la **implementación específica de las operaciones de acceso a datos** y **las consultas realizadas** en la capa de datos. Facilitando la optimización en cada capa.

Experiencia del Usuario:

La separación de capas contribuye a una **experiencia de usuario más consistente y predecible**. Al utilizar un repositorio, los desarrolladores pueden gestionar eficientemente las operaciones relacionadas con los datos, lo que **ayuda a evitar problemas inesperados que podrían afectar negativamente la experiencia del usuario**.

Mantenimiento y Escalabilidad:

La separación de preocupaciones entre la capa de datos y la interfaz de usuario facilita el **mantenimiento a largo plazo**. Las actualizaciones en la lógica de datos pueden realizarse **sin afectar directamente a la interfaz de usuario**, lo que simplifica el proceso de desarrollo y **reduce el riesgo de errores**.

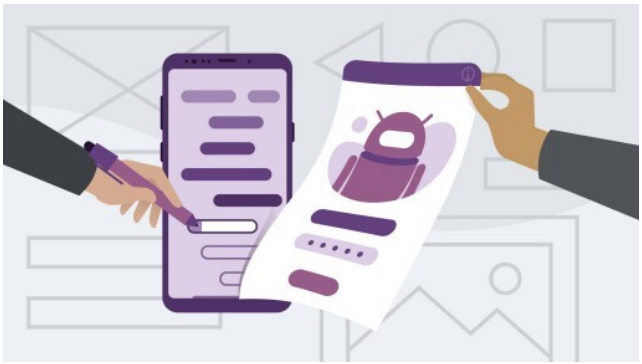

Figura 5: Repositorio

3. ¿Seguiremos estas recomendaciones en el proyecto?

Las **recomendaciones** que seguiremos son las siguientes:

- **Control de Dependencias**: Los beneficios son la **mejora la legibilidad del código, facilita pruebas unitarias y reduce el acoplamiento entre componentes**. Es esencial para la flexibilidad y evolución del sistema.
- **Capa de Datos**: Los beneficios son que facilita el **mantenimiento y la expansión del código al organizar la lógica de datos**. Permite adaptarse a cambios en la lógica empresarial **sin afectar otras partes de la aplicación**.
- **No Anular Métodos de Ciclo de Vida de Activity ni Fragment**: Los beneficios son mejorar la **gestión del ciclo de vida y la extensibilidad del código**, contribuyendo a la calidad del proyecto.

Consideraciones

- **Complejidad del Proyecto**: **Limitar la adopción de arquitecturas** dada la complejidad del diseño de interfaz existente.
- **Tiempo de Desarrollo**: Priorizar eficacia **sin comprometer el tiempo estimado**.
- **Experiencia del Usuario**: Evitar complicaciones excesivas para priorizar la **mejora de la experiencia del usuario y cumplir con los plazos de entrega**.

4. Conlusión

La selección de las arquitecturas "**muy recomendadas**"se basa en un análisis detallado de su impacto en la organización del código, mantenibilidad, rendimiento y experiencia del usuario. Optamos por seguir las recomendaciones de **Control de Dependencias**, que mejora la legibilidad y flexibilidad del código; **Capa de Datos**, facilitando la gestión de datos y adaptabilidad y **No Anular Métodos de Ciclo de Vida**, mejorando la estructura y extensibilidad del código.

Estas decisiones se toman considerando **la complejidad del proyecto, el tiempo estimado de desarrollo y la importancia de priorizar la experiencia del usuario**. La implementación de estas arquitecturas contribuirá significativamente a la **eficacia y éxito del proyecto**.