RESUMEN – TRABAJO DE GRADO

AUTORES	Javier Alfonso Santiago Salazar
FACULTAD	Ingenierías
PLAN DE ESTUDIOS	Ingeniería de Sistemas
DIRECTOR	Dewar Wilmer Rico Bautista
TÍTULO DE LA TESIS	Modelo de Clean Architecture para el desarrollo de proyectos Android nativos, optimizando la modularidad, escalabilidad y mantenibilidad del código.
TITULO EN INGLES	Clean Architecture model for the development of native Android projects, optimizing modularity, scalability and maintainability of the code.

RESUMEN
(70 palabras)

Clean architecture es una solución efectiva en el desarrollo android nativo, ya que mejora la modularidad, escalabilidad y mantenibilidad del código. esta investigación logra adaptar el modelo original propuesto por robert c. martin a un formato mucho más breve, práctico y alineado con mvvm o mvp. se validó su efectividad mediante un caso práctico, demostrando que facilita el trabajo colaborativo, las pruebas unitarias y la evolución del software.

RESUMEN EN INGLES

Clean architecture is an effective solution in native android development, since it improves modularity, scalability and maintainability of the code. this research manages to adapt the original model proposed by robert c. martin to a much shorter, practical format, aligned with mvvm or mvp. its effectiveness was validated through a practical case, demonstrating that it facilitates collaborative work, unit testing and software evolution.

PALABRAS CLAVES	Clean Architecture, desarrollo android, modularidad, escalabilidad, mantenibilidad, principios solid, mvvm, inyección de dependencias
PALABRAS CLAVES EN INGLES	Clean Architecture, android development, modularity, scalability, maintainability, solid principles, mvvm, dependency injection

CARACTERÍSTICAS

PÁGINAS: 72	PLANOS:	ILUSTRACIONES:	CD-ROM:

Modelo de Clean Architecture para el desarrollo de proyectos Android nativos, optimizando la modularidad, escalabilidad y mantenibilidad del código.

Javier Alfonso Santiago Salazar

Facultad de Ingenierías, Universidad Francisco de Paula Santander Ocaña

Ingeniería de Sistemas

Ph.D Dewar Wilmer Rico Bautista

26 de mayo de 2025

Índice

1. Modelo de Clean Architecture para el desarrollo de proyectos Android nativos, optimizando la modularidad, escalabilidad y mantenibilidad del código 8

1.1 Planteamiento del problema 8

1.2 Formulación del problema 12

1.3 Objetivos 12

1.3.1 Objetivo general 12

1.3.2 Objetivos específicos 12

1.4 Justificación 13

1.5 Delimitaciones 14

1.5.1 Geográficas 14

1.5.2 Temporales 14

1.5.3 Conceptuales 15

1.5.4 Operativa 15

2. Marco referencial 17

2.1 Marco histórico 17

2.2 Marco contextual 19

2.3 Marco conceptual 20

2.3.1 Desarrollo Android 20

2.3.2 Clean Architecture 22

2.3.3 Metodología Ágil – SCRUM 24

2.3.4 Patrón de presentación – Model View View Model 26

2.3.5 Modularidad 27

2.4 Marco teórico 29

2.5 Marco legal 31

3. Diseño metodológico 35

3.1 Tipo de investigación 35

3.2 Población y muestra 36

3.2.1 Población 36

3.2.2 Muestra 36

3.3 Diseño de instrumentos de recolección de la información y técnicas de análisis de datos 37

3.4 Análisis de información 38

3.5 Cronograma de actividades 39

4. Resultados 44

4.1 Integración de componentes tradicionalmente separados 44

4.2 Reducción de la complejidad estructural total 45

4.3 Evaluación práctica del impacto en el desarrollo 45

4.4 División por capas modulares 46

4.5 Integración ordenada de componentes separados 46

4.6 Beneficios derivados de este diseño estructurado 47

5. Conclusiones.....60

6. Recomendaciones 65

7. Referencias 67

Apéndice 72

Lista de Figuras

Figura 1 Clean Architecture desing model 24

Figura 2 Instaflix App ScreenShots 51

Figura 3 TheMovieDb.....52

Figura 4 Sprint Table ScreenShot 53

Figura 5 User History ScreenShot.....53

Figura 6 Clean Architecture Desing Model with changes 54

Figura 7 Clean Architecture Desing Model with changes pt.2 55

Figura 8 Clean Architecture Desing Model with changes pt.3 56

Figura 9 Project Structure ScreenShots.....57

Agradecimientos

En primer lugar, agradezco a **Dios**, fuente de vida, fuerza y propósito. Sin Él, nada de esto habría sido posible. Su guía y su presencia constante me sostuvieron en cada etapa de este proceso.

A mi madre, Clara Nineth Salazar, y a mi hermana, Natalia Nineth Santiago, por ser mi refugio y mi motor. Gracias por cada palabra, cada gesto, cada sacrificio. Su amor incondicional ha sido el pilar más fuerte en mi camino.

A mis amigos, aquellos que me han acompañado desde el colegio, la universidad, el trabajo y la vida misma. Gracias por su apoyo, por los momentos compartidos, por su fe en mí incluso cuando yo dudaba. Ustedes han sido una red, una red invisible pero firme que me sostuvo en los momentos más difíciles.

A las familias Trigos, Ascanio y Vázquez, por haber formado parte de este recorrido. Por su apoyo, su cariño y por haberme hecho sentir parte de algo más grande. Mi gratitud hacia ustedes permanecerá intacta.

A mi director de tesis, Dewar Wilmer Rico Bautista, por su orientación, paciencia y compromiso con mi proceso.

Al profesor Eduar Bayona Ibañez, por su constante apoyo y por ser una figura paternal durante mi vida universitaria.

Al profesor Wilder Andrés Duarte Neira, quien sembró en mí la pasión por el desarrollo web y siempre vio en mí un potencial que yo apenas comenzaba a descubrir.

Y al profesor Jerson Javier García Rodríguez, por creer en mí y motivarme a dar ese primer paso profesional, brindándome herramientas y confianza para hacerlo posible.

Agradezco profundamente a **World POS Solutions**, por ser el lugar donde comenzó mi carrera profesional, y por todo el aprendizaje y crecimiento que me brindaron.
Y a **CondorLabs**, por confiar en mí, y por motivarme a crecer tanto personal como profesionalmente.

Y finalmente, gracias a mí. Porque no fue fácil. Porque detrás de este proceso hay noches de insomnio, domingos trabajando, jornadas dobles entre estudio y empleo, estrés, sacrificios y renuncias. Me reconozco el haber persistido, el no haberme rendido cuando el camino se hacía cuesta arriba. Hoy celebro también ese compromiso conmigo mismo y con mis sueños.

Gracias de corazón.

1. Modelo de Clean Architecture para el desarrollo de proyectos Android nativos, optimizando la modularidad, escalabilidad y mantenibilidad del código

1.1 Planteamiento del problema

El desarrollo de aplicaciones nativas de Android presenta una variedad de desafíos a medida que los proyectos aumentan en tamaño y complejidad. Es importante asegurarse de que el código esté organizado, y que pueda crecer con el proyecto y sea fácil de trabajar. El código de la aplicación a menudo se vuelve demasiado grande y está interconectado, lo que dificulta agregar nuevas funciones, usar partes del código nuevamente y probar el código en partes pequeñas. La ausencia de una estructura bien organizada puede generar mayores gastos de mantenimiento y desafíos a la hora de modificar el software para satisfacer las necesidades cambiantes (Feng, Q., Ji, H., Ma, X., & Liang, P, 2024).

Para manejar estos problemas, necesitamos una arquitectura que divida claramente las tareas y facilite su desarrollo y mantenimiento a lo largo del tiempo. La Clean Architecture de Robert C. Martin, ahora es muy apreciada por los desarrolladores de Android para resolver ciertos problemas. Pero existen dificultades para ponerla en práctica en Proyectos de Android y es necesario ajustarla para que se ajuste a las características específicas de este entorno (Rachovski, T., Hadzhikoleva, S., Hadzhikolev, E., & Lengerov, A., 2022).

La Clean Architecture promete ser una solución efectiva para estos problemas, ya que proporciona una estructura clara y modular para el código, lo que facilita la mantenibilidad, escalabilidad y seguridad de las aplicaciones. Sin embargo, según Lano, K.,

& Tehrani, S. Y., en el 2023, la implementación de esta arquitectura en proyectos de Android es un desafío, ya que requiere una comprensión profunda de los principios de la Clean Architecture y de las características específicas de Android.

Es debido hacer que la arquitectura limpia sea más fácil y rápida de usar para el desarrollo de aplicaciones de Android, pero manteniendo sus beneficios clave, como ser modular, escalable y sostenible, para así lograr equilibrar la separación de capas y la simplicidad para facilitar el desarrollo y mantener el código mantenible para la posteridad (Lano, K., & Tehrani, S. Y., 2023).

Tratando esto, es de suma importancia investigar y desarrollar estrategias efectivas para implementar la Clean Architecture en proyectos de Android, considerando las características específicas de este entorno y todos los desafíos que conlleva. Como por incluye la identificación de patrones y principios de diseño que se adapten a todas las necesidades de los proyectos de Android, y la creación de herramientas, recursos o helpers que faciliten la implementación de la Clean Architecture en estos proyectos.

En la actualidad, uno de los principales desafíos en proyectos de desarrollo de software es la incorporación de nuevos desarrolladores al equipo. Estos deben comprender la arquitectura y la forma en que se desarrolló el proyecto para poder contribuir de manera efectiva. Sin embargo, la subjetividad en la interpretación de la arquitectura limpia y la falta de lineamientos claros pueden generar inconsistencias y afectar la estabilidad del proyecto, ya que para un proyecto puede

utilizarse una adaptación del modelo original (Ferreira, V. B. S., Ferreira, C. A., & Grande, E. T. G., 2022).

Siguiendo a Ferreira, V. B. S., Ferreira, C. A., & Grande, E. T. G. en el 2022, es fundamental establecer lineamientos y principios sólidos para la implementación de la arquitectura limpia, permitiendo que cada empresa adapte esta metodología a sus necesidades específicas. Sin embargo, es crucial mantener los principios fundamentales que la componen, evitando tergiversaciones que puedan comprometer su eficacia.

La arquitectura limpia debe ser flexible y adaptable a las necesidades del proyecto, pero también debe mantener su esencia y propósito. Es importante encontrar un equilibrio entre la flexibilidad y la coherencia, asegurando que el proyecto siga siendo mantenible, escalable y sostenible a largo plazo (Bukovcan, M., Blazevic, D., Nenadic, K., & Stevic, M., 2022)

Para lograr esto, es esencial:

Establecer lineamientos y principios claros para la implementación de la arquitectura limpia.

Proporcionar documentación y recursos adecuados para que los nuevos desarrolladores puedan comprender la arquitectura y contribuir de manera efectiva.

Fomentar una cultura de colaboración y comunicación efectiva dentro del equipo.

Realizar revisiones y ajustes periódicos para asegurar que el proyecto siga siendo coherente con los principios de la arquitectura limpia.

Sin embargo, la efectividad de esta arquitectura depende en gran medida de la aplicación de principios de diseño sólidos, como los principios SOLID (Single responsibility, Open/closed, Liskov substitution, Interface segregation y Dependency inversion). Estos principios son fundamentales para garantizar que el código sea mantenible, escalable y fácil de entender, lo que a su vez permite a los desarrolladores crear sistemas más robustos y flexibles. No obstante, la implementación de estos principios en la práctica puede ser un desafío, especialmente en proyectos complejos donde la presión para cumplir con plazos y presupuestos puede llevar a sacrificar la calidad del código. Por lo tanto, es crucial investigar y desarrollar estrategias efectivas para integrar los principios SOLID en la Clean Architecture, asegurando así la creación de sistemas de software que sean verdaderamente sostenibles y escalables (De Dieu, M. J., Liang, P., Shahin, M., & Khan, A. A., 2023).

Según Adwinda, C., & Suryodiningrat, D. en el 2020, La combinación de patrones de presentación como Model-View-ViewModel (MVVM) o Model-View-Presenter (MVP) con Clean Architecture ofrece una poderosa estrategia para crear aplicaciones más mantenibles, escalables y fáciles de probar. Al separar las preocupaciones del negocio de las tecnologías específicas, Clean Architecture proporciona una base sólida para la arquitectura de la aplicación. Por otro lado, patrones como MVVM y MVP permiten una clara separación de responsabilidades entre la lógica de presentación y el dominio del negocio, lo que facilita la creación de interfaces de usuario más flexibles y fáciles de mantener. Al integrar estos patrones, se puede lograr una arquitectura que no solo es más modular y sostenible, sino que también permite una mayor flexibilidad y escalabilidad. Además, la separación clara de responsabilidades entre las capas de presentación, dominio y datos permite una mayor

facilidad para probar y depurar el código, lo que a su vez reduce el tiempo de desarrollo y mejora la calidad general de la aplicación.

1.2 Formulación del problema

¿Cómo puede adaptarse el modelo de Clean Architecture, diseñado originalmente por Robert C. Martin, para mejorar la modularidad, escalabilidad y mantenibilidad en proyectos de desarrollo Android nativos, sin incrementar la complejidad del proceso de implementación?

1.3 Objetivos

1.3.1 Objetivo general

Proponer un modelo de Clean Architecture para el desarrollo de proyectos Android nativos, optimizando la modularidad, escalabilidad y mantenibilidad del código.

1.3.2 Objetivos específicos

- Diseñar la estructura de Clean Architecture en proyectos Android nativos, integrando componentes que tradicionalmente estarían separados, como los viewmodels, presenters, repositories y datasources, para reducir la complejidad y facilitar el desarrollo.
- Aplicar los principios SOLID y patrones de presentación como MVVM o MVP

en la arquitectura, promoviendo la separación de responsabilidades y la inyección de dependencias, asegurando la independencia entre las diferentes capas de la aplicación.

- Validar la eficiencia y mantenibilidad del código mediante el uso de Clean Architecture adaptada en un proyecto Android de ejemplo, realizando pruebas de integración, unitarias y de rendimiento, para asegurar que los cambios propuestos mejoren la calidad del software sin comprometer su funcionalidad o escalabilidad.

1.4 Justificación

En el desarrollo de aplicaciones Android, mantener el código de alta calidad y la escalabilidad es esencial ya que existe un crecimiento continuo y un cambio en las demandas del mercado. A medida que los proyectos crecen y se vuelven más grandes y complicados, los programadores deben mantener el código organizado de una manera que pueda cambiar y agregar nuevas features fácilmente sin que sea difícil trabajar con el código o que exista mucho acoplamiento. Adoptar una arquitectura robusta y bien estructurada es esencial para abordar estos desafíos. (Mateus, B. G., & Martinez, M., 2020).

Clean Architecture de Robert C. Martin, es una buena opción porque separa claramente las diferentes partes del código y de lo que consideramos la app en sí, lo que facilita el trabajo y ayuda a que el código sea escalable y dure por mucho más tiempo. Pero, implementar esta característica en Android nativo puede ser demasiado complicado, sobre todo si llegamos a necesitar desarrollos rápidos ya que como tal el marco de trabajo de Android es bastante

complejo y se le añaden muchas otras cosas externas y todo este conjunto puede complicar el código (Rubin, J., Henniche, A. N., Moha, N., Bouguessa, M., & Bousbia, N., 2019).

Este proyecto es de suma importancia ya que adapta lo que es Clean Architecture para Android de una manera más sencilla, manteniendo sus beneficios como la facilidad en el testing, la independencia de las capas y la adición de nuevas partes de código productivo sin problemas. El cambio propuesto hará que el código sea mejor, más claro y más económico de mantener, facilitará la actualización en el futuro y disminuirá la curva de aprendizaje para los nuevos.

Esta investigación ayuda a los desarrolladores de Android nuevos y experimentados brindándoles una forma comprobada de crear un buen software. El uso de principios SOLID y patrones de presentación como MVVM o MVP lo cual fomenta buenos hábitos que ayudan a que los proyectos de software crezcan bien con el tiempo.

1.5 Delimitaciones

1.5.1 Geográficas.

La investigación y desarrollo del proyecto se realiza en Ocaña, Norte de Santander, Colombia. Aunque su aplicación está orientada al desarrollo Android nativo a nivel nacional puesto que: los recursos, pruebas y validaciones del proyecto se llevarán a cabo en este entorno geográfico.

1.5.2 Temporales.

La duración de este proyecto será de seis (6) meses calendario contados a partir de la fecha de aprobación. Durante este tiempo se trabajará en la optimización de la Clean Architecture para proyectos Android nativos, lo que incluye el análisis de la arquitectura existente y la integración de mejoras en modularidad, escalabilidad y mantenibilidad.

Se desarrollará una aplicación de ejemplo llamada InstaFlix, en la cual se aplicará la Clean Architecture optimizada junto con patrones de presentación como MVVM y principios SOLID. A través de iteraciones continuas, se realizarán pruebas unitarias para verificar el comportamiento.

1.5.3 Conceptuales.

Este proyecto se delimita al análisis y adaptación de Clean Architecture en el desarrollo Android nativo. No incluye la implementación de Clean Architecture en otras plataformas como iOS o en aplicaciones web directamente. Se centra de forma exclusivamente en los principios SOLID, la separación de respinsabilidades, y la simplificación de las capas y los componentes dentro del contexto del desarrollo móvil y arquitectura limpia, dejando de lado otras metodologías de arquitectura no relacionadas con Clean Architecture.

1.5.4 Operativa.

El proyecto se centra en la implementación de Clean Architecture en el desarrollo de aplicaciones Android nativas. Se utilizará un proyecto de ejemplo llamado "InstaFlix" para así ilustrar el cómo se aplica esta arquitectura. La implementación de Clean Architecture se llevará a cabo bajo un enfoque ágil (Scrum), lo que permite la entrega iterativa del software

para así alcanzar mínimamente un Producto mínimo viable, facilitando la adaptación y optimización de la arquitectura en ciclos de desarrollo cortos. La investigación también se enfocará en la integración de patrones de presentación como MVVM o MVP y la inyección de dependencias, en lugar de otras arquitecturas o patrones de diseño.

2. Marco referencial

2.1 Marco histórico

Android Nativo y su lanzamiento inicial en 2008, ha evolucionado significativamente como plataforma y entorno de trabajo. En sus primeros años, el desarrollo de aplicaciones nativas se centraba en el uso de componentes básicos como Activities, Fragments y Services, junto con el lenguaje Java. Sin embargo, estas aplicaciones solían ser monolíticas y totalmente estructuradas, lo que a día de hoy se consideraría "código espagueti", lo cual dificultaba su escalabilidad y mantenibilidad en proyectos grandes (Coppola, R., Fulcini, T., & Torchiano, M., 2024).

A medida que pasaba el tiempo los desarrolladores Android y todo el marco de trabajo que lo envolvía lograron en 2017, introducir Kotlin by Google como lenguaje oficial para el desarrollo en Android, ofreciendo características modernas como programación funcional y seguridad contra nulos. Esto marcó un hito importante al facilitar la creación de aplicaciones más limpias y menos propensas a errores.

Pero faltaba algo que lograra dar por sentado el modelo de trabajo que se implementaría en adelante y que sentara un punto en la historia del desarrollo Android, pero gracias a la adopción de Clean Architecture, propuesta por Robert C. Martin en 2012, que comenzó a ganar relevancia con el aumento de la complejidad en los proyectos se logró marcar un antes y un después. Clean Architecture promueve una separación clara de responsabilidades mediante capas como Domain, Data, y Presentation. Su adopción en Android requirió

adaptaciones para integrar componentes específicos de la plataforma, como repositorios y fuentes de datos locales o remotas (Callan, J., & Petke, J., 2024).

Los primeros años de su adopción estuvieron marcados por desafíos técnicos, como la inversión de dependencias utilizando frameworks como Dagger o más recientemente Hilt, ambos esenciales para mantener la modularidad y escalabilidad gracias a la inyección de dependencias.

El desarrollo Android en general siempre ha estado en evolución constante y por esa razón en paralelo, según Ji, K. en el 2024 los patrones de presentación comenzaron a implementarse para gestionar mejor la lógica de la interfaz de usuario:

> Model-View-Presenter (MVP): Popularizado en la década de 2010, MVP separa la lógica de negocio de la vista, asignando al Presenter el papel de intermediario. Sin embargo, su implementación a menudo llevó a Presenters muy grandes, conocidos como "God Objects".
>
> Model-View-ViewModel (MVVM): Con la introducción de la librería Android Architecture Components en 2017, MVVM se convirtió en el estándar recomendado por Google. Este patrón utiliza LiveData y ViewModel para conectar de manera reactiva las capas de presentación y dominio, mejorando la mantenibilidad.

Ambos patrones, cuando se combinan con Clean Architecture, permiten crear aplicaciones más modulares y fáciles de probar.

Ahora en Android existe la manera de mantener aplicaciones más modulares, escalables y

fáciles de testear. Pero en empresas o proyectos donde se necesitan de muchos desarrolladores para un solo proyecto adoptaron Scrum, como marco ágil, el cual fue introducido por Ken Schwaber y Jeff Sutherland en 1995. Aunque originalmente no fue diseñado específicamente para el desarrollo de software, se convirtió en un estándar en la industria tecnológica debido a su flexibilidad y enfoque iterativo. La combinación de Scrum con prácticas de arquitectura como Clean Architecture permite que los equipos aborden proyectos grandes dividiendo tareas complejas en entregas incrementales (Arango, E. C., & Loaiza, O. L., 2021).

En el desarrollo Android, Scrum ha permitido una integración efectiva de patrones de diseño y arquitecturas al promover revisiones constantes, retrospectivas y adaptación de procesos. (Verwijs, C., & Russo, D., 2022) Esto ha sido clave para mantener la calidad del software y cumplir con los objetivos de negocio.

A lo largo del tiempo, la combinación de Clean Architecture, patrones de presentación como MVP y MVVM, y metodologías ágiles como Scrum ha transformado el desarrollo nativo en Android. Este enfoque totalmente holístico logra garantizar que las aplicaciones no solo sean técnicamente sólidas, sino que también respondan de manera efectiva a las necesidades cambiantes del usuario y del mercado.

2.2 Marco contextual

La investigación se lleva a cabo en el entorno de desarrollo Android nativo,

específicamente utilizando el IDE Android Studio como herramienta principal. El proyecto se desarrolla en Ocaña, Norte de Santander, Colombia, y se basa en una aplicación de ejemplo llamada InstaFlix, creada en el año 2023. InstaFlix fue diseñada inicialmente como una plataforma de información de contenido multimedia, pero su estructura original no cumplía con los estándares óptimos de modularidad, escalabilidad y mantenibilidad.

La implementación de Clean Architecture y patrones de presentación como MVVM (Model-View-ViewModel) en InstaFlix busca abordar estos desafíos. Este enfoque mejora la separación de responsabilidades entre capas, minimiza el acoplamiento entre los módulos y facilita el testing unitario e integral de los componentes. Además, se plantea como un marco de trabajo sólido para guiar la evolución futura de la aplicación, permitiendo la incorporación de nuevas funcionalidades sin comprometer la calidad del software.

El uso de Clean Architecture en InstaFlix refleja la tendencia global hacia arquitecturas modulares y adaptables en el desarrollo móvil. Este enfoque es particularmente relevante en un contexto donde las demandas de los usuarios y las tecnologías evolucionan rápidamente. Al integrar principios como SOLID y herramientas modernas de inyección de dependencias (por ejemplo, Dagger Hilt), el proyecto también se alinea con las mejores prácticas promovidas por Google y la comunidad de desarrolladores Android.

2.3 Marco conceptual

2.3.1 Desarrollo Android

El desarrollo Android abarca el diseño, implementación y mantenimiento de aplicaciones

destinadas al sistema operativo Android, ampliamente utilizado en dispositivos móviles, televisores, relojes inteligentes y otros aparatos conectados. Este entorno ha evolucionado significativamente desde su introducción, permitiendo a los desarrolladores trabajar con herramientas avanzadas y arquitecturas que aseguran la eficiencia, la modularidad y la calidad del código (Ardito, L., Coppola, R., Malnati, G., & Torchiano, M., 2020).

El desarrollo de aplicaciones Android ofrece flexibilidad al permitir el uso de diferentes lenguajes de programación, como Java y Kotlin, cada una con sus marcadas diferencias a nivel de eficacia, como mencionan Ardito, L., Coppola, R., Malnati, G., & Torchiano, M. en el 2020. El entorno de desarrollo se centra en Android Studio, un IDE que proporciona soporte para diseño visual, análisis de rendimiento y emulación de dispositivos.

A lo largo del tiempo, el desarrollo Android ha adoptado patrones arquitectónicos como MVP (Model-View-Presenter) y MVVM (Model-View-ViewModel), que favorecen una clara separación de responsabilidades, facilitando el mantenimiento y las pruebas. Más recientemente, la Clean Architecture se ha posicionado como un estándar para proyectos que requieren escalabilidad y modularidad, permitiendo dividir las aplicaciones en capas independientes.

El desarrollo de aplicaciones Android a menudo utiliza metodologías ágiles como Scrum para gestionar proyectos de manera iterativa y colaborativa. Esto permite realizar entregas incrementales, mejorando continuamente las funcionalidades de la aplicación. (Gao, S., Liu, L., Liu, Y., Liu, H., & Wang, Y., 2020).

La diversidad de dispositivos y las expectativas crecientes de los usuarios hacen que el desarrollo Android sea un campo dinámico y desafiante y que cambia constantemente. La adopción de arquitecturas limpias y patrones sólidos no solo mejora la calidad del software, sino que también facilita la incorporación de nuevos desarrolladores a los equipos de trabajo, asegurando la continuidad del proyecto, que a la larga es de mucha importancia por la alta rotación del sector.

2.3.2 Clean Architecture

La Clean Architecture es un modelo arquitectónico diseñado para estructurar sistemas de software de manera modular, escalable y mantenible. Su enfoque principal es la separación de responsabilidades, dividiendo las aplicaciones en capas que abstraen la lógica del negocio de los detalles de implementación técnica y se basa en la independencia entre las capas de la aplicación, lo que permite que los componentes centrales, como la lógica de negocio, no dependan de detalles externos, como bases de datos o interfaces de usuario. Este principio asegura que los cambios en las tecnologías externas no afecten la funcionalidad principal de la aplicación (De Dieu, M. J., Liang, P., Shahin, M., & Khan, A. A., 2023).

La arquitectura generalmente se divide en las siguientes capas como comentan Lano, K., & Tehrani, S. Y., en el 2023:

Entidad o Dominio: Contiene las reglas de negocio fundamentales y las entidades de la aplicación.

Casos de Uso: Define las operaciones específicas del sistema, orquestando la lógica de negocio.

Adaptadores: Incluyen las interfaces de usuario y los controladores, transformando los datos para su presentación.

Infraestructura: Maneja las dependencias externas, como bases de datos y servicios API. Ventajas de la Clean Architecture

Escalabilidad: Facilita el crecimiento del proyecto sin afectar su estructura.

Mantenibilidad: Promueve un código más limpio y fácil de entender.

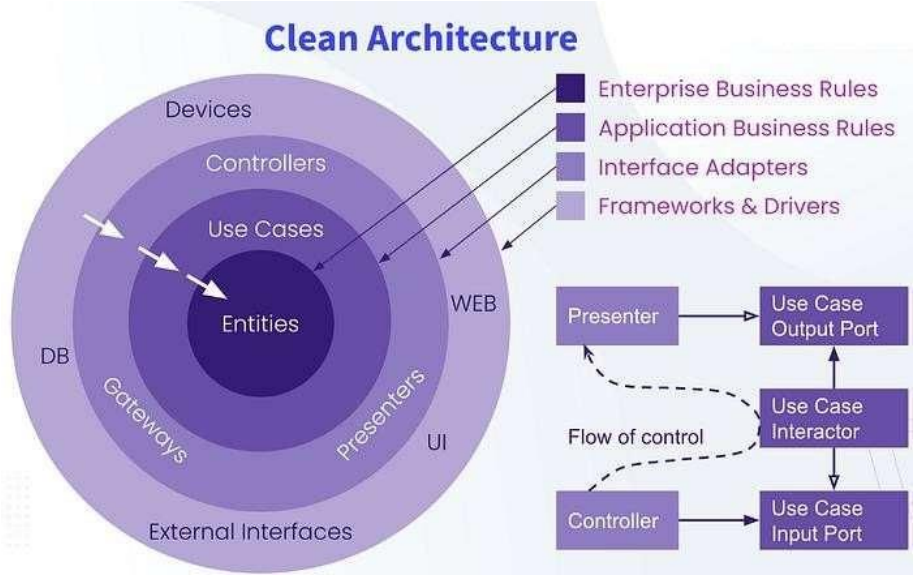
Reutilización: Las capas bien definidas permiten reaprovechar módulos y componentes.

Pruebas Efectivas: Al separar responsabilidades, la arquitectura facilita la implementación de pruebas unitarias e integrales.

En el contexto del desarrollo Android, la Clean Architecture se adapta bien mediante el uso de herramientas como los ViewModels, que trabajan en conjunto con patrones de presentación como MVVM o MVP. Estos patrones potencian la modularidad y simplifican la interacción entre las capas, optimizando la calidad del software. Este enfoque arquitectónico no solo mejora la organización del código, sino que también establece una base sólida para futuros desarrollos y la incorporación de nuevas tecnologías (Boukhary, S., & Colmenares, E., 2019).

Figura 1

Clean Architecture desing model



Fuente: Semih Tekin, 2023 https://semihtekin.medium.com/what-is-the-clean-architecture-c80c2a2ff69a

2.3.3 Metodología Ágil – SCRUM

La metodología ágil SCRUM es un marco de trabajo diseñado para gestionar proyectos de desarrollo de software de manera iterativa y colaborativa. Su enfoque se centra en la entrega continua de valor al cliente, mediante la división del proyecto en ciclos cortos de trabajo llamados sprints, que generalmente tienen una duración de entre dos y cuatro semanas. SCRUM fomenta la adaptación constante a los cambios y la mejora continua a través de una interacción frecuente con los interesados y un enfoque en el trabajo en equipo (Neto, A. S., Ramos, F., Albuquerque, D., Dantas, E., Perkusich, M., Almeida, H., & Perkusich, A., 2023)

El equipo SCRUM se compone de tres roles principales: el Product Owner, el Scrum Master y el Equipo de Desarrollo. El Product Owner es el encargado de definir y priorizar todas las funcionalidades necesarias, representadas en un documento vivo llamado Product Backlog. El Scrum Master actúa como facilitador, asegurándose de que el equipo siga las prácticas de SCRUM y ayudando a eliminar cualquier impedimento que pueda surgir. El Equipo de Desarrollo es el encargado de entregar los incrementos de producto al final de cada sprint (Oliveira, H., & Araújo, C., 2021).

Una de las características principales de SCRUM es su énfasis en las reuniones regulares, como las planificaciones de sprint, las reuniones diarias (Daily Scrum), las revisiones de sprint y las retrospectivas. Estas ceremonias aseguran que el equipo mantenga una comunicación abierta y una visión clara del progreso del proyecto, al mismo tiempo que identifican oportunidades de mejora tanto en el proceso como en el producto (Lawong, D. A., & Akanfe, O., 2024).

SCRUM fomenta la entrega incremental de un producto funcional al final de cada sprint, lo que permite a los clientes y partes interesadas evaluar el progreso y dar retroalimentación temprana. Este enfoque no solo mejora la calidad del producto, sino que también reduce los riesgos asociados con los cambios en los requisitos o prioridades durante el desarrollo.

En el contexto del desarrollo de software, SCRUM se combina frecuentemente con otras prácticas ágiles, como la integración continua y el desarrollo dirigido por pruebas (TDD). Esta combinación permite a los equipos maximizar la calidad y minimizar los tiempos de

entrega, haciendo de SCRUM un marco ideal para entornos dinámicos y competitivos. Además, su flexibilidad y enfoque en el empoderamiento del equipo lo han convertido en una de las metodologías ágiles más populares a nivel mundial (Feng, X., & Periyasamy, K., 2022).

2.3.4 Patrón de presentación – Model View View Model

El patrón de presentación Model-View-ViewModel (MVVM) es un enfoque arquitectónico diseñado para separar la lógica de la interfaz de usuario de la lógica del negocio y de la gestión de datos, con el objetivo de facilitar el desarrollo, mantenimiento y prueba de aplicaciones. Este patrón se compone de tres componentes principales: el Modelo (Model), la Vista (View) y el Modelo de Vista (ViewModel), cada uno con responsabilidades específicas (Kouraklis, J., 2016).

Como lo comenta Kouraklis, J. en el 2016, MVVM se divide en:

> El Modelo (Model) es responsable de gestionar los datos y la lógica del negocio. Incluye las estructuras de datos, las reglas de negocio y las operaciones necesarias para interactuar con las fuentes de datos, como bases de datos o servicios API. El modelo es independiente de la interfaz de usuario, lo que permite que se reutilice en diferentes contextos.
>
> La Vista (View) corresponde a la representación visual de la aplicación. Es el componente que los usuarios finales ven e interactúan. En este patrón, la vista se encarga exclusivamente de la presentación de los datos proporcionados por el ViewModel,

evitando incluir lógica del negocio o procesos complejos dentro de ella. Esto mejora la claridad y simplicidad del código de la interfaz de usuario.

El Modelo de Vista (ViewModel) actúa como un puente entre el Modelo y la Vista. Se encarga de exponer los datos del modelo en un formato que la vista pueda consumir fácilmente. Además, gestiona la interacción del usuario y actualiza el modelo cuando sea necesario. Una característica clave del ViewModel es su capacidad para soportar el data binding (vinculación de datos), lo que permite sincronizar automáticamente los cambios en el modelo con la vista y viceversa.

MVVM es especialmente útil en aplicaciones que hacen uso intensivo de interfaces de usuario dinámicas y complejas, como las desarrolladas para plataformas móviles y de escritorio. Al mantener una separación clara entre las capas, este patrón facilita la implementación de pruebas unitarias, mejora la escalabilidad y reduce el acoplamiento entre componentes, permitiendo que los desarrolladores trabajen de manera más eficiente y ordenada. (Sheikh, W., & Sheikh, N., 2020).

2.3.5 Modularidad

La modularidad es un principio fundamental en el desarrollo de software que se refiere a la división de un sistema complejo en componentes más pequeños y manejables, denominados módulos. Cada módulo está diseñado para cumplir una función específica de manera independiente, lo que permite una organización clara y estructurada del código. Este enfoque facilita la comprensión, el desarrollo, la prueba y el mantenimiento del software, ya que los cambios o mejoras pueden realizarse en un módulo sin afectar significativamente a

los demás. (Sullivan, K. J., Griswold, W. G., Cai, Y., & Hallen, B., 2001))

Un sistema modular presenta varias características clave. En primer lugar, los módulos deben ser autónomos, es decir, cada uno debe ser capaz de realizar su función sin depender excesivamente de otros. En segundo lugar, deben estar bien definidos, con interfaces claras que especifiquen cómo interactúan con otros módulos. Por último, deben ser reutilizables, lo que significa que un módulo puede ser integrado en diferentes sistemas o proyectos con poco o ningún ajuste (Moon, E., & Howison, J., 2024).

La modularidad tiene múltiples ventajas. Reduce la complejidad del desarrollo al permitir que los equipos trabajen en diferentes módulos simultáneamente. También mejora la escalabilidad, ya que facilita la adición de nuevas funcionalidades sin reestructurar el sistema completo. Además, incrementa la mantenibilidad al hacer que sea más sencillo localizar y corregir errores, o actualizar partes específicas del sistema.

Este principio se aplica ampliamente en arquitecturas de software como Clean Architecture, donde se fomenta la separación en capas con responsabilidades específicas. También es una base esencial en el diseño orientado a objetos y en el desarrollo de sistemas basados en microservicios. Al adoptar un enfoque modular, los desarrolladores pueden crear sistemas más robustos, flexibles y

preparados para adaptarse a las necesidades cambiantes de los proyectos. (Moon, E., & Howison, J., 2024).

2.4 Marco teórico

El desarrollo de software para aplicaciones móviles, específicamente en Android nativo, ha evolucionado significativamente desde su inicio. Android, como sistema operativo, introdujo herramientas y estándares que facilitaron la creación de aplicaciones eficientes y adaptables, permitiendo a los desarrolladores estructurar su código de manera más organizada. Este entorno ha sido el punto de partida para la implementación de diferentes arquitecturas y metodologías de desarrollo, como la Arquitectura Limpia (Clean Architecture), que promueve la separación de responsabilidades en capas bien definidas, optimizando así la escalabilidad y mantenibilidad del código (Rachovski, T., Hadzhikoleva, S., Hadzhikolev, E., & Lengerov, A., 2022). Este enfoque arquitectónico permite que las aplicaciones sean más flexibles frente a cambios tecnológicos o de negocio, al tiempo que garantiza que las dependencias externas no interfieran directamente con la lógica central del sistema.

En paralelo al desarrollo técnico, las metodologías ágiles, como SCRUM, han transformado la manera en que los equipos de software abordan los proyectos. SCRUM ofrece un marco iterativo e incremental que fomenta la colaboración entre los miembros del equipo y los interesados en el proyecto. (Verwijs, C., & Russo, D., 2022) En el contexto del desarrollo Android, esta metodología se alinea perfectamente con la necesidad de entregar productos funcionales rápidamente y adaptarse a las necesidades cambiantes de los usuarios. SCRUM no solo mejora la eficiencia del equipo, sino que también facilita la integración continua de componentes arquitectónicos como los definidos por Clean Architecture, asegurando que cada iteración del producto sea funcional y de alta calidad.

El patrón de presentación Model-View-ViewModel (MVVM) se ha consolidado como una solución popular en el desarrollo Android para manejar la interacción entre la interfaz de usuario y la lógica de negocio. MVVM separa claramente la lógica de presentación del resto de la aplicación, lo que facilita las pruebas unitarias y reduce el acoplamiento. La modularidad inherente a este patrón permite la reutilización de componentes, contribuyendo a una arquitectura más ordenada y fácil de mantener. La combinación de MVVM con Clean Architecture en aplicaciones Android ha demostrado ser una estrategia poderosa para construir sistemas robustos y preparados para el futuro (Sheikh, W., & Sheikh, N., 2020).

Un aspecto clave en el desarrollo de aplicaciones modernas es la modularidad, que implica dividir una aplicación en componentes o módulos independientes que pueden desarrollarse, probarse y desplegarse de manera autónoma. La modularidad no solo mejora la organización del código, sino que también acelera el desarrollo al permitir que diferentes equipos trabajen en paralelo en distintos módulos. Además, facilita la integración de nuevas funcionalidades sin afectar el núcleo de la aplicación, un beneficio crítico en un entorno tan dinámico como el desarrollo móvil (Moon, E., & Howison, J., 2024).

Otro concepto fundamental en el desarrollo Android es el acoplamiento, que se refiere al grado de dependencia entre los diferentes componentes de un sistema. Una arquitectura con bajo acoplamiento es más flexible, ya que los cambios en un componente no afectan directamente a los demás. Este principio es esencial en la implementación de Clean Architecture, donde se busca que las dependencias fluyan hacia adentro, es decir, desde

componentes externos hacia el núcleo de la aplicación, nunca en sentido contrario. Este enfoque garantiza que la lógica de negocio esté protegida frente a cambios en las capas externas, como la interfaz de usuario o los frameworks específicos. (Lano, K., & Tehrani, S. Y., 2023)

La combinación de estos elementos –Clean Architecture, SCRUM, MVVM, modularidad y bajo acoplamiento– no solo optimiza el desarrollo técnico de aplicaciones Android, sino que también permite crear soluciones sostenibles a largo plazo. En proyectos como "InstaFlix", la integración de estas prácticas no solo mejora la calidad del software, sino que también establece un estándar para futuros desarrollos. La implementación iterativa de estas estrategias, acompañada de pruebas unitarias y de integración, asegura que las aplicaciones no solo sean funcionales, sino también escalables y mantenibles en un entorno tecnológico en constante evolución.

2.5 Marco legal

La ley del software legal, esta ley se refiere a la protección de los derechos de autor en Colombia. Recuerde: el software es un activo, además está protegido por el Derecho de Autor y la Ley 603 de 2000 (El Congreso de Colombia) obliga a las empresas a declarar si los problemas de software son o no legales.

La cual dicta que cualquier ciudadano tendrá la posibilidad de acceder a su información personal y solicitar la supresión o corrección de la misma frente a toda base de datos en que se

encuentre registrado.

Establece los principios que deben ser obligatoriamente observados por quienes hagan uso, de alguna manera realicen el tratamiento o mantengan una base de datos con información personal, cualquiera que sea su finalidad.

Aclara la diferencia entre clases de datos personales construyendo las bases para la instauración de los diversos grados de protección que deben presentar si son públicos o privados, así como las finalidades permitidas para su utilización.

Crea una especial protección a los datos de menores de edad.

Establece los lineamientos para la cesión de datos entre entidades y los procesos de importación y exportación de información personal que se realicen en adelante.

Define las obligaciones y responsabilidades que empresas de servicios tercerizados tales como Call y Contact Center, entidades de cobranza y, en general, todos aquellos que manejen datos personales por cuenta de un tercero, deben cumplir en adelante.

Asigna la vigilancia y control de las bases de datos personales a la ya creada Superintendencia delegada para la Protección de Datos Personales, de la Superintendencia de Industria y Comercio.

Crea el Registro Nacional de Bases de Datos.

Establece una serie de sanciones de carácter personal e institucional dirigidas a entidades y funcionarios responsables del cumplimiento de sus lineamientos.

Algunas de las principales regulaciones incluyen:

Protección de Datos Personales: La ley colombiana exige que las empresas protejan la información personal de sus clientes y empleados. El desarrollo de software a medida debe cumplir con los estándares de seguridad de datos establecidos por la Ley 1581 de 2012.

Propiedad Intelectual: Es vital entender los aspectos legales de la propiedad intelectual al desarrollar software personalizado. Esto incluye licencias de software, derechos de autor y acuerdos de propiedad intelectual.

Normas de Seguridad: Las regulaciones exigen que el software cumpla con estándares de seguridad cibernética, especialmente en sectores sensibles como el financiero o la salud.

El desarrollo de software sí está regulado en Colombia y es reconocido por la Ley 23 de 1982 y por la Decisión Andina 351 de 1993 como una obra literaria y bajo esta calidad se protege por las normas de propiedad intelectual y el derecho de autor. La creación “literaria” del ingeniero o desarrollador se refleja en lo que técnicamente se conoce como código fuente.

Según el artículo 271 del Código Penal Colombiano, la reproducción, almacenamiento,

adquisición para la venta o distribución o suministro de cualquier tipo de reproducciones realizadas sin autorización previa y expresa del titular, de cualquier soporte lógico o programa de ordenador, se constituye como un delito denominado Defraudación a los derechos patrimoniales de autor. La norma dice que quien viole los derechos de autor en el país se expone a recibir de dos a cinco años de prisión y una multa de 20 a 1.000 salarios mínimos legales mensuales vigentes.

3. Diseño metodológico

3.1 Tipo de investigación

La metodología utilizada para este proyecto se basa en un enfoque de investigación cuantitativa con alcance descriptivo y experimental. Este enfoque permite analizar datos de manera objetiva, cuantificando características y patrones relacionados con la Clean Architecture en proyectos Android. A través del alcance descriptivo, se busca identificar y detallar las mejoras posibles en la modularidad, escalabilidad y mantenibilidad de esta arquitectura, basándose en información obtenida tanto de la literatura especializada como de experimentos controlados. (Em, S., 2024).

La investigación experimental aplicada en este proyecto consiste en implementar la Clean Architecture optimizada en un entorno controlado, utilizando una aplicación prototipo como objeto de estudio. En este contexto, se manipularán variables independientes, como la implementación de principios SOLID o la adopción de patrones de diseño específicos, para observar su efecto sobre variables dependientes, como el rendimiento del sistema, el tiempo de desarrollo y la facilidad de mantenimiento. Estos resultados se medirán utilizando herramientas automatizadas y métricas establecidas, garantizando la objetividad del análisis. (Knight, K. L., 2010)

El enfoque experimental permite validar hipótesis relacionadas con la optimización propuesta y, mediante la recopilación de datos empíricos, respaldar con evidencia la efectividad de las mejoras implementadas. La naturaleza descriptiva de la investigación

complementa este proceso, al proporcionar un marco teórico que contextualiza y sustenta los experimentos realizados, contribuyendo al cumplimiento de los objetivos planteados.

3.2 Población y muestra

3.2.1 Población

Para la realización de este proyecto, se tomó en cuenta como población a los desarrolladores de software que trabajan con proyectos Android en los cuales se implementa o se podría implementar la arquitectura Clean Architecture. Esta población incluye tanto desarrolladores principiantes como avanzados, así como equipos de trabajo que utilizan esta arquitectura en sus proyectos, dentro de diversas industrias y aplicaciones móviles.

3.2.2 Muestra

Se tomará una muestra representativa de la población total para llevar a cabo la investigación.

La selección de la muestra será realizada de manera aleatoria, eligiendo un grupo de 10 a 15 desarrolladores de Medellín y Ocaña, o equipos de trabajo que estén involucrados en proyectos reales que utilicen Clean Architecture en el desarrollo de aplicaciones Android. Esta muestra permitirá obtener datos relevantes para evaluar el impacto de las optimizaciones propuestas y su efectividad en la modularidad, escalabilidad y mantenibilidad de las aplicaciones.

3.3 Diseño de instrumentos de recolección de la información y técnicas de análisis de datos

Dentro de las técnicas que se utilizarán en el presente proyecto se encuentran:

Entrevista

Se realizará una entrevista a desarrolladores de aplicaciones Android que trabajen con la arquitectura Clean Architecture y que están ubicados en Ocaña y Medellín. El objetivo es conocer cómo implementan esta arquitectura en sus proyectos, qué beneficios han observado en términos de modularidad, escalabilidad y mantenibilidad, y cuáles son los desafíos y dificultades que enfrentan durante su implementación. Además, se buscará obtener recomendaciones y buenas prácticas por parte de los desarrolladores, así como identificar las áreas en las que las optimizaciones propuestas podrían generar un impacto positivo. Esta información será clave para analizar el uso de la arquitectura Clean Architecture en el contexto de aplicaciones Android y evaluar posibles mejoras.

Encuesta

Se llevarán a cabo encuestas dirigidas a desarrolladores que hayan trabajado con Clean Architecture en el desarrollo de aplicaciones Android. Estas encuestas se aplicarán en dos momentos clave: antes de iniciar el proyecto, para identificar las principales problemáticas, experiencias, y percepciones sobre el uso de Clean Architecture en proyectos actuales; y después de implementar las optimizaciones propuestas, con el fin de evaluar la percepción de los participantes sobre las mejoras en modularidad, eficiencia y facilidad de mantenimiento. Además, se incluirán

preguntas específicas para obtener retroalimentación sobre la utilidad de las optimizaciones implementadas, su facilidad de adopción y el impacto en los tiempos de desarrollo y calidad del software.

3.4 Análisis de información

El análisis de la información se llevará a cabo mediante un enfoque estructurado para interpretar y evaluar los datos recogidos durante el proyecto. Para ello, se centralizarán las respuestas obtenidas en las entrevistas y cuestionarios utilizados con los desarrolladores participantes en el estudio. La información se organizará sistemáticamente para identificar patrones, tendencias y áreas de mejora relacionadas con la implementación de la arquitectura limpia en Android.

La primera fase del análisis consistirá en codificar y categorizar los datos cualitativos de las entrevistas, destacando las percepciones, opiniones y sugerencias de los participantes en relación con las optimizaciones propuestas. A continuación, los datos cuantitativos recogidos en las entrevistas se procesarán mediante herramientas estadísticas para evaluar indicadores clave como la eficacia, la comprensión y la satisfacción general con la arquitectura.

Estos resultados se analizarán en su conjunto para responder a las preguntas de la investigación, teniendo en cuenta los aspectos técnicos y la experiencia de los desarrolladores. Este enfoque permitirá extraer conclusiones adecuadas para validar la viabilidad y usabilidad de las optimizaciones realizadas, así como identificar áreas de oportunidad para futuras investigaciones o implementaciones.

3.5 Cronograma de actividades

Mes 1						mes 2				mes 3				mes 4			
Objetivo	Actividad	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1. Analizar la estructura de Clean Architecture en proyectos Android nativos, integrando componentes que tradicionalmente estarían separados, como los viewmodels, presenters, repositories y datasources, para reducir la complejidad y facilitar el desarrollo.	Revisar documentación y ejemplos de Clean Architecture en proyectos Android: Investigar artículos, repositorios de código abierto y documentación oficial que aborden la integración de Clean Architecture en Android, prestando atención a cómo se estructuran componentes como ViewModels, Presenters, Repositories y DataSources.	x															
	Mapeo de los componentes clave de la arquitectura: Identificar y documentar las responsabilidades principales de los		x														

	componentes mencionados (ViewModels, Presenters, Repositories y DataSources) y cómo interactúan entre sí dentro de la arquitectura.		
	Comparar implementaciones existentes de Clean Architecture: Evaluar proyectos de ejemplo en plataformas como GitHub o foros de desarrolladores, con el fin de observar diferentes enfoques de integración y extraer buenas prácticas.	x	
	Análisis crítico de la estructura propuesta: Identificar las posibles ventajas, desventajas y desafíos de integrar componentes tradicionalmente separados dentro de Clean Architecture, considerando aspectos como la escalabilidad, mantenibilidad y facilidad de implementación.	x	
2. Diseñar la estructura de Clean Architecture en proyectos Android nativos, integrando componentes	Diseñar un diagrama de arquitectura inicial: Crear un diagrama que represente cómo los ViewModels, Presenters, Repositories y DataSources se relacionarán		x

que tradicionalmente estarían separados, como los viewmodels, presenters, repositories y datasources, para reducir la complejidad y facilitar el desarrollo.	entre sí, asegurando una separación clara de responsabilidades y siguiendo los principios de Clean Architecture.		
	Definir interfaces y contratos entre las capas: Especificar las interfaces que conectarán cada capa (por ejemplo, entre Repositories y DataSources), estableciendo contratos que aseguren una comunicación coherente y desacoplada.	x	x
	Elaborar prototipos de componentes clave: Implementar prototipos iniciales de cada componente (ViewModels, Presenters, Repositories y DataSources) para validar la viabilidad del diseño propuesto antes de la implementación completa.	x	x
3. Aplicar los principios SOLID y	Implementar el patrón de presentación elegido (MVVM o		x

patrones de presentación como MVVM o MVP en la arquitectura, promoviendo la separación de responsabilidades y la inyección de dependencias, asegurando la independencia entre las diferentes capas de la aplicación.	MVP): Crear una estructura base en el proyecto Android que aplique el patrón de presentación seleccionado, asegurando que las vistas (Activity o Fragment) se mantengan libres de lógica empresarial.			
	Integrar principios de inyección de dependencias: Configurar un framework de inyección de dependencias (como Dagger o Hilt) para desacoplar las dependencias entre capas, facilitando la prueba y mantenimiento del código.	x		
	Refactorizar clases existentes para alinearse con los principios SOLID: Revisar y ajustar las clases del proyecto para cumplir con principios como la Responsabilidad Única y Inversión de Dependencias, asegurando que cada componente cumpla exclusivamente su función específica dentro de la arquitectura.	x		x
4. Validar la eficiencia y mantenibilidad del código	Realizar pruebas unitarias en cada capa de la arquitectura: Implementar pruebas unitarias para validar el correcto			x x

mediante el uso de Clean Architecture adaptada en un proyecto Android de ejemplo, realizando pruebas de integración, unitarias y de rendimiento, para asegurar que los cambios propuestos mejoren la calidad del software sin comprometer su funcionalidad o escalabilidad.	funcionamiento de los ViewModels, repositorios y otras clases clave dentro del proyecto, asegurando que cada componente opere de manera independiente y cumpla con su propósito.	
	Evaluar el rendimiento del proyecto mediante pruebas de estrés: Ejecutar pruebas de rendimiento para medir la velocidad de respuesta, uso de memoria y escalabilidad del sistema, identificando posibles cuellos de botella o áreas de mejora en la implementación de Clean Architecture.	x

Fuente. Autor del proyecto

4. Resultados

Durante todo el desarrollo de la aplicación móvil "InstaFlix", se realizó un análisis muy práctico y detallado de cómo se logra adaptar Clean Architecture a proyectos Android nativos. La estructura general del proyecto se dividió en múltiples módulos (aplicando modularización) organizados por capas que son claramente diferenciadas entre sí: app, data, domain, y usecases, lo cual está muy alineado con los principios de separación de responsabilidades y modularidad que se promueven en Clean Architecture.

4.1 Integración de componentes tradicionalmente separados

Uno de los resultados con más relevancia fue la integración de componentes que normalmente se encuentran en las capas aisladas, como lo son:

ViewModel / Presenter: Se ubicaron dentro del módulo de app, más específicamente en la carpeta UI, donde se manejan: la lógica de presentación como la interacción directa con los casos de uso.

UseCases: Se implementaron en un módulo aparte (usecases) permitiendo una independencia clara de lo que es: framework de presentación). Aquí se definieron los casos de uso, agrupados por el contexto (movie, tvshow, detail), esto facilitó el mantenimiento y la escalabilidad del código.

Repositories y DataSources: Estos dos fueron ubicados en el módulo data, y organizados en subpaquetes como common, datasource y repository. Realizar esto permitió mantener una fuente de datos totalmente centralizada y reutilizable, aplicando el principio de inversión de dependencias (los casos de uso dependen de interfaces, no de implementaciones concretas).

4.2 Reducción de la complejidad estructural total

La unificación y organización directa y jerárquica de todos los componentes logró reducir significativamente la complejidad en el código. Esto se traduce en:

Menor acoplamiento entre las diversas capas, dado que se está aplicando DI y las dependencias están invertidas y se inyectan a través de mecanismos como Koin o Hilt.

Reutilización de código: Un ejemplo de esto se logra ver con el componente PermissionChecker dentro del paquete “common” ya que puede ser utilizado desde diferentes casos de uso sin duplicación.

Facilidad para testing: Gracias a la estructura modular, fue posible incluir tests en los módulos test, testShared, e inclusive androidTest, permitiendo validar comportamiento tanto unitario como visual.

4.3 Evaluación práctica del impacto en el desarrollo

A lo largo del proyecto se logró evidenciar que esta estructura:

Aceleró el tiempo de desarrollo de nuevas funcionalidades.

Permitió la fácil incorporación de nuevas fuentes de datos externas (por ejemplo, cambiar un DataSource local a uno remoto) sin afectar la lógica de negocio ni la presentación.

Para el diseño de la arquitectura del proyecto "InstaFlix" se planteó siguiendo los principios fundamentales de Clean Architecture que fueron propuestos por Robert C. Martin. Se evaluó y optó por una estructura modular basada en la separación de responsabilidades y

dependencias dirigidas hacia el dominio, garantizando así la flexibilidad y mantenibilidad del código.

4.4 División por capas modulares

Se diseñó el proyecto para dividirlo en cuatro módulos principales, cada uno cumpliendo una responsabilidad muy específica dentro de la propia arquitectura:

app: Contiene la lógica de presentación, control de navegación y la orquestación de componentes. Aquí se encuentran los ViewModels, interfaces de usuario (ui) y el punto de entrada de la app (App.kt).

data: Que es el encargado de la gestión de los datos. Contiene implementaciones concretas de los repositories, datasources, servicios de red o locales y utilidades compartidas como PermissionChecker.

domain: Representa el “núcleo” o “negocio” de la aplicación. Aquí se definen las entidades del dominio y los diversos contratos (interfaces) que deben cumplir los repositorios. Esta capa no depende de ninguna otra.

usecases: Son alojados en un módulo independiente, permitiendo encapsular reglas de negocio específicas y reutilizables en diversos puntos de la aplicación. Fueron divididos por contexto (películas, series, detalles), facilitando el acceso a funciones como "Obtener lista de películas", "Buscar serie", etc.

4.5 Integración ordenada de componentes separados

Acá se presenta uno de los retos más comunes al diseñar Clean Architecture para

Android, es decidir dónde colocar elementos como ViewModels y Presenters, ya que Android no ofrece una guía específica sobre esto. En este diseño:

ViewModel se ubicó en la capa app, dentro del paquete ui, conectado con la vista a través de LiveData o Flow.

Presenters que como tal no se usaron en el sentido clásico del MVP, ya que la ViewModel cumplió ese rol, y esto se alinea con prácticas modernas de Jetpack.

Repositories y DataSources que se estructuraron dentro del módulo data, cumpliendo contratos definidos en domain, manteniendo así el principio de inversión de dependencias.

4.6 Beneficios derivados de este diseño estructurado

El diseño planificado de esta arquitectura trajo consigo como resultado beneficios muy tangibles en el desarrollo de la aplicación:

Facilidad para la incorporación de nuevas funcionalidades: por ejemplo, agregar un nuevo tipo de contenido (como documentales) solo requiere implementar nuevos casos de uso y agregar un repositorio especializado sin alterar el resto del sistema.

Escalabilidad: puesto que los módulos están separados, cada uno puede evolucionar de manera independiente. E inclusive podrían convertirse en librerías reutilizables en otros proyectos.

Mantenibilidad: el diseño modular permite identificar rápidamente qué parre de este sistema necesita atención ante un bug o requerimiento, reduciendo el tiempo de depuración.

Independencia del framework: gracias a mantener la lógica del negocio y los casos de uso por separado de Android, se podría reemplazar el frontend (por ejemplo, una app en

Compose o Flutter) sin modificar el núcleo del negocio.

Durante el desarrollo de la aplicación se hizo un especial énfasis en la aplicación correcta de los principios SOLID y de el uso de patrones de presentación como lo es MVVM, con el afán de lograr una arquitectura desacoplada, mantenible y escalable. La implementación de estos principios no fue directamente un aspecto superficial del proyecto, sino más bien una decisión central que afectó directamente la forma en la que se construyeron las distintas capas de la aplicación.

Uno de los primeros principios puestos en práctica fue el Single Responsibility Principle (SRP), visible tanto en la organización del código como en la forma en que se estructuraron los componentes. Cada clase, interfaz y módulo fue diseñado para cumplir una única responsabilidad bien definida. Por ejemplo, los UseCases tienen una única función bien definida (como obtener una lista de películas o buscar una serie por título), mientras que las clases encargadas de interactuar con las APIs externas o bases de datos se limitaron exclusivamente a eso mismo, delegando cualquier lógica adicional a otras capas. Esto permitió evitar clases "Dios", con múltiples propósitos, y facilitó la comprensión del sistema incluso para nuevos desarrolladores que se integren al proyecto en el futuro.

Respecto al Open/Closed Principle, se diseñaron las abstracciones que permitieron extender las funcionalidades sin modificar el código existente. Los repositorios, definidos como interfaces en la capa de dominio, son un claro ejemplo de esto: sus implementaciones concretas pueden modificarse o reemplazarse sin alterar directamente la lógica de negocio o los casos de

uso que dependen de ellas. Esta estructura fomenta la evolución del sistema sin comprometer su estabilidad. En escenarios como el cambio del origen de datos (de una API REST a una base de datos local, por ejemplo), simplemente funciona con implementar un nuevo DataSource e inyectarlo sin alterar el dominio.

El Liskov Substitution Principle fue considerado al lograr mantener siempre el contrato entre clases padre e hijas o interfaces e implementaciones. Los Repositories, por ejemplo, pueden ser utilizados por los casos de uso sin importar qué implementación específica se haya inyectado, lo que garantiza que el comportamiento del sistema no se vea alterado inesperadamente

De esa manera, el Interface Segregation Principle se abordó evitando interfaces monolíticas o lo que llaman “flat interfaces”. En lugar de tener una interfaz de repositorio con múltiples métodos no relacionados, se dividieron en interfaces más pequeñas, enfocadas en contextos específicos como MovieRepository, TvShowRepository o RegionRepository. Esto permitió que las implementaciones solo deban preocuparse por cumplir con los métodos que realmente necesitan, evitando acoplamientos innecesarios.

Con el Dependency Inversion Principle, se usó la inyección de dependencias mediante Dagger-Hilt, permitiendo que las clases de alto nivel (como los ViewModels) no conozcan las implementaciones concretas de sus dependencias, sino solo las abstracciones necesarias. Esta inversión de dependencias redujo considerablemente el acoplamiento entre capas y facilitó la realización de pruebas unitarias, al poder reemplazar fácilmente las dependencias reales por

mocks o fakes en ambientes controlados.

En lo que respecta al patrón de presentación, se optó por usar MVVM (Model-View-ViewModel) por su compatibilidad con las arquitecturas modernas de Android y su integración nativa con componentes como LiveData, StateFlow, y ViewModel de Jetpack. Esta elección permitió mantener las vistas (Activities, Fragments o Composables) completamente ajenas en lo que concierne a la lógica de negocio, delegando esta responsabilidad netamente a los ViewModels, quienes reciben los casos de uso como dependencias. La vista se limitó a observar el estado emitido por los ViewModels, reaccionando únicamente a cambios en dicho estado, lo que se tradujo en interfaces limpias, declarativas y con una lógica mínima. Gracias a esto, cada capa del patrón MVVM quedó claramente delimitada: el Model lo conformaron los casos de uso y el dominio, el ViewModel actuó como mediador de lógica y transformación de datos, y la View como receptor final de los datos para ser renderizados en la vista final.

De esa misma manera, se optimizó la escalabilidad del proyecto, permitiendo que nuevas funcionalidades sean añadidas sin que ello implique una refactorización de todo el sistema. Por último, la estructura permitió realizar pruebas automatizadas de forma más sencilla, ya que los componentes estaban desacoplados y fácilmente testeables de forma aislada. En conclusión, la correcta aplicación de SOLID y MVVM fue uno de los pilares que permitió construir una arquitectura robusta, profesional y alineada con las buenas prácticas de la ingeniería de software moderna.

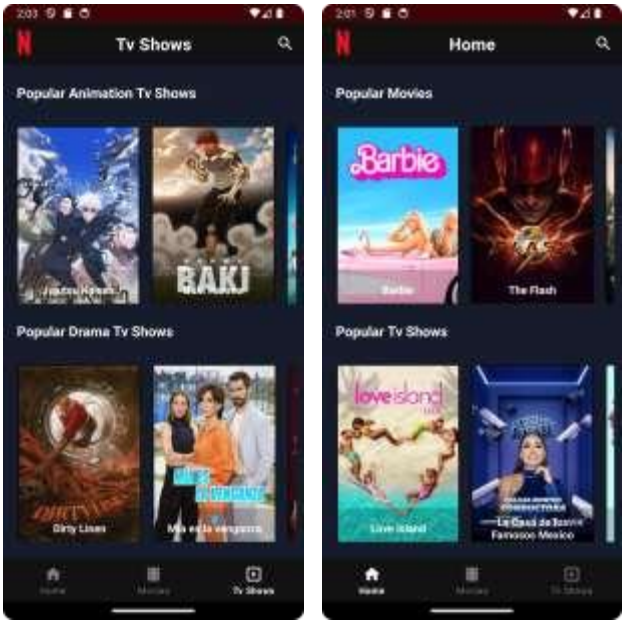
Con el fin de aclarar la naturaleza y el alcance de los resultados del proyecto

desarrollado, se incluye a continuación una breve, pero detallada descripción de la aplicación móvil realizada, así como las evidencias gráficas del proceso de su implementación y evolución de la arquitectura.

La aplicación desarrollada, se denomina InstaFlix, que es una plataforma móvil inspirada en servicios de streaming como Netflix, que permite visualizar información sobre películas y series populares del momento. Dentro de la app, el usuario puede navegar a entre las diferentes categorías como "Popular Movies" y "Popular TV Shows", accediendo a la carátula y título de cada uno

Figura 2

Instaflix App ScreenShots



Fuente: autor

La fuente de datos utilizada para alimentar la aplicación fue la API pública de The Movie

Database (TMDB) (Figura 1), lo que permitió obtener información actualizada de manera dinámica para cada sección de la app. TMDB ofrece una base de datos robusta de contenido audiovisual que se adaptó perfectamente a los requerimientos funcionales del proyecto.

Figura 3

TheMovieDb



Fuente: The Movie Database: https://www.themoviedb.org/?language=es

En cuanto a la planificación del desarrollo, el proyecto fue organizado en sprints utilizando principios de metodologías ágiles. A modo de ejemplo, el Sprint 1 se estructuró en varias fases que comprendieron desde la creación de la estructura de la arquitectura, hasta la implementación de las primeras vistas (Figura 2).

Figura 4

Sprint Table ScreenShot

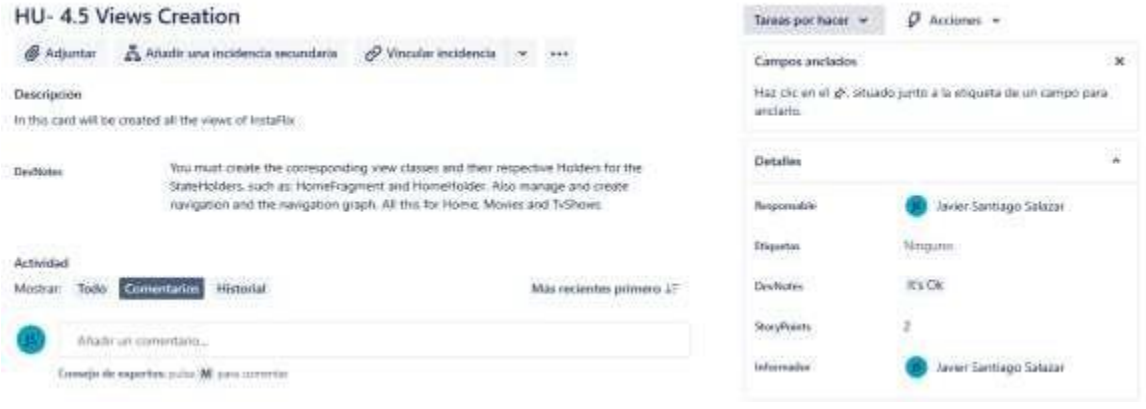
SPRINT 1		
PHASE (DAY)	**UH**	**StoryPoints (SP)**
1	OnBoarding / Welcome	1
2	Architecture Structure	1
3	Models and Data	1
4 - 5	Views creation	2

Fuente: Autor

De igual forma, cada funcionalidad fue desglosada en Historias de Usuario (HU). La HU 4.5, correspondiente a la creación de las vistas principales, muestra cómo se definieron claramente las tareas, responsables y los Story Points asociados (Figura 4).

Figura 5

User History ScreenShot



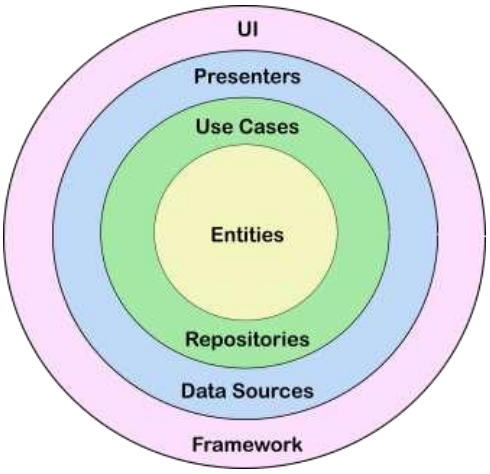
Fuente: Autor + Jira

Con respecto a la evolución de la arquitectura, el proyecto partió de una estructura

tradicional de Clean Architecture, pero con unas adaptaciones específicas para el entorno Android nativo. El primer paso consistió en fusionar los Repositories con los DataSources, buscando una reducción de capas innecesarias y una mayor eficiencia en la gestión de datos

Figura 6

Clean Architecture Desing Model with changes

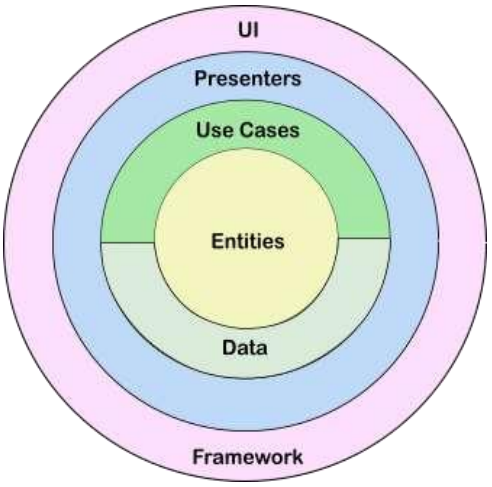


Fuente: Autor

Posteriormente, se realizó una segunda optimización importante que fue al fusionar la capa de Presenters con la UI y el Framework, eliminando las redundancias y adecuando el flujo a los componentes nativos de Android como ViewModels y Fragments (Figura 6).

Figura 7

Clean Architecture Desing Model with changes pt.2

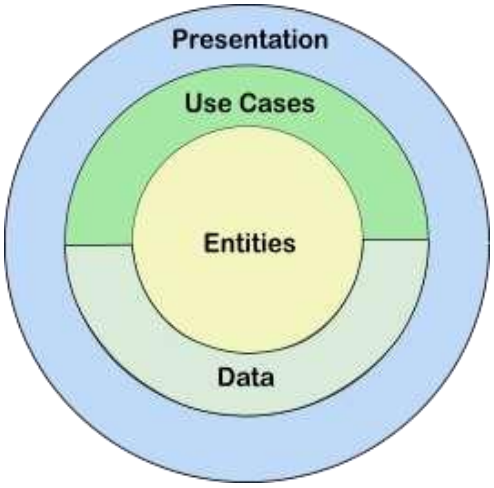


Fuente: Autor

Finalmente, se logró consolidar una versión optimizada de la Clean Architecture, pero esta vez adaptada a las necesidades de desarrollo móvil en Android. Esta versión mantiene el núcleo de la arquitectura limpia (Entities, UseCases, Data), pero simplifica mucho la interacción con el framework, enfocándose en minimizar dependencias y maximizar la claridad del flujo de datos (Figura 7).

Figura 8

Clean Architecture Desing Model with changes pt.3



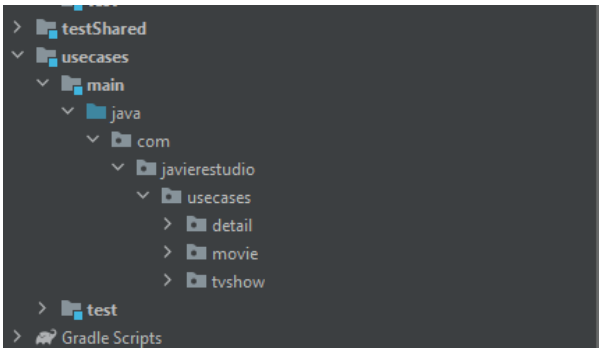
Fuete: Autor

Adicionalmente, la organización modular del proyecto puede observarse en la estructura de carpetas del repositorio de código (Figura 8), evidenciando una separación bastante clara entre app, domain, data y usecases.

Figura 9

Project Structure ScreenShots





Fuente: Autor

En conclusión, la aplicación InstaFlix constituye un ejemplo muy práctico del impacto positivo que tiene el aplicar una arquitectura bien definida desde el inicio del desarrollo, y no solo en términos meramente técnicos, sino también en cuanto a la organización, planificación y ejecución eficiente del proyecto.

Los resultados que se obtuvieron en el desarrollo de este proyecto han permitido que se valide la efectividad de la adaptación de Clean Architecture en el desarrollo de aplicaciones Android nativas. A través del análisis y la implementación de las optimizaciones que se propusieron, se han logrado alcanzar unas mejoras significativas en términos de modularidad, escalabilidad y mantenibilidad del código.

Optimización de la Modularidad: Se logró una división clara entre las diversas responsabilidades de las capas de la arquitectura, lo cual, permitió una mejor organización del código y redujo la dependencia entre los módulos. Esto facilitó la integración de nuevos componentes o features sin afectar la estructura existente.

Escalabilidad Mejorada: La implementación de patrones de presentación como lo son: MVVM y el uso de los principios SOLID permitieron que la arquitectura fuese más flexible y que sea capaz de adaptarse a futuras ampliaciones del proyecto sin comprometer el rendimiento o la estabilidad del mismo.

Mantenibilidad y Reducción de la Complejidad: Gracias a la separación por capas y a la inyección de dependencias, los desarrolladores encontraron más sencillo entender y modificar el código. Esto ayudó a disminuir la curva de aprendizaje para los nuevos integrantes del equipo y también optimizó el tiempo de desarrollo.

Validación mediante Pruebas: Se llevaron a cabo pruebas unitarias e integración en la aplicación de ejemplo InstaFlix, logrando confirmar que la adaptación de Clean Architecture permite reducir errores y mejorar la estabilidad general del sistema.

Percepción de Desarrolladores Mediante encuestas y entrevistas a desarrolladores Android, se evidenció una mejor comprensión del modelo arquitectónico y una mayor aceptación de su aplicabilidad en los proyectos reales.

Recalcando que, la adaptación de Clean Architecture a proyectos Android nativos demostró significativamente ser una estrategia efectiva para optimizar la calidad del software, permitiendo así que los desarrolladores trabajen con mayor eficiencia y asegurando que las aplicaciones sean más robustas, escalables, sostenibles en el tiempo y setteando un marco directo de trabajo en un proyecto productivo.

5. Conclusiones

La investigación realizada permitió lograr una evaluación de manera crítica a la aplicabilidad de Clean Architecture en proyectos productivos de Android nativo, identificando las estrategias y las mejores prácticas para optimizar la modularidad, escalabilidad y mantenibilidad del software. Directamente a través de la revisión de literatura y la experimentación con la implementación del modelo, se logró establecer un marco sólido que ayuda a respaldar el desarrollo eficiente de aplicaciones móviles.

El enfoque metodológico que se adoptó en el proyecto permitió no solo analizar la arquitectura desde un punto de vista teórico, sino también ayudó a aplicar los conceptos en un caso práctico, validando la efectividad a través de pruebas y evaluaciones de rendimiento. Se determinó que la correcta separación de las capas, el uso de los principios SOLID y la integración de patrones de presentación como MVVM resultan fundamentales para lograr mejorar la calidad del código y la facilidad de mantenimiento en proyectos de mediana y gran escala.

El análisis de resultados logró demostrar que tener una implementación estructurada de Clean Architecture facilita la incorporación de nuevos desarrolladores al equipo, reduce la complejidad del software y mejora la organización general del código en todo sentido. De la misma manera, la adopción de metodologías ágiles y buenas prácticas de desarrollo en general, refuerza la capacidad del sistema para adaptarse a cambios y nuevas funcionalidades sin comprometer su estabilidad.

Esta investigación abre la puerta a futuras exploraciones sobre la aplicabilidad de Clean Architecture u otras arquitecturas derivadas, en otros entornos de desarrollo, como aplicaciones multiplataforma y sistemas empresariales. Adicionalmente, el estudio de integración con tecnologías emergentes en el sector, como la inteligencia artificial y la computación en la nube, lo cual podría proporcionar nuevas oportunidades para mejorar la eficiencia y el rendimiento de estas aplicaciones.

Finiquitando, la documentación y difusión de estos hallazgos en la comunidad de desarrolladores contribuirá va a fomentar el uso de arquitecturas escalables y mantenibles en el desarrollo de software, así como promover estándares de calidad y buenas prácticas en esta industria. Además, este artículo fue presentado en la edición 17 del Congreso Colombiano de Computación (17CCC), realizado en Medellín entre los días 10 y 11 de agosto de 2023. Este evento, organizado por la Sociedad Colombiana de Computación, RENATA y la Universidad EAFIT, logra permitir la difusión de los hallazgos ante una audiencia nacional e internacional. La presentación se llevó a cabo en la sesión "Software Engineering & Human Machine Interaction" y permitió el intercambio de ideas con investigadores y profesionales del sector. La participación en eventos de este tipo tiene un carácter fundamental para fortalecer la investigación, recibir retroalimentación y fomentar el desarrollo de soluciones innovadoras en la computación y la ingeniería de software.

Además de los logros directamente mencionados, es de suma importancia destacar que este proceso de implementación permitió adquirir un conocimiento más profundo sobre la

interrelación entre los diferentes componentes arquitectónicos en Android Nativo. Se evidenció el cómo una arquitectura bien estructurada no solo mejora la calidad técnica del software, sino que también incide directamente en aspectos organizacionales y colaborativos tales como la colaboración entre miembros un equipo, la gestión de deuda técnica y la estandarización de los procesos de desarrollo. La claridad de las responsabilidades entre las capas y los módulos redujo el tiempo necesario para comprender y modificar directamente el código, y facilitó la integración continua y la automatización de las pruebas.

De la misma manera, se observó un cambio muy positivo en la forma de abordar los requerimientos funcionales y los no funcionales. La arquitectura limpia incentivó un enfoque más disciplinado frente a la planificación de las nuevas funcionalidades, priorizando primero la definición del dominio (negocio), luego los casos de uso, y al final el desarrollo de interfaces. Este flujo se contrapone a enfoques más impulsivos o desorganizados donde se comienza directamente desde la vista o la interfaz gráfica, por lo cual suele derivar en un sistema muy acoplado y difícil de mantener.

Desde otro lado, el uso de principios SOLID logra promover una cultura de desarrollo orientada a la calidad del código, donde el mismo no solo busca "funcionar", sino más bien hacerlo de manera correcta, extensible y sostenible en el tiempo. Lo cual convirtió en una práctica habitual, por ejemplo, el uso de la inyección de dependencias como mecanismo central para desacoplar los componentes, lo que se tradujo también en una mayor facilidad para realizar pruebas unitarias y pruebas de integración. Esto también, contribuyó a una mayor confianza en cada entrega del producto, dado que los errores eran detectados de forma más temprana.

También es muy relevante señalar que, a pesar de los beneficios ya mencionados, la aplicación de Clean Architecture no estuvo exenta de desafíos. Uno de los más notorios fue la curva de aprendizaje inicial, esto pasa con muchas arquitecturas y más especialmente para desarrolladores con experiencia limitada en arquitectura de software o sin un conocimiento sólido en patrones de presentación como MVVM, principios SOLID o gestión de las dependencias. Este obstáculo se mitigó con sesiones de formación internas, documentación, y revisión de código, lo cual no solo resolvió el problema, sino que fortaleció el aprendizaje.

Otro desafío identificado fue el gran incremento del número de archivos y capas en comparación con arquitecturas más tradicionales o monolíticas. Aunque esto puede generar inicialmente la percepción de mayor complejidad, con el tiempo se logra demostrar que esta granularidad permite una navegación más predecible y lógica dentro del mismo proyecto, reduciendo el tiempo de búsqueda y la comprensión de funcionalidades específicas. La clave estuvo en usar una convención de nombres clara y una estructura de carpetas coherente con el diseño de capas y dominios definidos.

En términos de mantenimiento, la estructura del sistema claramente facilitó la identificación de bugs y la refactorización de módulos individuales, sin temor a afectar otras partes del código, lo cual es muy útil. Esto es especialmente importante en sistemas en crecimiento, donde las funcionalidades evolucionan constantemente. Aparte, la implementación permitió preparar el sistema para soportar cambios tecnológicos futuros, como lo que son las migraciones a otras versiones de Android, cambios de las librerías o incluso la integración con arquitecturas de microservicios y APIs de terceros.

Finalmente, el hecho de haber presentado este trabajo en el 17 Congreso Colombiano de Computación permitió validar el enfoque adoptado ante una audiencia académica y profesional diversa. Las preguntas recibidas, los comentarios y todas las discusiones surgidas durante el evento ofrecieron una oportunidad invaluable para repensar aspectos clave del diseño, considerar perspectivas distintas, y reafirmar la pertinencia de estudiar la arquitectura del software como una disciplina central en el desarrollo de productos de calidad. La retroalimentación obtenida logra reafirmar que este tipo de trabajos no solo tienen impacto en el código, sino también en la cultura de desarrollo y software en general, en la profesionalización del sector y en la formación de futuros ingenieros de software.

6. Recomendaciones

Se recomienda a la Universidad Francisco de Paula Santander fomentar y promover el uso de arquitecturas de software modernas y escalables en sus programas de Ingeniería de Sistemas, mejorando e incentivando el estudio y aplicación de Clean Architecture en el desarrollo de software mobile y otras áreas tecnológicas.

Es fundamental que los docentes incentiven a los estudiantes a investigar sobre nuevas metodologías de diseño y desarrollo de software, ofreciendo la mentoría necesaria el y acompañamiento en proyectos que impliquen la aplicación de arquitecturas avanzadas y buenas prácticas de programación.

Se recomienda también la integración del estudio de Arquitecturas en el currículo académico, permitiendo a los estudiantes adquirir experiencia práctica en la implementación de arquitecturas escalables y mantenibles, asegurando así su preparación para enfrentar los retos del mundo laboral como posibles Arquitectos de Software.

Por otra parte, se sugiere continuar con esta línea de investigación para evaluar la aplicabilidad de Clean Architecture en otros entornos de desarrollo, como aplicaciones multiplataforma y algunas soluciones empresariales, con el fin de lograr determinar su eficacia en distintos contextos y escalas de proyectos.

Finalmente, se recomienda a los desarrolladores Android adoptar buenas prácticas de arquitectura desde las primeras etapasde un proyecto, asegurando la mantenibilidad y escalabilidad del software a largo plazo, y así fomentando el uso de metodologías ágiles para

mejorar la eficiencia del desarrollo.

7. Referencias

Adwinda, C., & Suryodiningrat, D. (2020). Developing an Android-based running application. Journal of Critical Reviews, 7(8). https://doi.org/10.31838/jcr.07.08.182

Arango, E. C., & Loaiza, O. L. (2021). SCRUM Framework Extended with Clean Architecture Practices for Software Maintainability. En Lecture Notes in Networks and Systems (pp. 667–681). https://doi.org/10.1007/978-3-030-77442-4_56

Ardito, L., Coppola, R., Malnati, G., & Torchiano, M. (2020). Effectiveness of Kotlin vs. Java in Android app development tasks. Information And Software Technology, 127, 106374. https://doi.org/10.1016/j.infsof.2020.106374

Boukhary, S., & Colmenares, E. (2019). A Clean Approach to Flutter Development through the Flutter Clean Architecture Package. 2021 International Conference On Computational Science And Computational Intelligence (CSCI). https://doi.org/10.1109/csci49370.2019.00211

Bukovcan, M., Blazevic, D., Nenadic, K., & Stevic, M. (2022). Clean Architecture of Client-Side Software Development for Smart Furniture Control. 2022 11th Mediterranean Conference On Embedded Computing (MECO), 1–4. https://doi.org/10.1109/meco55406.2022.9797122

Callan, J., & Petke, J. (2024). Multi-objective improvement of Android applications. Automated Software Engineering, 32(1). https://doi.org/10.1007/s10515-024-00472-7

Chen, Z. (2024). Studying privacy leaks in Android app logs. Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024), 692–694. Association for Computing Machinery. https://doi.org/10.1145/3663529.3664461

Coppola, R., Fulcini, T., & Torchiano, M. (2024). Poster: Kotlin Assimilating the Android Ecosystem – An Appraisal of Diffusion and Impact on Maintainability. Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), 266–267. https://doi.org/10.1145/3639478.3643071

De Dieu, M. J., Liang, P., Shahin, M., & Khan, A. A. (2023). Characterizing architecture related posts and their usefulness in Stack Overflow. Journal Of Systems And Software, 198, 111608. https://doi.org/10.1016/j.jss.2023.111608

Em, S. (2024). Exploring Experimental Research: Methodologies, Designs, and Applications Across Disciplines. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.4801767

Feng, Q., Ji, H., Ma, X., & Liang, P. (2024). Cross-language dependencies: An empirical study of Kotlin-Java. Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), 189–199. https://doi.org/10.1145/3674805.3686680

Feng, X., & Periyasamy, K. (2022). A Cost Estimation Model for Scrum Projects. Journal of Computing Sciences in Colleges, 38(4), 30–37. https://doi.org/10.5555/3580619.3580624

Ferreira, V. B. S., Ferreira, C. A., & Grande, E. T. G. (2022). Estado da arte da pesquisa em: Clean Architecture e princípios de SOLID. Research Society And Development, 11(16), e335111637198. https://doi.org/10.33448/rsd-v11i16.37198

Gao, S., Liu, L., Liu, Y., Liu, H., & Wang, Y. (2020). API recommendation for the development of Android App features based on the knowledge mined from App stores. Science Of Computer Programming, 202, 102556. https://doi.org/10.1016/j.scico.2020.102556

Ji, K. (2024). A study of accessibility improvement on Android IoT system. Proceedings of the 2024 4th International Conference on Internet of Things and Machine Learning (IoTML '24), 80–84. https://doi.org/10.1145/3697467.3697605

Kouraklis, J. (2016). MVVM as Design Pattern. En Apress eBooks (pp. 1–12). https://doi.org/10.1007/978-1-4842-2214-0_1

Knight, K. L. (2010). Study/Experimental/Research design: much more than statistics. Journal Of Athletic Training, 45(1), 98–100. https://doi.org/10.4085/1062-6050-45.1.98

Lano, K., & Tehrani, S. Y. (2023). Introduction to Clean Architecture Concepts. Undergraduate Topics In Computer Science, 35–49. https://doi.org/10.1007/978-3-031-44143-1_2

Lawong, D. A., & Akanfe, O. (2024). Overcoming team challenges in project management: The scrum framework. Organizational Dynamics, 101073. https://doi.org/10.1016/j.orgdyn.2024.101073

Mateus, B. G., & Martinez, M. (2020). On the adoption, usage and evolution of Kotlin features in Android development. Proceedings of the 14th ACM/IEEE International Symposium

on Empirical Software Engineering and Measurement (ESEM '20), Article 15, 1–12. https://doi.org/10.1145/3382494.3410676

Moon, E., & Howison, J. (2024). A dynamic perspective on software modularity in open source software (OSS) development: A configurational approach. Information And Organization, 34(1), 100499. https://doi.org/10.1016/j.infoandorg.2023.100499

Neto, A. S., Ramos, F., Albuquerque, D., Dantas, E., Perkusich, M., Almeida, H., & Perkusich, A. (2023). Towards a Recommender System-based Process for Managing Risks in Scrum Projects. Proceedings Of The 37th ACM/SIGAPP Symposium On Applied Computing, 1051–1059. https://doi.org/10.1145/3555776.3577748

Oliveira, H., & Araújo, C. (2021). An agile learning management method based on Scrum. En ACM Conference On Innovation And Technology In Computer Science Education. https://doi.org/10.1145/3408877.3439650

Rachovski, T., Hadzhikoleva, S., Hadzhikolev, E., & Lengerov, A. (2022). Using Clean Architecture Principles to Improve the Design and Implementation of the Mobiles Online Platform. En Lecture Notes in Networks and Systems (pp. 11–19). https://doi.org/10.1007/978-981-16-7657-4_2

Rubin, J., Henniche, A. N., Moha, N., Bouguessa, M., & Bousbia, N. (2019). Sniffing Android code smells: An association rules mining-based approach. Proceedings of the 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft '19), 123–127. https://doi.org/10.5555/3340730.3340753

Sheikh, W., & Sheikh, N. (2020). A Model-View-ViewModel (MVVM) Application Framework for Hearing Impairment Diagnosis - Class Inheritance Architecture. 2022 Intermountain Engineering, Technology And Computing (IETC), 1–6. https://doi.org/10.1109/ietc47856.2020.9249215

Sullivan, K. J., Griswold, W. G., Cai, Y., & Hallen, B. (2001). The structure and value of modularity in software design. ACM SIGSOFT Software Engineering Notes, 26(5), 99–108. https://doi.org/10.1145/503271.503224

Verwijs, C., & Russo, D. (2022). A Theory of Scrum Team Effectiveness. ACM Transactions On Software Engineering And Methodology, 32(3), 1–51. https://doi.org/10.1145/3571849

Apéndice

En este apéndice se presentan los certificados y evidencia obtenida por participación en eventos y publicación de artículo, estos hechos son relevantes para el proceso de investigación.

Estos certificados demuestran la capacitación y el conocimiento adquirido que contribuyen a la validez y solidez del estudio.

Publicación en la revista del artículo de investigación realizado



Nota. Imagen que valida la publicación del articulo realizado “Clean Architecture: Impact on Performance and Maintainability of Native Android Projects“ en ResearchGate.

Enlace: https://link.springer.com/chapter/10.1007/978-3-031-47372-2_8