UNIVERSIDAD
REY JUAN CARLOS

GRADO EN INGENIERÍA INFORMÁTICA

Curso Académico 2024/2025

Trabajo Fin de Grado

APLICACIÓN NATIVA ANDROID PARA FACILITAR AL USUARIO LA GESTIÓN DE LA COMPRA DE ALIMENTOS DEL HOGAR, RECETAS E INVITADOS.

Autor : Beatriz Esteban Alcántara

Tutor : Manuel Rubio Sanchéz

Trabajo Fin de Grado/Máster

Aplicación nativa Android para facilitar al usuario la gestión de la compra de alimentos del hogar, recetas e invitados

Autor : Beatriz Esteban Alcántara

Tutor : Manuel Rubio Sanchéz

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 20XX, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Mostolés, a de de 20XX

Dedicado a

mi familia, mis amigos y mi pareja

II

Agradecimientos

Aquí vienen los agradecimientos... Aunque esté bien acordarse de la pareja, no hay que olvidarse de dar las gracias a tu madre, que aunque a veces no lo parezca disfrutará tanto de tus logros como tú... Además, la pareja quizás no sea para siempre, pero tu madre sí.

Resumen

Este Trabajo de Fin de Grado presenta el desarrollo de **Pinche**, una aplicación móvil nativa Android cuyo objetivo es facilitar la planificación y organización de la compra doméstica. Se ofrece una visión completa del proceso de desarrollo desde la perspectiva de distintos roles: equipo de UX (Experiencia de usuario), producto y técnico, centrándonos especialemnte en este último. El objetivo de entender este proceso es poner en relieve la importancia de que cada pieza que conforma un equipo digital entienda los requisitos del producto y el razonamiento que hay detrás de cada uno de ellos a la hora de implementarlo.

El proyecto se ha desarrollado siguiendo una arquitectura basada en el patrón MVVM, utilizando tecnologías modernas como Kotlin (lenguaje nativo de Android), Jetpack Compose, Hilt y Firebase (Firestore y Authentication) y aplicando buenas práticas de diseño. También se ha seguido una estrategia de testeo completa que incluye pruebas unitarias, de interfaz, integración, E2E y cobertura de código. El diseño de la interfaz se ha prototipado en Figma, y la gestión del proyecto se ha organizado en Trello.

El trabajo se enmarca en el contexto del desarrollo de software actual y estilo de vida social. Poniendo en relieve cómo las aplicaciones móviles pueden impactar en la mejora de la calidad de vida de las personas, optimizando tareas de su día a día, y la importancia de colocar al usuario en el centro del proceso de su desarrollo.

Índice general

1. Introducción y Motivación **1**

1.1. Mercado actual de las aplicaciones móviles 1

1.2. Tipos de aplicaciones móviles 4

1.3. Análisis del entorno 5

2. Objetivos **7**

3. Tecnologías, Herramientas y Metodologías **9**

3.1. Lenguaje de programación 9

3.2. Frameworks y librerías 10

3.2.1. Jetpack Compose 10

3.2.2. Jetpack Navigation 11

3.2.3. Hilt 11

3.3. Firebase 11

3.4. Entorno de desarrollo 12

3.5. Control de versiones 12

3.6. Metodologías de desarrollo 13

3.6.1. Design Thinking 13

3.6.2. Lean Startup 13

3.6.3. Agile 14

3.7. Diseño de la interfaz 17

3.8. Herramientas de testeo 17

3.8.1. Test unitarios 17

3.8.2. Test de interfaz de usuario (UI) 17

3.8.3. Test de integración 18

4. Descripción informática **19**

4.1. Requisitos 19

4.1.1. Requisitos funcionales 19

4.1.2. Requisitos no funcionales 25

4.2. Arquitectura y análisis 25

4.2.1. Principios generales 26

4.2.2. Capa de UI 27

4.2.3. Capa de dominio 28

4.2.4. Capa de datos 29

4.2.5. Gestión de dependencias con Hilt 29

4.2.6. Ventajas de esta arquitectura 30

4.3. Diseño e implementación 30

4.3.1. Análisis de requerimientos y estructura funcional 30

4.3.2. Decisiones técnicas relevantes HABLAR DE Jetpack Navigation SOLO? aqui o donde? Y SU INTEGRACION CON HILT NAVIGATION 31

4.4. Pruebas 31

4.4.1. Pruebas unitarias 32

4.4.2. Pruebas de interfaz (UI) 33

4.4.3. Pruebas de integración 34

4.4.4. Pruebas end-to-end (E2E) 34

4.4.5. Cobertura de código 34

5. Conclusiones y trabajos futuros **35**

Bibliografía **37**

A. Manual de usuario **39**

Índice de figuras

1.1. Aplicaciones gratuitas y de pago, Android vs iOS. 2
1.2. Apps por categoría Google Play 2
1.3. Apps por categoría App Store 2
1.4. Nuevas aplicaciones por mes en cada tienda de aplicaciones (27/01/2024) . . . 3
1.5. Mapa mundial de Android e iOS (27/01/2024) 4

3.1. Esquema de funcionamiento de Scrum. 16

4.1. Historia de usuario en Trello 24
4.2. Estado del tablero Kanban al final del primer Sprint 24
4.3. Diagrama arquitectura MVVM. 26
4.4. Arquitectura de la aplicación Pinche. 27
4.5. Componente Compose `RecoverPasswordForm` de Pinche 28
4.6. ViewModel `RecoverPasswordForm` de Pinche 28
4.7. Caso de uso `SendPasswordResetEmailUseCase` de Pinche 28
4.8. Función `sendPasswordResetEmail` en el repositorio de autenticación . . 29

Capítulo 1

Introducción y Motivación

1.1. Mercado actual de las aplicaciones móviles

Las aplicaciones móviles han transformado por completo nuestro día a día y tienen un gran impacto en cómo nos comunicamos, cómo trabajamos, cómo nos divertimos y cómo realizamos nuestras tareas.

Entre esas tareas se encuentra realizar la compra del hogar, tarea que requiere de organización si el usuario quiere realizarla de manera óptima. Este trabajo de fin de grado se enfoca en generar una herramienta que sirva de ayuda para que los usuarios la lleven a cabo. A partir de este momento, la aplicación que se desarrolla a lo largo de este proyecto la llamaremos Pinche.

Según los datos de 42matters, compañía que se encarga de recopilar y ofrecer datos a diferentes empresas, a 27 de enero de 2024 hay 2 millones aplicaciones Android en Google Play y algo más de 1.900.000 aplicaciones iOS en App Store, Figura 1.1. De las cuales, el porcentaje de aplicaciones que los usuarios se pueden descargar de manera gratuita es aproximadamente de 95 % en ambas tiendas de aplicaciones.

Figura 1.1: Aplicaciones gratuitas y de pago, Android vs iOS.

Fuente: `https://42matters.com/stats#available-apps-count`

En cuanto a categorías, Figura 1.2 y Figura 1.3, la educación lidera en Google Play, mientras que los juegos son predominantes en App Store. Pinche formaría parte de la categoría herramientas, la cual aparece en tercer lugar en ambos casos (Tools en Play Store y Utilities en App Store).

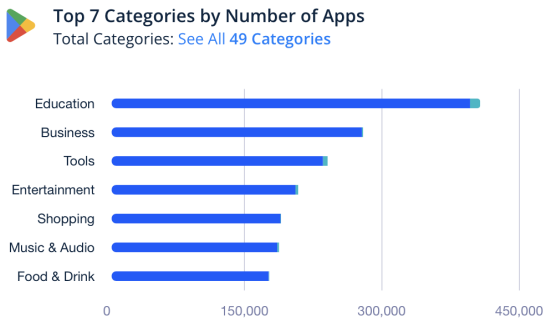

Figura 1.2: Apps por categoría Google Play

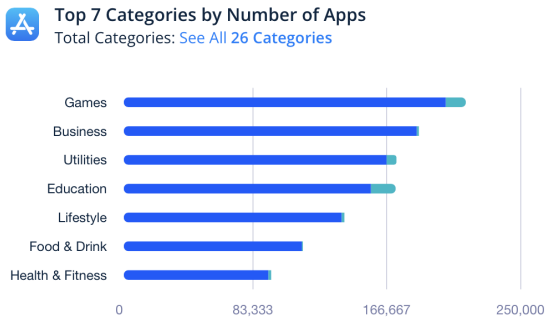

Figura 1.3: Apps por categoría App Store

Fuente: `https://42matters.com/stats#apps-by-category`

Además, se lanzan diariamente más de 2.300 nuevas aplicaciones en Google Play y unas 1.100 en App Store. Esto representa más de 90.000 aplicaciones nuevas al mes en la platafor-

ma de Android, lo que pone de manifiesto la alta competencia y dinamismo de este mercado, Figura 1.4.

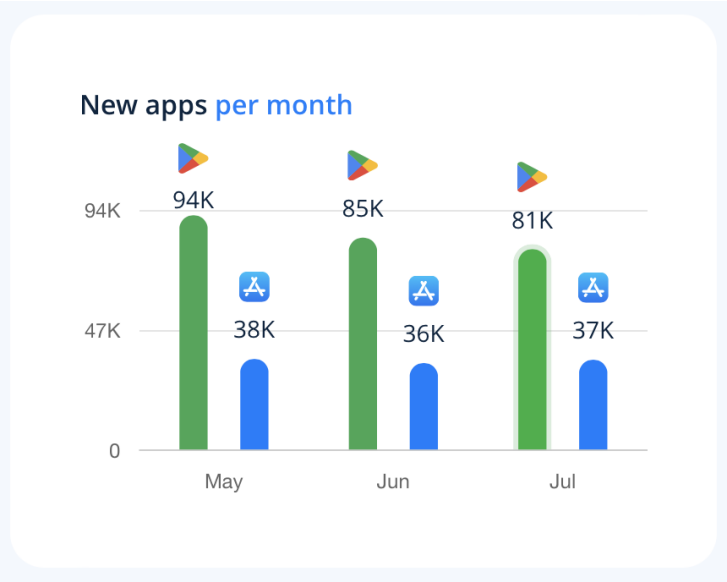

Figura 1.4: Nuevas aplicaciones por mes en cada tienda de aplicaciones (27/01/2024)

Fuente: `https://42matters.com/stats#apps-by-category`

En cuanto a los sistemas operativos más utilizados, como se puede ver en la Figura 1.5, Android domina el panorama global (no tiene en cuenta dispositivos tablets). Según Statista (junio de 2024), su cuota de mercado alcanza el 72,15 %, frente al 27,19 % de iOS. Aunque en países como Estados Unidos e Irlanda iOS tiene más presencia, Android es el sistema operativo predominante en regiones como América Latina, África, Asia y, especialmente, España.

Figura 1.5: Mapa mundial de Android e iOS (27/01/2024)

Fuente: `https://www.statista.com`

En el caso concreto de España, que es donde se publicaría esta aplicación, el 77 % de los smartphones son Android frente a un 23 % de iOS (Statista, diciembre 2023). Este dato resulta decisivo a la hora de seleccionar la plataforma de desarrollo, ya que permite orientar el producto a la mayoría de los usuarios potenciales.

1.2. Tipos de aplicaciones móviles

Se distinguen tres tipos principales: aplicaciones nativas, aplicaciones web y aplicaciones híbridas.

Una aplicación nativa es aquella que se crea específicamente para un sistema operativo móvil, utilizando su lenguaje y herramientas oficiales. Esto permite aprovechar al máximo las capacidades del hardware del dispositivo, lo que se traduce en un mayor rendimiento y más fluidez en la experiencia de usuario. Por ejemplo, se considera nativa una aplicación desarrollada en Kotlin para Android o en Swift para iOS. Su distribucción se realiza a través de su instalación desde plataformas oficiales como Google Play o App Store y, en caso de querer abarcar varios

sistemas operativos, implica desarrollar una versión distinta para cada uno.

Las aplicaciones web, por el contrario, son páginas web optimizadas para dispositivos móviles, a las que se accede desde el navegador sin necesidad de instalación. Se desarrollan con tecnologías como HTML, CSS y JavaScript, y su principal ventaja es la portabilidad entre sistemas. Sin embargo, su rendimiento es inferior al de una aplicación nativa y tienen acceso limitado a las funcionalidades del dispositivo.

Las aplicaciones híbridas combinan elementos de ambas. Básicamente, se trata de aplicaciones web que se ejecutan dentro de un contenedor nativo, lo que permite distribuirlas desde tiendas oficiales y acceder a algunas funcionalidades del hardware. No obstante, la integración entre la parte web y la parte nativa puede ser compleja, sobre todo en términos de rendimiento y mantenimiento.

Tras analizar las características de cada enfoque, para este trabajo de fin de grado se optó por desarrollar una aplicación nativa Android. Esta decisión responde tanto al interés por aprender en profundidad el desarrollo específico para esta plataforma, como al deseo de ofrecer una experiencia más fluida, potente y adaptada a los dispositivos que predominan en el mercado español.

1.3. Análisis del entorno

Antes de iniciar el desarrollo de la aplicación, se llevó a cabo un análisis del entorno para identificar qué soluciones digitales existen.

Una de las aplicaciones más instaladas para la gestión de listas de la compra es *Bring!*. Su funcionamiento se basa en la creación de listas de la compra mediante una interfaz visual con iconos organizados por categorías (frutas, lácteos, etc) y permite que varios usuarios participen en una lista. Sin embargo, no permite especificar cantidades con precisión, ni se integra con recetas o planificación semanal, lo que limita su utilidad para quienes desean gestionar la compra de forma más detallada.

En número de instalaciones le sigue de cerca *Listonic*, con un diseño más clásico: listas de verificación con productos agrupados por su sección en el supermercado. Es rápida y práctica. No obstante, su interfaz resulta más básica y tampoco tiene en cuenta la planificación de menús, recetas o la personalización de listas según hábitos domésticos o eventos.

A diferencia de otras soluciones del mercado, Pinche aborda el problema de la organización de la compra doméstica desde una perspectiva práctica, personalizada y centrada en la realidad cotidiana de quienes gestionan los menús y las compras del hogar.

La aplicación se estructura en tres secciones: listas de la compra, recetas e invitados. En la sección de listas, el usuario puede crear múltiples listas adaptadas a distintas ocasiones —por ejemplo, "Lista semanal" o "Cumpleaños de María"— y añadir productos con su cantidad, e incluso el supermercado donde se prefiere comprarlos. En la sección de recetas, se puede incluir su elaboración detallada e ingredientes para un número determinado de comensales, además de añadir fácilmente esos ingredientes a cualquier lista activa si desea realizar la receta. Por último, en la sección de invitados, se puede registrar información personalizada sobre las personas que vienen a comer a casa, como sus intolerancias, preferencias y qué platos se les han preparado recientemente.

En definitiva, ofrece una solución que permite la planificación, optimizar el tiempo y evitar errores comunes como compras duplicadas, olvidos o preparación de menús inadecuados para los invitados.

Desde el punto de vista académico, el proyecto ofrece un caso completo para aplicar competencias clave del grado: desarrollo de interfaces modernas con Jetpack Compose, gestión de datos con Firebase, diseño orientado al usuario y trabajo bajo metodologías ágiles.

Capítulo 2

Objetivos

El objetivo principal de este proyecto es comprender y aplicar todo el proceso que conlleva la creación de una aplicación desde cero. En este caso, nos centraremos en cómo facilitar al usuario la tarea de realizar la compra doméstica, tanto de alimentos como de productos del hogar, de manera eficiente y organizada. También se tendrá en cuenta cómo gestiona las recetas y los invitados que van a comer a su hogar.

Para ello el trabajo se centra en el desarrollo de una aplicación móvil nativa para el sistema operativo Android, denominada *Pinche*.

La forma de alcanzar este objetivo consistirá en recorrer cada una de las etapas del proceso de desarrollo, adoptando los distintos roles que forman parte de un equipo digital: experiencia de usuario (UX), producto y desarrollo. Aunque el núcleo del trabajo se centrará en la parte técnica de desarrollo, también se tendrán en cuenta aspectos clave de diseño y definición del producto.

Adoptar esta perspectiva permite también tomar decisiones informadas a la hora de implementarla sobre las necesidades reales de los usuarios, los problemas que enfrentan en su día a día y cómo una solución tecnológica puede aportar valor real. Al asumir diferentes roles permite entenderemos los retos de la comunicación entre perfiles técnicos y no técnicos y la importancia de no perder el foco en el usuario final.

La motivación del proyecto radica tiene en cuenta también aplicar y reforzar conocimientos adquiridos durante el grado, desde el diseño de interfaces hasta la integración de servicios en la nube. Pero sin permanecer como técnicos o programadores aislados que no tienen el contexto del proyecto.

Desde ese punto de vista, el puramente técnico, se ha optado por utilizar tecnologías modernas del ecosistema Android: el lenguaje de programación Kotlin, el framework Jetpack Compose para la construcción de interfaces declarativas y los servicios de Firebase: Firestore como base de datos y Firebase Authentication para la gestión de usuarios. Estas tecnologías permiten un desarrollo flexible, escalable y están alineadas con las demandas actuales del sector.

Desde la perspectiva de diseño UX se ha adoptado una combinación de la metodología Design Thinking para la definición inicial del problema y el uso de la herramienta Figma para llevar acabo un diseño final intuitivo y que sea fácilmente iterable.

Por último, desde el lado de producto y gestión del proyecto, se aplica la metodología Lean Startup para validar hipótesis y metodologías ágiles como Scrum para organizar el desarrollo en iteraciones en Trello.

En resumen, el proyecto se plantea como una oportunidad para consolidar conocimientos técnicos, mejorar habilidades de diseño y comunicación, y resolver una necesidad real con impacto práctico.

Capítulo 3

Tecnologías, Herramientas y Metodologías

3.1. Lenguaje de programación

Para implementar una aplicación nativa en Android se puede usar como lenguaje de programacion Java o Kotlin. El desarrollo de Pinche se ha realizado utilizando Kotlin.

Las principales razones por las que se ha decidido usar Kotlin para el desarrollo de esta aplicación frente a Java son:

- **Sintaxis más concisa y expresiva**: Kotlin permite reducir la cantidad de código, optimizando la Implementación. Por ejemplo, la gestión de getters/setters, estructuras de datos y operaciones sobre colecciones se realiza de forma mucho más directa y legible.

- **Seguridad frente a errores de null**: Kotlin incluye un sistema de tipos que distingue claramente entre referencias anulables y no anulables, lo que reduce la probabilidad de errores en tiempo de ejecución relacionados con punteros nulos (el conocido *NullPointerException* en Java).

- **Interoperabilidad con Java**: Kotlin es totalmente interoperable con Java, lo que permite utilizar bibliotecas existentes. Esto es especialmente útil en Android, ya que muchas APIs aún están escritas en Java.

- **Compatibilidad con herramientas modernas de Android**: Kotlin se integra de forma nativa con bibliotecas modernas como Jetpack Compose, Hilt, Coroutines o Navigation,

lo que simplifica el uso de arquitecturas modernas (como MVVM) y prácticas de desarrollo actuales.

- **Soporte oficial, comunidad y futuro garantizado**: Google ha declarado que "Android is Kotlin-first", lo que coloca a Kotlin como lenguaje oficial recomendado.

3.2. Frameworks y librerías

3.2.1. Jetpack Compose

Para la construcción de la interfaz de usuario se ha utilizado **Jetpack Compose**, el framework de Android más moderno para crear interfaces de forma declarativa. Esta elección responde a usar herramientas actuales, recomendadas por Google, que permiten mayor productividad y fluidez en el desarrollo.

Durante mucho tiempo, el desarrollo de interfaces en Android se ha basado en el uso de archivos `XML` que describen los elementos de la interfaz de forma estática, combinados con código Java o Kotlin para enlazar y actualizar esos elementos en tiempo de ejecución. Este enfoque impone una separación forzada entre lógica e interfaz, y obliga a utilizar métodos como `findViewById()` o `ViewBinding`, lo que complica la sincronización entre los datos y la vista.

Jetpack Compose rompe con este modelo tradicional al introducir una forma de construir la UI de manera totalmente **declarativa**, es decir, describiendo *qué* debe mostrarse en lugar de *cómo* hacerlo paso a paso. La interfaz se define directamente en Kotlin, mediante funciones composables, lo que permite una mejor integración con la lógica de negocio, la reutilización de componentes y mejorar la legibilidad del código.

Entre las principales ventajas de Jetpack Compose frente a XML destacan:

- **Menor complejidad**: No es necesario gestionar manualmente el enlace entre XML y código. La UI responde automáticamente a los cambios de estado.

- **Código más conciso y reutilizable**: Al trabajar con funciones de Kotlin, se pueden componer interfaces complejas a partir de pequeños componentes reutilizables.

- **Mejor integración con la arquitectura moderna**: Compose se integra de forma nativa con patrones como MVVM que es el que seguimos en este proyecto, flujos reactivos como StateFlow e incluso herramientas de testing específicas para UI.
- **Vista previa en tiempo real**: Android Studio permite visualizar cambios en la UI mientras se escribe el código, lo que acelera el diseño y validación de pantallas. A diferencia de XML con el que tienes que construir tu aplicación cada vez que quieras ver el impacto de un cambio.
- **Mantenimiento más sencillo**: Al eliminar el archivo XML y mantener toda la lógica en un solo lenguaje, se reduce el esfuerzo de mantenimiento y depuración.

3.2.2. Jetpack Navigation

Para la gestión de la navegación entre pantallas se ha utilizado **Jetpack Navigation**, que permite definir flujos de navegación mediante un grafo centralizado, asegurando la coherencia del estado de la aplicación y facilitando el paso de argumentos entre distintas pantallas y componentes.

3.2.3. Hilt

Con el objetivo de aportar robustex y escalabilidad a la aplicación, hemos recurrido a **Hilt** como sistema de *inyección de dependencias*. Esto nos ha permitido simplificar la gestión de instancias y facilitar su sustitución por instancias de prueba para llevar a cabo los test. También se ha tenido en cuenta su integración con el ciclo de vida recomendado de Android y su compatibilidad con Jetpack.

3.3. Firebase

Firebase, de Google, es una plataforma de desarrollo de aplicaciones que proporciona una serie de servicios backend listos para usar: bases de datos en tiempo real, autenticación de usuarios, almacenamiento en la nube, análisis, herramientas de pruebas y despliegue. Está pensada especialmente para aplicaciones móviles y web. Su uso agiliza significamente el desarrollo del backend en aplicaciones.

En este proyecto se utilizan principalmente dos componentes clave de Firebase:

- **Firebase Authentication:** permite implementar de forma sencilla el mecanismo de registro y acceso mediante email y contraseña que hemos utilizado en Pinche.

- **Cloud Firestore:** una base de datos NoSQL que permite almacenar y sincronizar datos en tiempo real entre los dispositivos de los usuarios y la nube, con una estructura basada en colecciones y documentos [7].

Una de las principales ventajas de Firebase es su integración nativa con Android, lo que simplifica su uso desde Kotlin, además de su escalabilidad y su soporte para herramientas de desarrollo como Android Studio y *Firebase Emulator Suite*, que permite hacer pruebas locales de testeo sin impactar en la base de datos real.

3.4. Entorno de desarrollo

El entorno de desarrollo utilizado ha sido **Android Studio**, IDE oficial de Android. Proporciona herramientas avanzadas como el emulador y da soporte completo para Compose, Kotlin y Firebase. Para las pruebas de la app se han utilizado tanto emuladores como dispositivos físicos.

3.5. Control de versiones

Para la gestión del código fuente se ha utilizado el conocido sistema de control de versiones **Git** y **GitHub** como repositorio remoto. El trabajo se ha organizado siguiendo prácticas comunes en entornos colaborativos. Esto ha permitido mantener versiones estables y realizar integraciones progresivas de nuevas funcionalidades. El flujo de trabajo ha sido el siguiente: se han creado ramas para la implementación de tareas concretas, por ejemplo: `feature/login`, una vez finalizada la tarea y testeada de manera atómica su funcionalidad se mergea en la rama `dev` donde se comprueba que la nueva funcionalidad no afecta a las implementadas anteriormente y tras las comprobaciones se realiza la actualizacion de la rama `main` que es la rama estable del proyecto.

3.6. Metodologías de desarrollo

3.6.1. Design Thinking

Durante la fase inicial se aplicó el enfoque de **Design Thinking**, con especial énfasis en la definición del problema y centrada en el usuario objetivo. Su objetivo es generar soluciones de acuerdo con problemas detectados en un determinado marco de trabajo. [2] [3].

3.6.2. Lean Startup

Lean Startup es una metodología que ayuda al desarrollo de productos o servicios de manera ágil, reduciendo los riesgos, promoviendo el aprendizaje y disminuyendo el tiempo de lanzamiento. Todo esto contribuye también a reducir los gastos y riesgos, al situar al cliente real en el centro de todas las decisiones de desarrollo [9].

Los principios fundamentales de Lean Startup son:

- **Producto Mínimo Viable (MVP)**: Se definen unas hipótesis a confirmar y se genera una versión básica del producto que cubra la funcionalidad necesaria para comprobar dichas hipótesis y redirigir la definición del producto según lo que funciona y lo que no.

- **Construir-Medir-Aprender**: Se construye el MVP, se mide su desempeño y aceptación por parte del usuario, se recopilan datos y se decide en base a ellos. Este ciclo se repite durante toda la definición del producto.

- **Experimentación continua**: Se evalúa constantemente el producto o servicio, de manera que no hay un plan fijo de acción ni se comprometen recursos desde el principio.

- **Iteraciones rápidas**: La generación de MVPs debe ser ágil, demostrando flexibilidad a la hora de aplicar los cambios necesarios tras la última iteración.

- **Validación de hipótesis**: Se utilizan métricas y datos que ayudan a vislumbrar si una idea de negocio tiene mercado o no, evitando el gasto innecesario de recursos.

3.6.3. Agile

La metodología **Agile** se centra en la flexibilidad, la colaboración y la entrega incremental de valor al cliente durante el desarrollo de un proyecto. Con esta metodología se pretende generar una mayor adaptabilidad frente a los cambios que surgen durante el desarrollo, reduciendo los tiempos de entrega y mejorando la calidad del producto.

Agile se basa en el *Manifiesto Ágil*, publicado en 2001, que consta de cuatro valores principales y doce principios.

Valores del Manifiesto Ágil:

- Individuos e interacciones frente a procesos y herramientas.
- Software funcional sobre documentación extensiva.
- Colaboración con el cliente frente a negociación de contratos.
- Responder ágilmente al cambio frente a seguir un plan rígido.

Principios:

- Satisfacer al cliente mediante la entrega rápida y continua de software funcional.
- Aceptar los cambios, los requisitos por cumplir pueden ser incluidos en cualquier etapa del desarrollo.
- Entregar frecuentemente un producto funcional, en ciclos cortos de tiempo.
- Colaboración constante con el cliente para asegurar que se cubren todas sus necesidades.
- Motivar a los equipos de trabajo para fomentar su implicación y confianza.
- Comunicación cara a cara como medio más eficiente y efectivo.
- Valorar el software funcional como principal medida de progreso.
- Sostenibilidad mediante una velocidad constante y sostenible para que el ritmo al que trabaja el equipo sea accesible.

- Atención a la excelencia técnica y al buen diseño.
- Simplicidad, eliminando lo innecesario o lo que no aporta valor real.
- Equipos autoorganizados con capacidad de decisión.
- Revisión y ajuste constantes para mejorar la efectividad.

Algunas metodologías que surgen del enfoque Agile son **Scrum**, **Kanban** y **Extreme Programming (XP)**.

Scrum. Es una metodología de trabajo que permite el manejo de proyectos complejos permitiendo que los equipos trabajen de manera iterativa e incremental. El trabajo se organiza en **Sprints**, que son ciclos cortos de tiempo que tienen como objetivo un incremento funcional del producto que el equipo está desarrollando.[10]

Entre sus principales elementos dentro de la metodología Scrum encontramos roles claves como Product Owner, Scrum Master y el equipo de desarrollo. El **Product Owner** define cuáles son las prioridades que maximizan el valor del producto. El **Scrum Master** facilita el proceso Scrum y se asegura de que se aplica correctamente. Y, por último, el **equipo de desarrollo** es el encargado de implementar el producto.

Dentro de esta metodología también encontramos elementos o herramientas clave que facilitan su aplicación: **Product Backlog**, una lista priorizada de todas las tareas del proyecto, **Sprint Backlog**, que abarca todas las tareas a realizar en un sprint, y el **incremento** que sería el producto funcional que entrega el equipo de desarrollo al final de cada sprint.

Para la gestión del tiempo dentro de un sprint, Scrum recomienda realizar cuatro encuentros: Sprint Planning, Daily Scrum, Sprint Review y Sprint Retrospective. En el **Sprint Planning** el objetivo será planificar qué tareas, que deben ser atómicas, se van a desarrollar este sprint. Diariamente el progreso se coordinará en la sesión de **Daily Scrum**, será el momento de poner sobre la mesa posibles bloqueos que el equipo ha encontrado o los avances que ha ido realizando. Una vez finalizado el sprint tendremos el **Sprint Review** donde se revisará qué cantidad de trabajo ha sido realizado. Por último el **Sprint Retrospective**, una sesión en la que el equipo pueda reflexionar sobre cómo mejorar para el próximo Sprint.

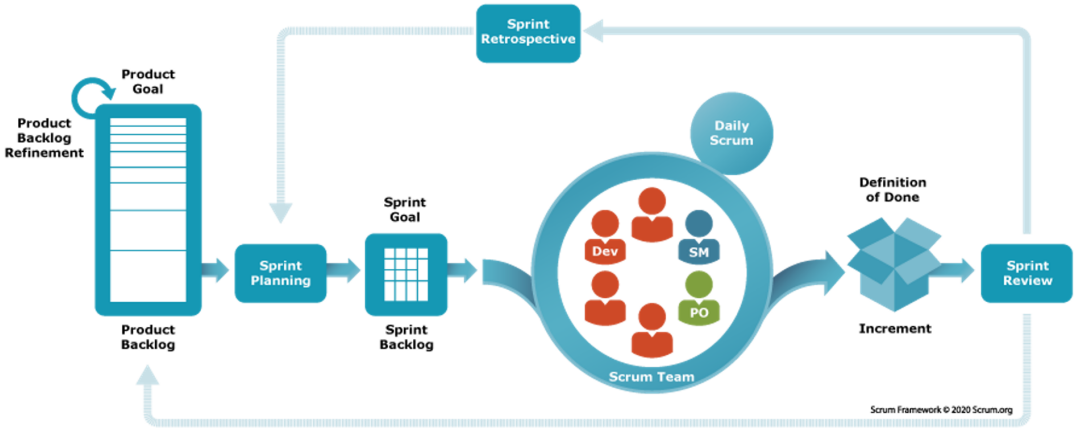

Figura 3.1: Esquema de funcionamiento de Scrum.

Fuente: `https://www.scrum.org/learning-series/what-is-scrum/`

Kanban. Para esta metodología se suele utilizar un tablero visual (*Tablero Kanban*) para representar el flujo de trabajo. Las tareas se representan como tarjetas que se mueven por columnas como *Por hacer*, *En progreso* y *Terminado*. Se establece un límite de trabajo en curso (WIP: Work In Progress, trabajo en progreso) para evitar sobrecargas. Kanban permite detectar cuellos de botella, optimizar procesos y mejorar de forma continua la eficiencia del equipo.

Extreme Programming (XP). XP enfatiza la mejora continua y la satisfacción del cliente. Reduce riesgos mediante prácticas técnicas rigurosas:

1. Desarrollo iterativo con entregas frecuentes de software funcional.
2. Pruebas constantes (TDD: Test-Driven Development, desarrollo guiado por tests).
3. Programación en pareja. Dos desarrolladores trabajan simultáneamente en la misma tarea con el propósito de reducir errores y mejorar el aprendizaje del equipo.
4. Integración continua.
5. Código simple y funcional.
6. Retroalimentación rápida del cliente. El cliente está presente de manera frecuente en el proceso de desarrollo y proporciona su punto de vista para ajustar el desarrollo a sus necesidades

3.7. Diseño de la interfaz

La interfaz ha sido diseñada siguiendo principios de UX/UI centrados en la simplicidad, accesibilidad y claridad visual. Se han creado prototipos en Figma, permitiendo iterar sobre la estructura de navegación, la jerarquía visual y la organización de las secciones: listas, recetas e invitados.

Figma es una herramienta web de diseño de interfaces que permite crear prototipos interactivos, colaborar en tiempo real y compartir diseños entre equipos multidisciplinares. Es ampliamente utilizada en entornos de desarrollo ágil, ya que facilita la comunicación entre diseñadores, desarrolladores y clientes [5].

3.8. Herramientas de testeo

Durante el desarrollo se han implementado diferentes tipos de tests para garantizar la calidad del producto, cada una con su propósito específico dentro del flujo de desarrollo. A continuación se describen los tipos principales de pruebas y las herramientas utilizadas en cada caso.

3.8.1. Test unitarios

Las pruebas unitarias se centran en comprobar el comportamiento de unidades individuales de código, normalmente funciones o clases, de forma aislada. Estas pruebas permiten validar la lógica de negocio sin dependencias externas.

JUnit4 ha sido la herramienta principal para implementar los test unitarios en este proyecto [1]. Se trata del framework más consolidado en el ecosistema Java/Kotlin, ampliamente soportado y con buena integración en Android Studio. Junto a JUnit, se ha utilizado **MockK**, una librería de mocking especialmente diseñada para Kotlin, que permite simular dependencias como repositorios o servicios, da soporte para coroutines y se integra sin problema con los constructores Kotlin puros.

3.8.2. Test de interfaz de usuario (UI)

Los tests de UI tienen como objetivo comprobar que los componentes visuales se comportan correctamente frente a interacciones del usuario (clics, navegación, aparición de textos, etc.).

La libreria , específica para Jetpack Compose, ha sido la elegida para implementar este tipo de test en Pinche. Es una herramienta que permite simular eventos de usuario y verificar la presencia y el estado de los elementos de la pantalla [1].

La ventaja frente a herramientas clásicas como Espresso para XML es que está optimizada para el modelo declarativo de Compose, lo que la hace más sencilla, menos propensa a errores de sincronización y mejor integrada con el ciclo de recomposición de la UI.

3.8.3. Test de integración

Los tests de integración validan el correcto funcionamiento de múltiples componentes combinados, como la lógica de presentación con la capa de datos.

Para este tipo de pruebas se ha utilizado **Firebase Emulator Suite**, una herramienta que permite simular servicios de Firebase (Firestore y Authentication en nuestro caso) localmente sin necesidad de acceder a entornos reales [8]. Esto permite hacer pruebas reproducibles, seguras y con bajo coste.

Además, se han utilizado **repositorios falsos** (fakes) y la capacidad de Hilt para sustituir dependencias reales por versiones de prueba. Esto permite realizar pruebas sin tocar los datos reales ni alterar la lógica de producción.

Frente a enfoques como instrumentación real con dispositivos, esta estrategia ofrece mayor velocidad, control del entorno y testeo determinista.

Capítulo 4

Descripción informática

4.1. Requisitos

La aplicación **Pinche** nace con el objetivo de facilitar la organización de la compra doméstica mediante una aplicación móvil accesible, intuitiva y práctica. Para ello, se han definido una serie de requisitos funcionales que reflejan las necesidades del usuario final, así como requisitos no funcionales que garantizan la calidad de la aplicación.

4.1.1. Requisitos funcionales

Para definir los requisitos funcionales se ha seguido una metodología personalizada que combina las fortalezas de las metodologías **Scrum**, **Kanban** y **Extreme Programming (XP)**. Se ha seguido la estructura iterativa de Scrum, utilizando tableros Kanban para visualizar el flujo de tareas y adoptando buenas prácticas como pruebas continuas e integración frecuente propias de XP.

La organización del desarrollo ha comenzado con la definición de **historias de usuario**, expresadas en el formato: "Como [tipo de usuario], quiero [funcionalidad] para [beneficio]". Estas historias fueron elaboradas y refinadas por el equipo de desarrollo en conjunto con el Product Owner, y se han estimado mediante la técnica de **Story Points** basada en la sucesión de Fibonacci.

Las historias se han agrupado en **épicas**, agrupaciones lógicas que organizan funcionalidades principales: *Cuenta de usuario*, *Lista de la compra*, *Recetas* e *Invitados*.

La épica *Cuenta de usuario* agrupa un total de seis historias de usuario con sus respectivas tareas:

1. Como usuario, necesito registrarme en la aplicación con la finalidad de poder utilizarla. Estimación: 2 puntos de historia.

2. Como usuario, necesito iniciar sesión en la aplicación con la finalidad de acceder a mi cuenta de usuario y mis datos. Estimación: 1 punto de historia.

3. Como usuario, necesito cerrar sesión en la aplicación con la finalidad de no dejar mi cuenta activa en un dispositivo. Estimación: 1 punto de historia.

4. Como usuario, quiero recuperar mi contraseña de mi cuenta de usuario de la aplicación con la finalidad de poder acceder a mi cuenta. Estimación: 1 punto de historia.

5. Como usuario, quiero ver un resumen de mi actividad con la finalidad de conocer cuál es mi uso de la aplicación. Estimación: 2 puntos de historia.

6. Como usuario, necesito eliminar mi cuenta de usuario de la aplicación con la finalidad de borrar mis datos y dejar de tener cuenta de usuario en la aplicación. Estimación: 1 punto de historia.

La épica *Lista de la compra* agrupa un total de trece historias de usuario:

1. Como usuario, quiero ver el listado de las listas de la compra que he creado con la finalidad de saber cuántas y qué listas de la compra tengo. Estimación: 2 puntos de historia.

2. Como usuario, quiero ver el listado de listas de la compra que aún no han sido completadas con la finalidad de poder visualizar más fácilmente que listas tienen artículos por comprar. Estimación: 1 punto de historia.

3. Como usuario, quiero acceder al detalle de una lista de la compra en concreto con la finalidad de ver sus artículos. Estimación: 1 punto de historia.

4. Como usuario, quiero eliminar una lista de la compra, aunque haya sido o no completada, con la finalidad de dejar de visualizarla en mi listado. Estimación: 1 punto de historia.

5. Como usuario, quiero seleccionar como comprados todos los artículos de una lista de la compra con la finalidad de poder completar una lista en un solo click. Estimación: 1 punto de historia.

6. Como usuario, quiero seleccionar como pendientes de comprar todos los artículos de una lista con la finalidad de volver a reutilizar esa lista. Estimación: 1 punto de historia.

7. Como usuario, quiero añadir una nueva lista de la compra con la finalidad de tener una nueva lista donde organizar mis artículos. Estimación: 1 punto de historia.

8. Como usuario, quiero añadir un artículo a una lista de la compra con la finalidad de recordar que tengo que comprar dicho artículo. Estimación: 1 punto de historia.

9. Como usuario, quiero seleccionar un artículo como comprado con la finalidad de poder visualizar qué artículos ya he comprado. Estimación: 1 punto de historia.

10. Como usuario, quiero seleccionar un artículo que ya había seleccionado como comprado como no comprado en una lista de la compra con la finalidad de poder volver a tener ese artículo como pendiente de comprar. Estimación: 1 punto de historia.

11. Como usuario, quiero eliminar un artículo de una lista de la compra con la finalidad de dicho artículo deje de formar parte de la lista de la compra. Estimación: 1 punto de historia.

12. Como usuario, quiero especificar el nombre del artículo, la cantidad, la unidad de medida de esa cantidad, la tienda donde prefiero comprar y opcionalmente una fotografía del artículo, con la finalidad de disponer de toda la información que necesito de ese artículo. Estimación: 2 puntos de historia.

13. Como usuario, quiero editar la información de un artículo de una lista de la compra con la finalidad de modificar su contenido. Estimación: 1 punto de historia.

La épica *Recetas* agrupa un total de ocho historias de usuario:

1. Como usuario, quiero ver un listado con mis recetas con la finalidad de conocer que recetas tengo registradas. Estimación: 2 puntos de historia.

2. Como usuario, quiero añadir una nueva receta desde la pantalla de listas de recetas con la finalidad de añadir una nueva receta y sus datos. Estimación: 1 punto de historia.

3. Como usuario, quiero editar una receta con la finalidad de actualizar datos de esa receta. Estimación: 1 punto de historia.

4. Como usuario, quiero eliminar una receta con la finalidad de que no aparezca esa receta en mi listado de recetas. Estimación: 1 punto de historia.

5. Como usuario, quiero acceder a la información de una receta desde la pantalla de lista de recetas con la finalidad de ver en detalle los datos de esa receta. Estimación: 1 punto de historia.

6. Como usuario, quiero añadir información a una nueva receta con la finalidad de tener toda su información recopilada. Estimación: 2 puntos de historia.

7. Como usuario, quiero editar la información de una receta con la finalidad de tener la información de esa receta actualizada. Estimación: 1 punto de historia.

8. Como usuario, quiero eliminar una receta con la finalidad de eliminar esa receta de mis recetas. Estimación: 1 punto de historia.

La épica *Invitados*, agrupa un total de ocho historias de usuario:

1. Como usuario, quiero ver un listado con mis invitados con la finalidad de conocer que invitados tengo registrados. Estimación: 2 puntos de historia.

2. Como usuario, quiero añadir un nuevo invitado desde la pantalla de listas de invitados con la finalidad de añadir un nuevo invitado y sus datos. Estimación: 1 punto de historia.

3. Como usuario, quiero editar un invitado con la finalidad de actualizar datos de ese invitado. Estimación: 1 punto de historia.

4. Como usuario, quiero eliminar un invitado con la finalidad de que no aparezca ese invitado en mi listado de invitados. Estimación: 1 punto de historia.

5. Como usuario, quiero acceder a la información de un invitado desde la pantalla de lista de invitados con la finalidad de ver en detalle los datos de ese invitado. Estimación: 1 punto de historia.

6. Como usuario, quiero añadir información a un nuevo invitado con la finalidad de tener toda su información recopilada. Estimación: 2 puntos de historia.

7. Como usuario, quiero editar la información de un invitado con la finalidad de tener la información de ese invitado actualizada. Estimación: 1 punto de historia.

8. Como usuario, quiero eliminar un invitado con la finalidad de eliminar ese invitado de mis invitados. Estimación: 1 punto de historia.

Una vez refinadas y estimadas, se ha preparado el primer Sprint de dos semanas de duración. Se ha fijado un límite de 13 puntos de historia por Sprint. En este primer ciclo, se priorizaron historias de cada épica para implementar las funcionalidades básicas. Sin embargo, solo se completaron 6 de las 8 historias planificadas debido a la carga de requisitos no funcionales que se requeria implementar en este primer Sprint.

El seguimiento del Sprint se ha realizado mediante un tablero **Kanban** creado en **Trello**, donde se representan las historias como tarjetas con sus tareas asociadas. La Figura 4.1 muestra cómo se define una historia de usuario en Trello y la Figura 4.2 el estado final del tablero en el primer Sprint.

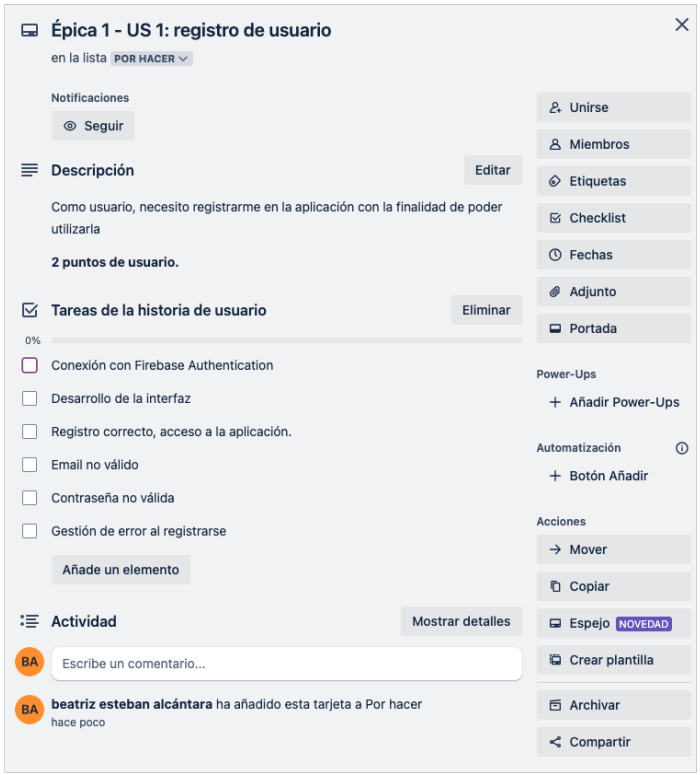

Figura 4.1: Historia de usuario en Trello

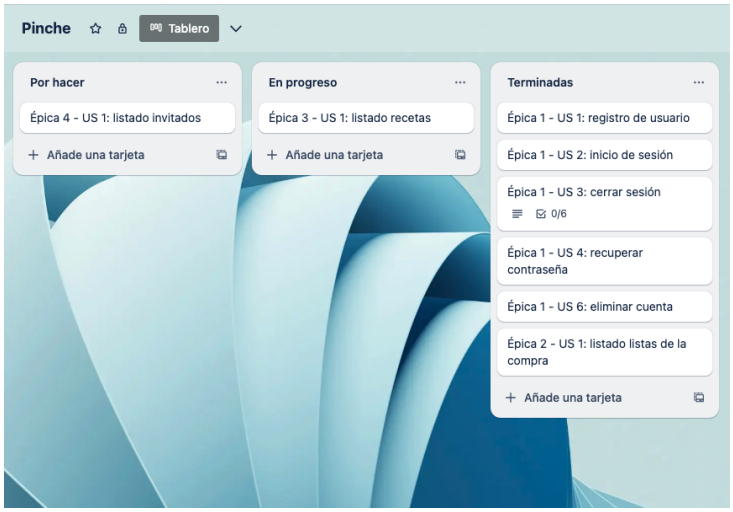

Figura 4.2: Estado del tablero Kanban al final del primer Sprint

Tras la sesión de *Sprint Review* y la de *Sprint Retrospective*, se ajustó el alcance estimado para los siguientes Sprints a un máximo de 8 puntos de historia por iteración, adaptando así la velocidad al contexto real del desarrollo.

Una hoja resumen con todas las historias de usuario, agrupadas por épica, sus tareas y estimaciones, se adjunta en el anexo `Historias de usuario.xlsx`.

4.1.2. Requisitos no funcionales

Además de la funcionalidad, se han establecido los siguientes requisitos no funcionales:

- Conexión con Firestore y creación de repositorios de datos.
- Autenticación de usuarios con Firebase Authentication.
- Estructura modular y escalable, basada en el patrón arquitectónico **MVVM** (Model-View-ViewModel).
- Inyección de dependecias con Hilt.
- Incluir tests tanto unitarios como de interfaz.

Estos requisitos definen el alcance funcional y técnico del proyecto, garantizando que el producto final responda tanto a las expectativas del usuario como a criterios de calidad del software.

4.2. Arquitectura y análisis

La arquitectura de una aplicación móvil resulta crítica para garantizar escalabilidad, optimización del código, rendimiento, solidez, mantenibilidad y la correcta separación de responsabilidades. En este proyecto se ha adoptado la arquitectura recomendada por Android, basada en una estructura en capas y el patrón MVVM (Model-View-ViewModel), con la inclusión de una capa de dominio opcional y la gestión de dependencias mediante Hilt.

4.2.1. Principios generales

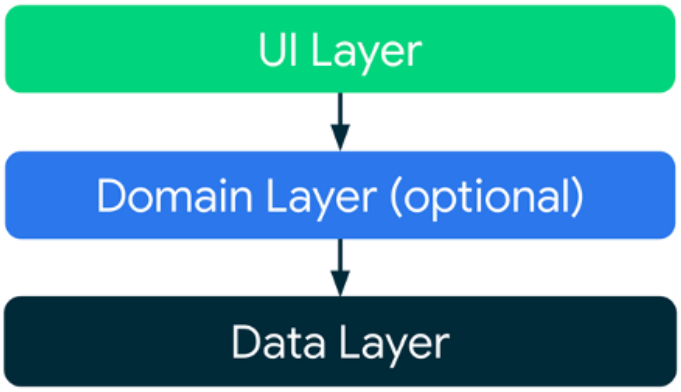

Figura 4.3: Diagrama arquitectura MVVM.

Fuente: `https://developer.android.com/topic/architecture?hl=es-419`

Esta arquitectura promueve los principios de separación de responsabilidades, flujo unidireccional de datos, modularidad y testabilidad [**?**]. El modelo propuesto divide la lógica de la aplicación en tres capas principales: UI, dominio (opcional) y datos, Figura 4.3. En nuestro caso vamos a utilizar la arquitectura recomendada incluyendo la capa opcional de dominio, Figura 4.4.

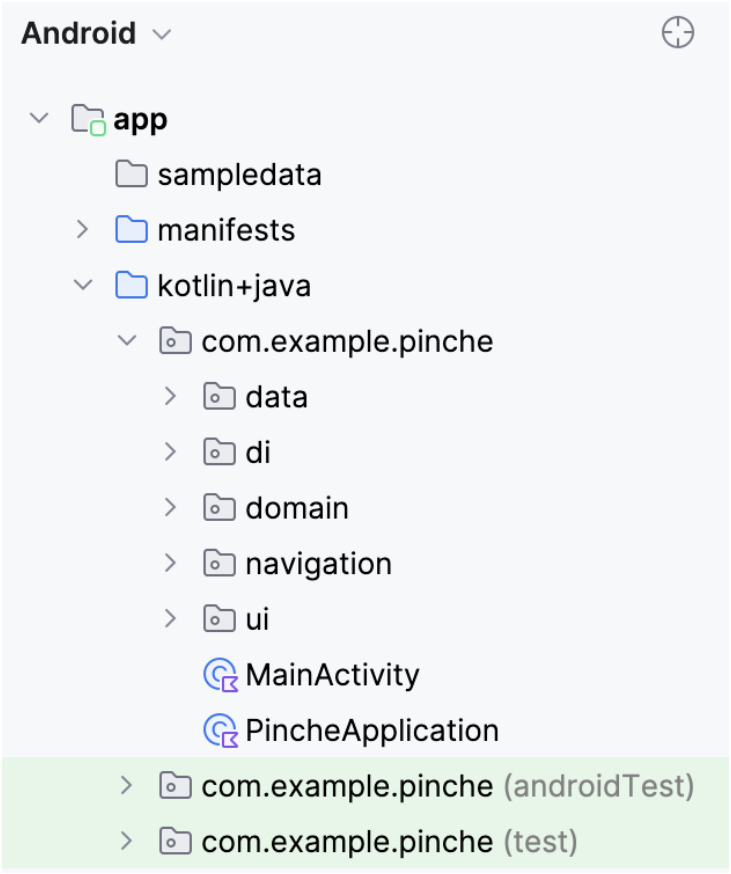

Figura 4.4: Arquitectura de la aplicación Pinche.

4.2.2. Capa de UI

La capa de interfaz de usuario es responsable de mostrar los datos al usuario y recoger sus interacciones. Esta capa se compone de dos tipos de elementos:

- **Elementos de la UI**: Son funciones de Jetpack Compose que renderizan los datos en pantalla, Figura 4.5.

- **Contenedores de estado**: Los ViewModel que retienen el estado de la UI y exponen datos reactivos mediante `StateFlow`, Figura 4.6.

```kotlin
@Composable
fun RecoverPasswordForm(
    modifier: Modifier = Modifier,
    viewModel: RecoverPasswordViewModel = hiltViewModel()
) {
    var email by remember { mutableStateOf( value: "") }
    var formError by remember { mutableStateOf( value: false) }

    fun errorInForm(): Boolean {
        return !email.contains( other: "@")
    }

    Column (
        modifier = modifier,
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {...}
}
```

Figura 4.5: Componente Compose `RecoverPasswordForm` de Pinche

```kotlin
@HiltViewModel
class RecoverPasswordViewModel @Inject constructor(
    private val sendPaswordResetEmailUseCase: SendPaswordResetEmailUseCase
): ViewModel() {
    var recoverPasswordResponse by mutableStateOf<SendPasswordResetEmailResponse>(Inactive)
        private set

    fun sendPasswordResetEmail(email: String) = viewModelScope.launch( Dispatchers.IO ) {
        recoverPasswordResponse = Loading
        recoverPasswordResponse = sendPaswordResetEmailUseCase(email)
    }

    fun resetRecoverPasswordResponse() {
        recoverPasswordResponse = Inactive
    }
}
```

Figura 4.6: ViewModel `RecoverPasswordForm` de Pinche

4.2.3. Capa de dominio

La capa de dominio encapsula la lógica empresarial que puede ser compartida entre distintos ViewModels. En ella se definen los casos de uso (*use cases*) como clases que representan acciones o procesos específicos de la aplicación, facilitando la reutilización de código y la claridad del flujo de datos. La Figura 4.7 muestra un ejemplo de caso de uso de la aplicación Pinche.

```kotlin
class SendPaswordResetEmailUseCase @Inject constructor(
    private val authRepository: AuthRepository
) {
    suspend operator fun invoke(email: String) = authRepository.sendPasswordResetEmail(email)
}
```

Figura 4.7: Caso de uso `SendPasswordResetEmailUseCase` de Pinche

4.2.4. Capa de datos

Esta capa es responsable de la obtención y persistencia de datos. Se organiza en:

- **Repositorios**: Actúan como intermediarios entre las fuentes de datos y la capa de dominio.
- **Fuentes de datos**: Pueden ser locales o remotas. En nuestro caso, usamos la fuente de datos remota Firebase Authentication que proporciona servicios de autenticación y la base de datos en la nube Firestore [7, 6].

En la Figura 4.8 muestra la función del reposositorio de autenticación responsable de intermediar con Firebase Authentication para recuperar la contraseña de un usuario enviándole un correo electrónico a su dirección de correo.

```
class AuthRepositoryImpl @Inject constructor(

    //Recover user password
    override suspend fun sendPasswordResetEmail(
        email: String
    ) = try {
        Log.d(logTag, msg: "Starting reset user password")
        auth.sendPasswordResetEmail(email).await()
        Log.d(logTag, msg: "Recover user password: success")
        Success( data: true)
    } catch (e: Exception) {
        Log.d(logTag, msg: "Recover user password: exception. ${e.message}")
        Failure(e)
    }
```

Figura 4.8: Función `sendPasswordResetEmail` en el repositorio de autenticación

Gracias a esta separación, si por ejemplo en el futuro se quisiera cambiar de base de datos o el método de autenticación, solo se vería afectada la capa de datos, manteniendo intacto el resto del sistema.

4.2.5. Gestión de dependencias con Hilt

La gestión de dependencias se realiza mediante la biblioteca Hilt como ya hemos comentado. Hilt proporciona objetos preconfigurados a las clases de Android y administra automáticamente su ciclo de vida [**?**].

4.2.6. Ventajas de esta arquitectura

La aplicación Pinche se beneficia de esta arquitectura al conseguir:

- Mejor separación de responsabilidades entre componentes.
- Mayor facilidad de testeo en todas las capas.
- Reutilización de lógica empresarial entre distintas pantallas.
- Facilidad para mantener y escalar el código base en el futuro.

Este enfoque permite un desarrollo estructurado, coherente y sostenible, favoreciendo además la implementación de buenas prácticas de ingeniería de software.

4.3. Diseño e implementación

El desarrollo de la aplicación **Pinche** se ha centrado en aplicar buenas prácticas de ingeniería de software para Android, incluyendo el patrón MVVM, el uso de Jetpack Compose y la organización modular del código.

4.3.1. Análisis de requerimientos y estructura funcional

La aplicación ha sido diseñada en base a un conjunto de funcionalidades organizadas en tres secciones principales: listas de la compra, recetas e invitados. Este enfoque permite abordar de forma modular las necesidades de los usuarios y facilita la extensión del sistema.

METER AQUI LO DE FIGMA Y DIAGRAMAS DE COMPONENTES, DE CASOS DE USO Y ESAS COSAS?

METER CAPTURAS DE LA PROPIA APLICACION.

El modelo de datos se ha estructurado para cubrir las siguientes entidades principales:

- **Lista**: contiene un conjunto de productos, su nombre y estado.
- **Producto**: asociado a una lista, con nombre, cantidad y supermercado.
- **Receta**: incluye nombre, pasos de elaboración, ingredientes y número de comensales.

- **Ingrediente**: definido por nombre y cantidad relativa a los comensales.
- **Invitado**: con nombre, preferencias alimentarias, intolerancias y registro de comidas anteriores.

Este modelo facilita la integración de funcionalidades como el cálculo de ingredientes según el número de comensales, la generación de listas automáticas a partir de recetas y la personalización de menús para invitados con restricciones.

4.3.2. Decisiones técnicas relevantes HABLAR DE Jetpack Navigation SOLO? aqui o donde? Y SU INTEGRACION CON HILT NAVIGATION

Durante el desarrollo se tomaron decisiones clave como:

- Uso de **Jetpack Navigation** para gestionar la navegación entre pantallas de manera estructurada.
- Implementación de **Hilt** para inyección de dependencias, facilitando el testeo y la escalabilidad.
- Empleo de **StateFlow** para la gestión del estado de la UI, evitando fugas de memoria.
- Utilización de **Git** como sistema de control de versiones, junto con ramas por tarea y buenas prácticas de integración continua (CI).
- Desarrollo colaborativo organizado a través de tableros **Trello** y prototipos creados en **Figma**.

Esta implementación modular y centrada en buenas prácticas garantiza que Pinche pueda mantenerse y escalarse en el tiempo, facilitando la extensión de funcionalidades futuras.

4.4. Pruebas

El aseguramiento de la calidad en el desarrollo de la aplicación **Pinche** se ha abordado mediante una estrategia de pruebas estructurada que cubre distintos niveles de validación del

sistema: pruebas unitarias, de interfaz, de integración, de extremo a extremo (E2E) y cobertura de código. Esta estrategia ha permitido verificar que los componentes funcionan correctamente de forma individual y en conjunto.

4.4.1. Pruebas unitarias

Las pruebas unitarias validan pequeñas unidades de código de forma aislada, principalmente funciones puras o clases sin dependencias externas. Para ello se ha utilizado **JUnit4** como framework base, complementado con **MockK** para simular comportamientos de dependencias externas como repositorios o servicios [1].

MockK se ha elegido frente a otras alternativas como Mockito debido a su compatibilidad nativa con Kotlin, su soporte para coroutines y su sintaxis más idiomática.

el Listado 4.1,

```
@ExperimentalCoroutinesApi
class SignInUserUseCaseTest {

  @get:Rule
  val coroutinesTestRule = MainDispatcherRule()

  private val authRepository: AuthRepository = mockk()
  private lateinit var signInUserUseCase: SignInUserUseCase

  private val email = "test@example.com"
  private val password = "123456"

  @Before
  fun setup() {
      signInUserUseCase = SignInUserUseCase(authRepository)
  }

  @Test
  fun `sign in returns Success when repository succeeds`() = runTest {
      coEvery { authRepository.firebaseSignInWithEmailAndPassword(email,
          password) } returns Success(true)

```

```
22          val result = signInUserUseCase(email, password)
23
24          assertTrue(result is Success)
25          coVerify {
                authRepository.firebaseSignInWithEmailAndPassword(email,
                password) }
26      }
27
28      @Test
29      fun `sign in returns Failure when repository fails`() = runTest {
30          val exception = Exception("Auth error")
31          coEvery { authRepository.firebaseSignInWithEmailAndPassword(email,
                password) } returns Failure(exception)
32
33          val result = signInUserUseCase(email, password)
34
35          assertTrue(result is Failure)
36          coVerify {
                authRepository.firebaseSignInWithEmailAndPassword(email,
                password) }
37      }
38  }
```

Listado 4.1: SignInUserUseCaseTest

4.4.2. Pruebas de interfaz (UI)

Para comprobar el comportamiento de la interfaz de usuario, se ha utilizado la librería `androidx.compose.ui:ui-test-junit4`, integrada en Jetpack Compose. Esta herramienta permite verificar la presencia de elementos, sus estados, y simular acciones del usuario como clics o introducción de texto.

Frente a otras librerías tradicionales como Espresso, esta herramienta ofrece mejor sincronización con el ciclo de recomposición de Compose y facilita pruebas más declarativas [1].

4.4.3. Pruebas de integración

Las pruebas de integración validan que múltiples componentes trabajen juntos de forma coherente. En este proyecto se han utilizado:

- **Firebase Emulator Suite**, que simula los servicios de Firebase (Firestore y Auth) localmente [8].
- **Repositorios simulados** combinados con **Hilt** para sustituir dependencias reales durante las pruebas.

Este enfoque permite realizar pruebas realistas pero controladas, sin depender de servicios externos ni afectar datos en producción.

4.4.4. Pruebas end-to-end (E2E)

Las pruebas E2E verifican que la aplicación funciona correctamente en su conjunto, desde la interacción del usuario hasta el acceso a datos y navegación.

Se ha empleado **Espresso**, el framework oficial de Google para pruebas E2E en Android, junto con Firebase Emulator Suite para simular los servicios remotos. Estas pruebas son útiles para validar flujos completos como el inicio de sesión, la creación de listas o la adición de productos [1].

4.4.5. Cobertura de código

Como métrica adicional de calidad, se ha utilizado **JaCoCo** para generar informes de cobertura de código. Esto permite conocer qué líneas del proyecto han sido ejecutadas durante las pruebas y detectar zonas sin cobertura suficiente [4].

JaCoCo se integra fácilmente con Gradle y Android Studio, proporcionando estadísticas precisas que se han utilizado para ajustar el alcance de las pruebas implementadas.

Capítulo 5

Conclusiones y trabajos futuros

La realización de este trabajo me ha permitido conocer la complejidad y los beneficios de realizar un proyecto siguiendo buenas prácticas desde la definición a la implementación pasando por la etapa de diseño y el control del progreso.

Surgío de las dificultades que sufren equipos multidisciplinares digitales para comunicarse y entender las decisiones de cada capa. Parte fundamental para el trabajo de un desarrollador a parte de conocer, entender e implementar correctamente las tecnologias actuales y las buenas prácticas de generación de código. Es una habilidad que en cualquier empresa del sector o en cualquier proyecto que se realiza en grupo hay que cuidar y mejorar. (aplicar)

por supuesto aprender a implementar una aplicación nativa y el uso de tecnologías actuales.

Bibliografía

[1] Android Developers. Test apps on android. https://developer.android.com/training/testing, 2024. Consultado el 10 de diciembre de 2024.

[2] Design Thinking España. ¿qué es design thinking? https://designthinkingespaÃśa.com, 2024. Consultado el 10 de diciembre de 2024.

[3] Design Thinking España. ¿qué es design thinking? https://designthinkingespa%C3%B1a.com/#:~:text=El%20Design%20Thinking%20es%20una,muy%20poco%20tiempo%20soluciones%20innovadoras., 2024. Consultado el 10 de diciembre de 2024.

[4] EclEmma. Jacoco java code coverage library. https://www.eclemma.org/jacoco. Consultado el 10 de diciembre de 2024.

[5] Figma Inc. Figma: the collaborative interface design tool. https://www.figma.com, 2024. Consultado el 10 de diciembre de 2024.

[6] Google Firebase. Firebase authentication. https://firebase.google.com/docs/auth, 2024. Consultado el 10 de diciembre de 2024.

[7] Google Firebase. Firebase firestore documentation. https://firebase.google.com/docs/firestore, 2024. Consultado el 10 de diciembre de 2024.

[8] Google Firebase. Test your app with firebase emulator suite. https://firebase.google.com/docs/emulator-suite, 2024. Consultado el 10 de diciembre de 2024.

[9] E. Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Publishing Group, 2011.

[10] Scrum.org. What is scrum. https://www.scrum.org/learning-series/what-is-scrum/, 2024. Consultado el 10 de diciembre de 2024.

Apéndice A

Manual de usuario